

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

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

На тему: Розробка інструментальних засобів дослідження та контролю активних процесів операційної системи Windows

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

Студент гр. ІПЗ–23ск

_____ /Д. В. Бацко/

Керівник кваліфікаційної
роботи

_____ / Д. В. Швець /

В.о. завідувача кафедри

_____ / А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ–23ск Бацко Данилу Вячеславовичу

1. Тема: «Розробка інструментальних засобів дослідження та контролю активних процесів операційної системи Windows» затверджена наказом по КНУ № 283с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «31» травня 2026 р.
3. Вихідні дані по роботі: створене програмне забезпечення має забезпечити контроль активних процесів операційної системи Windows.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих на ринку аналогічних програмних продуктів, обґрунтувати функціонал розроблюваної системи, створити програмне забезпечення, здійснити тестування розробленого додатку.
5. Перелік ілюстративного матеріалу: функціональна схема, блок–схема алгоритму, зображення екранних форм додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	05.01.26 – 24.01.26
2	Аналіз існуючих методів вирішення проблеми	25.01.26 – 07.02.26
3	Формулювання актуальності роботи і постановка завдань	08.02.26 – 18.02.26
4	Оформлення матеріалів першого розділу роботи	18.02.26 – 02.03.26
5	Створення функціональної системи та алгоритму додатку	03.03.26 – 15.03.26
6	Оформлення матеріалів другого розділу роботи	16.04.26 – 09.04.26
7	Розробка баз даних, інтерфейсу програмного забезпечення, програмних модулів	10.04.26 – 01.05.26
8	Оформлення додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 30.05.26

Дата видачі завдання: «04» січня 2026 р.

Студент: _____ / Д. В. Бацко /

Керівник роботи: _____ / Д. В. Швець /

РЕФЕРАТ

ОПЕРАЦІЙНА СИСТЕМА WINDOWS, АКТИВНІ ПРОЦЕСИ, МОНІТОРИНГ, ПЛАТФОРМА .NET, ІНСТРУМЕНТАРІЙ УПРАВЛІННЯ WINDOWS, ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, C#, ЕКСПОРТ ДАНИХ.

Пояснювальна записка: 60 с., 12 рис., 1 табл., 1 дод., 20 джерел.

Об'єкт дослідження: процеси операційної системи Windows.

Предмет дослідження: інструментальні засоби та методи збору інформації, аналізу та контролю життєвого циклу процесів у середовищі Windows.

Розроблено додаток на базі сучасної платформи .NET із застосуванням мови програмування C#. Для реалізації графічного інтерфейсу користувача було обрано класичну технологію Windows Forms, що дозволило створити зручне та зрозуміле середовище для роботи. Архітектура програми базується на принципах об'єктно-орієнтованого програмування, зокрема на розділенні логіки на окремі шари: моделі даних, сервіси бізнес-логіки, допоміжні класи та форми інтерфейсу.

Реалізований функціонал охоплює отримання загального списку активних процесів із детальною інформацією про кожен з них, включаючи унікальний ідентифікатор, час запуску, шлях до виконуваного файлу та дані про компанію-розробника. Розроблений засіб містить механізми ведення журналу подій, фільтрації записів, а також пошуку процесів за ідентифікатором або частковим збігом назви. Для подальшого аналізу результатів реалізовано можливість експорту зібраних даних у формати електронних таблиць та текстових файлів із роздільниками, де враховано особливості локалізації та форматування даних.

ABSTRACT

WINDOWS OPERATING SYSTEM, ACTIVE PROCESSES, MONITORING, .NET PLATFORM, WINDOWS MANAGEMENT INSTRUMENTATION, OBJECT-ORIENTED PROGRAMMING, C#, DATA EXPORT.

Explanatory note: 60 pages, 12 figures, 1 table, 1 appendices, 20 sources.

Object of research: Windows operating system processes.

Subject of research: tools and methods for collecting information, analyzing, and controlling the lifecycle of processes in the Windows environment.

An application was developed based on the modern .NET platform using the C# programming language. To implement the graphical user interface, the classic Windows Forms technology was chosen, which allowed for the creation of a convenient and intuitive working environment. The program's architecture is based on the principles of object-oriented programming, in particular on the separation of logic into distinct layers: data models, business logic services, helper classes, and interface forms.

The implemented functionality covers obtaining a general list of active processes with detailed information about each of them, including the unique identifier, startup time, executable file path, and developer company data. The developed tool includes mechanisms for maintaining an event log, filtering records, and searching for processes by identifier or partial name match. For further analysis of the results, the ability to export the collected data into spreadsheet and delimited text file formats was implemented, taking into account localization and data formatting features.

ЗМІСТ

ВСТУП.....	7
1 АНАЛІЗ ПИТАНЬ КОНТРОЛЮ АКТИВНИХ ПРОЦЕСІВ ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS.....	9
1.1. Поняття процесу в операційній системі Windows.....	9
1.2. Архітектура керування процесами в Windows	10
1.3. Аналіз існуючих засобів моніторингу процесів	12
1.4. Постановка задачі розробки	15
2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ОТРИМАННЯ ТА ОБРОБКИ ІНФОРМАЦІЇ ПРО ПРОЦЕСИ WINDOWS.....	17
2.1 Технологія System.Diagnostics для роботи з процесами	17
2.2 Технологія Windows Management Instrumentation (WMI).....	18
2.3 Використання WinAPI для отримання додаткової інформації про процеси	20
3 РОЗРОБКА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДОСЛІДЖЕННЯ ТА КОНТРОЛЮ АКТИВНИХ ПРОЦЕСІВ ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS.....	23
3.1 Архітектура інструментальних засобів.....	23
3.2 Проєктування структури даних	24
3.2.1 Модель ProcessInfo	24
3.2.2 Модель EventLogEntry	25
3.3 Функціональна схема програми	26
3.4 Алгоритм роботи програми	28
3.5 Огляд роботи програми.....	29
ВИСНОВКИ	34
ПЕРЕЛІК ПОСИЛАНЬ	36
Додаток А – Фрагменти програмного коду	38

ВСТУП

Операційна система Windows керує тисячами потоків і сотнями активних процесів, розподіляючи між ними системні ресурси. Зростання складності програмного забезпечення та підвищення вимог до безпеки роблять моніторинг процесів та поведінки системи важливим завданням. Для адміністраторів, розробників та звичайних користувачів важливо розуміти, які програми запущені, які ресурси вони споживають, та мати змогу відстежувати їхній життєвий цикл. Існуючі рішення, такі як вбудований Диспетчер завдань, забезпечують базовий функціонал, однак для глибшого аналізу часто потрібні спеціалізовані інструменти, здатні вести журнал подій та експортувати дані для подальшої обробки.

Використання технології Windows Management Instrumentation (WMI) дозволяє отримувати детальну інформацію про стан системи та процеси за допомогою моніторингу подій [1]. Проте робота з такими подіями вимагає обережного підходу до проектування програмної архітектури, оскільки необхідно забезпечити асинхронне оновлення інтерфейсу та безпечну роботу з даними у фоновому потоці без блокування вікна програми. Платформа .NET надає необхідні механізми для розробки надійної шарової архітектури, що базується на принципах об'єктно-орієнтованого програмування. Таким чином, створення власного додатку для моніторингу є актуальною задачею.

Метою дипломної роботи є розробка інструментальних засобів для дослідження та контролю активних процесів операційної системи Windows з метою підвищення ефективності моніторингу програмного забезпечення.

Для досягнення поставленої мети було визначено такі завдання:

- а) проаналізувати теоретичну базу та існуючі методи управління процесами в ОС Windows;
- б) обґрунтувати вибір технологічного стеку та спроектувати архітектуру системи;
- в) розробити модуль отримання списку процесів та їхніх властивостей;

- г) реалізувати алгоритми відстеження запуску та завершення процесів;
- д) розробити графічний інтерфейс користувача та забезпечити його адаптивність і українську локалізацію;
- е) впровадити можливість збереження історії в журнал подій та експорту даних у формати CSV і Excel.

Об'єкт дослідження – процеси життєвого циклу активних програмних компонентів в операційній системі Windows.

Предмет дослідження – методи та інструментальні засоби отримання, обробки й відображення інформації про активні процеси ОС Windows на платформі .NET.

У роботі застосовуються методи системного аналізу для формування вимог до програмного забезпечення, методи об'єктно-орієнтованого проектування для розробки архітектури додатку, а також методи подієво-орієнтованого програмування для обробки системних сповіщень і взаємодії між модулями.

1 АНАЛІЗ ПИТАНЬ КОНТРОЛЮ АКТИВНИХ ПРОЦЕСІВ ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

1.1. Поняття процесу в операційній системі Windows

В архітектурі сучасних обчислювальних систем програма розглядається як пасивний набір інструкцій, тоді як процес – це безпосереднє виконання цих інструкцій, об'єднане з необхідними системними ресурсами. В операційній системі Windows процес виступає як своєрідний контейнер, що містить віртуальний адресний простір, код, дані, а також маркери доступу та таблицю описувачів об'єктів. Відповідно, моніторинг операційного середовища вимагає ефективних механізмів отримання інформації про ці контейнери.

Життєвий цикл процесу в ОС Windows складається з кількох етапів: створення, готовність до виконання, безпосереднє виконання, очікування та завершення [2]. Спостереження за переходами між цими станами є важливою складовою системної діагностики та контролю за стабільністю роботи комп'ютера. Для фіксації етапів створення та завершення на рівні системи існують спеціалізовані інструменти, зокрема технологія WMI. Зокрема, підписка на системні події дозволяє операційній системі генерувати сповіщення при зміні стану будь-якого процесу [3]. Збирання таких подій дає змогу вести журнали для подальшого аналізу поведінки програмного забезпечення.

Кожен процес описується низкою унікальних характеристик. Основним ідентифікатором є PID (Process ID), який дозволяє ОС відрізнити один екземпляр виконуваної програми від іншого. Крім того, важливими атрибутами є час старту, розташування виконуваного файлу на диску, а також додаткові метадані, такі як оригінальна назва компанії-розробника та версія продукту. Аналіз цих характеристик допомагає виявляти підозрілу активність, контролювати споживання ресурсів та здійснювати пошук конкретних процесів серед сотень запущених у системі компонентів.

Сам по собі процес в ОС Windows не виконує код; він лише надає ізольоване середовище. Фактичне виконання команд покладається на потоки (threads) [4]. Один процес завжди містить щонайменше один головний потік, але заради продуктивності може генерувати додаткові потоки для паралельної роботи. Під час розробки інструментів моніторингу розуміння взаємодії між потоками є надзвичайно важливим. Зокрема, збір інформації від операційної системи зазвичай відбувається у фоновому середовищі. Тому безпечна передача даних про стан процесів до графічного інтерфейсу користувача вимагатиме обережної синхронізації, щоб уникнути блокування вікон програм під час інтенсивного обміну даними між потоками.

1.2. Архітектура керування процесами в Windows

Операційна система Windows використовує модель витискальної багатозадачності для керування виконанням програм. Згідно з цією моделлю, планувальник ядра операційної системи самостійно розподіляє процесорний час між активними потоками. Сам процес у цій архітектурі виступає лише як контейнер, що володіє віртуальним адресним простором та системними ресурсами, тоді як фактичне виконання інструкцій здійснюється його потоками. Завдяки такому підходу система гарантує, що жодна програма не зможе монополізувати процесор. Якщо певний потік зависає або потребує тривалого очікування на операції вводу-виводу, планувальник призупиняє його і передає керування іншому. Це створює ілюзію паралельного виконання багатьох завдань навіть на одноядерних процесорах та забезпечує стабільність роботи всього комп'ютера.

Архітектура безпеки Windows поділяє всі процеси на дві великі категорії залежно від режиму їх виконання та рівня привілеїв: системні та користувацькі. Такий поділ є основою для захисту критично важливих компонентів від випадкового або навмисного втручання з боку прикладних програм.

Системні процеси (наприклад, smss.exe, csrss.exe, lsass.exe або ядро System) виконують базові функції обслуговування системи. Вони часто працюють у привілейованому режимі (режимі ядра) або мають найвищі права доступу. Їхнє випадкове завершення або збій зазвичай призводить до нестабільності або аварійної зупинки ОС.

Користувацькі процеси – це звичайні прикладні програми (браузери, текстові редактори, антивіруси), які запускаються від імені поточного користувача. Вони працюють в ізольованому режимі з обмеженими правами. Щоб отримати доступ до апаратного забезпечення або виділити додаткову пам'ять, такі процеси повинні робити безпечні запити до ядра системи. Завдяки цій ізоляції аварійне завершення користувацького процесу не шкодить операційній системі в цілому.

Для однозначної ідентифікації кожного активного контейнера виконання в системі використовується ідентифікатор процесу PID. Під час запуску будь-якої програми операційна система генерує та присвоює їй унікальне числове значення [5]. PID використовується внутрішніми механізмами Windows та діагностичними інструментами для точного звернення до конкретного процесу, наприклад, для зчитування його властивостей або для примусового завершення.

Важливо зазначити, що PID є тимчасовим значенням. Він залишається закріпленим за процесом лише до моменту його повного завершення. Після того, як програма закривається, її ідентифікатор вивільняється, і через певний час операційна система може призначити це ж числове значення абсолютно іншому процесу.

Хоча кожен процес у Windows працює у власному ізольованому просторі, між ними існують родинні зв'язки, що формують ієрархічну структуру – дерево процесів. Коли одна програма ініціює запуск іншої, перша стає "батьківським" процесом (Parent Process), а друга – "дочірнім" (Child Process). Операційна система фіксує ідентифікатор батьківського процесу (Parent PID) у властивостях дочірнього під час його створення.

Проте, на відміну від деяких інших сімейств операційних систем (наприклад, Unix-подібних), у Windows цей зв'язок є переважно інформаційним. Якщо батьківський процес завершується, дочірні процеси продовжують своє виконання незалежно від нього, перетворюючись на самостійні одиниці. Аналіз дерева процесів є корисним методом під час дослідження поведінки програмного забезпечення, оскільки він дозволяє встановити ланцюжок подій і зрозуміти, яка саме програма ініціювала запуск певного компонента.

1.3. Аналіз існуючих засобів моніторингу процесів

Щоб спроектувати ефективну систему відстеження стану операційного середовища, необхідно проаналізувати вже наявні на ринку рішення. Найбільш відомими аналогами є стандартний Диспетчер завдань (Task Manager), а також спеціалізовані утиліти Process Explorer та Process Hacker. Кожна з цих програм має свої переваги та обмеження, дослідження яких дозволяє краще зрозуміти потреби користувачів та сформулювати вимоги до нових розробок у цій галузі.

Task Manager є вбудованим компонентом операційної системи Windows. Він забезпечує базовий рівень моніторингу, дозволяючи користувачам переглядати список активних програм та фонових служб, а також оцінювати споживання апаратних ресурсів: процесорного часу, оперативної пам'яті, дискового простору та мережі [6].

У сучасних версіях Windows цей інструмент отримав оновлений інтерфейс, проте його основне призначення залишається незмінним – швидке реагування на зависання програм (наприклад, через примусове закриття) та базовий контроль швидкодії комп'ютера. Головним недоліком Task Manager є відсутність механізмів глибокого ведення журналу подій у реальному часі та неможливість зручного автоматизованого експорту зібраних даних для їхнього подальшого аналізу.

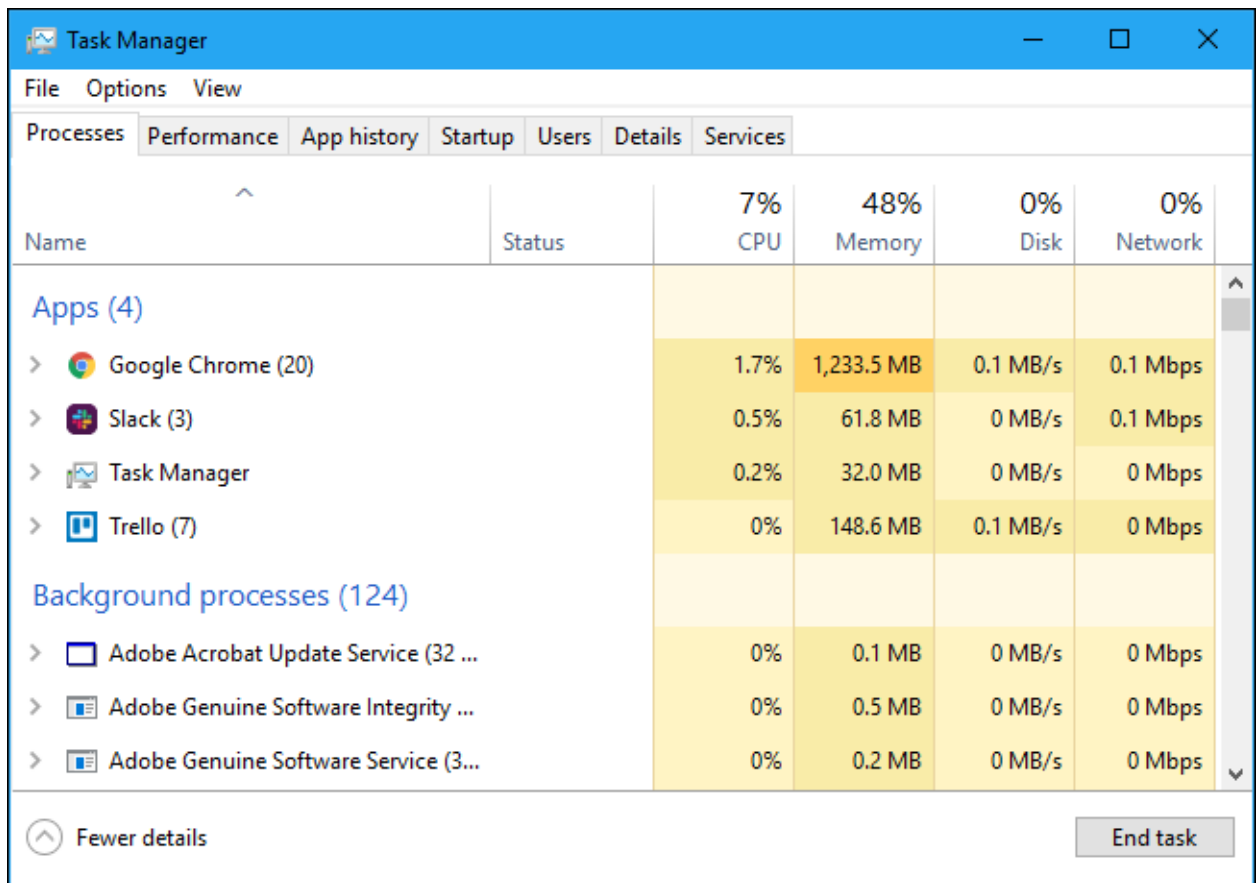


Рисунок 1.1 – Task Manager

Process Explorer [7] – це потужна утиліта з пакету Sysinternals Suite, що офіційно підтримувана компанією Microsoft. На відміну від стандартного Диспетчера завдань, вона відображає ієрархічне дерево запущених компонентів, що дозволяє легко відстежувати родинні зв'язки між батьківськими та дочірніми програмами. Додаток надає вичерпну інформацію про відкриті дескриптори та завантажені динамічні бібліотеки, що робить його незамінним для пошуку витоків пам'яті або вирішення конфліктів доступу до файлів.

Водночас інтерфейс Process Explorer може бути занадто перевантаженим для завдань, де вимагається лише відстеження часу запуску, завершення роботи та проста фільтрація інформації. Крім того, ця утиліта фокусується переважно на живому перегляді, не маючи вбудованої простої функції періодичного архівування стану системи у розповсюджені табличні формати.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Autostart Location	Verified Signer	Process Timeline	Integrity	VirusTotal
System Idle Process	85.99	0 K	4 K	0						System	
System	0.72	180 K	6,244 K	4						System	
smss.exe	0.87	0 K	0 K	n/a	Hardware Interrupts and DPCs					System	
Memory Compression		376 K	504 K	364	Windows Session Manager	Microsoft Corporation		(Verified) Microsoft		System	
csrss.exe		716 K	55,848 K	2672				(Verified) Microsoft		System	
svchost.exe	< 0.01	1,944 K	2,088 K	528	Client Server Runtime Process	Microsoft Corporation		(Verified) Microsoft		System	
svchost.exe	< 0.01	1,444 K	3,020 K	644	Windows Start-Up Application	Microsoft Corporation		(Verified) Microsoft		System	
svchost.exe	< 0.01	4,356 K	5,748 K	788	Services and Controller app	Microsoft Corporation		(Verified) Microsoft		System	
svchost.exe	< 0.01	17,372 K	20,016 K	892	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
WinPvSE.exe		3,548 K	7,208 K	5932	WMI Provider Host	Microsoft Corporation		(Verified) Microsoft		System	
RuntimeBroker.exe		17,944 K	46,132 K	4020	Runtime Broker	Microsoft Corporation		(Verified) Microsoft		Medium	
ShellExperienceHost.exe	0.29	47,824 K	89,584 K	4956	Windows Shell Experience H...	Microsoft Corporation		(Verified) Microsoft		AppContainer	
SearchUI.exe	Susp...	68,488 K	1,15,868 K	10196	Search and Cortana applicat...	Microsoft Corporation		(Verified) Microsoft		AppContainer	
SkypeHost.exe	Susp...	32,496 K	10,984 K	7300	Microsoft Skype Preview	Microsoft Corporation		(No signature was...		AppContainer	
SettingSyncHost.exe		14,804 K	5,176 K	5164	Host Process for Setting Syn...	Microsoft Corporation		(Verified) Microsoft		Medium	
CastSrv.exe		3,000 K	5,820 K	3740	Casting protocol connection I...	Microsoft Corporation		(Verified) Microsoft		Medium	
dhcpcd.exe		3,420 K	10,472 K	12084	COM Surrogate	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		Medium	
ApplicationFrameHost.exe		4,716 K	19,752 K	5284	Application Frame Host	Microsoft Corporation		(Verified) Microsoft		Medium	
SystemSettingsBroker.exe		4,892 K	20,084 K	7940	System Settings Broker	Microsoft Corporation		(Verified) Microsoft		Medium	
smartscreen.exe		8,316 K	13,828 K	9424	SmartScreen	Microsoft Corporation		(Verified) Microsoft		Medium	
BackgroundTaskHost.exe	Susp...	4,736 K	17,252 K	4256	Background Task Host	Microsoft Corporation		(Verified) Microsoft		AppContainer	
svchost.exe		8,652 K	10,356 K	964	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
svchost.exe	< 0.01	1,10,636 K	95,948 K	524	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
WUDFHost.exe		1,756 K	1,824 K	1116	Windows Driver Foundation ...	Microsoft Corporation		(Verified) Microsoft		System	
dashost.exe		972 K	356 K	2244	Device Association Framewo...	Microsoft Corporation		(Verified) Microsoft		System	
svchost.exe	0.01	18,796 K	20,540 K	852	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
svchost.exe	< 0.01	13,832 K	17,616 K	396	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
svchost.exe		19,216 K	26,648 K	1104	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
svchost.exe	0.01	68,396 K	75,216 K	1292	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
iphcst.exe		6,212 K	23,852 K	200	Shell Infrastructure Host	Microsoft Corporation		(Verified) Microsoft		Medium	
taskhostw.exe		7,368 K	21,000 K	12112	Host Process for Windows T...	Microsoft Corporation		(Verified) Microsoft		Medium	
mpoia.exe		6,580 K	9,936 K	4732	Office Telemetry Agent	Microsoft Corporation	Task Scheduler\...	(Verified) Microsoft		Medium	
SIHClient.exe	< 0.01	1,744 K	4,692 K	8680	SIH Client	Microsoft Corporation	Task Scheduler\...	(Verified) Microsoft		System	
conhost.exe	0.01	1,356 K	2,740 K	652	Console Window Host	Microsoft Corporation		(Verified) Microsoft		System	
gfhCUIService.exe		2,680 K	4,624 K	1392	gfhCUIService Module	Intel Corporation	HKLM\System\Cu...	(Verified) Intel(R) p...		System	
svchost.exe		2,948 K	6,540 K	1576	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	
audodg.exe	6.76	85,000 K	72,148 K	840	Windows Audio Device Grap...	Microsoft Corporation		(Verified) Microsoft		System	
RtkAudioService64.exe		1,956 K	2,964 K	1680	Realtek Audio Service	Realtek Semiconductor	HKLM\System\Cu...	Verifying...		System	
RAVBg64.exe		5,940 K	14,288 K	11144	HD Audio Background Proc...	Realtek Semiconductor		Verifying...		System	
RAVBg64.exe		5,624 K	13,724 K	424	HD Audio Background Proc...	Realtek Semiconductor		Verifying...		System	
svchost.exe	< 0.01	28,888 K	32,272 K	1808	Host Process for Windows S...	Microsoft Corporation	HKLM\System\Cu...	(Verified) Microsoft		System	

CPU Usage: 14.01% Commit Charge: 38.32% Processes: 104 Physical Usage: 41.74%

Рисунок 1.2 – Process Explorer

Process Hacker [8] (який у нових версіях розвивається під назвою System Informer) – це безкоштовний проект із відкритим вихідним кодом. Завдяки наявності власного драйвера рівня ядра ця програма дозволяє обходити обмеження стандартного користувацького режиму та забезпечує безпрецедентний рівень контролю: від прямого редагування віртуальної пам'яті до аналізу мережеских з'єднань кожного окремого компонента. Цей інструмент широко застосовується фахівцями з кібербезпеки для виявлення шкідливого програмного забезпечення та руткітів.

Однак така глибина доступу вимагає специфічних знань і робить додаток надлишково складним для звичайного використання. Інтерфейс вимагає від користувача глибокого розуміння внутрішньої архітектури Windows, а велика кількість вкладок і таблиць ускладнює пошук базових властивостей.

Проведений аналіз показує, що хоча існуючі інструменти володіють значним функціоналом, на ринку залишається потреба в рішеннях, які

поєднують простоту використання з можливістю автоматизованого ведення історії подій (наприклад, через реєстрацію системних сповіщень WMI).

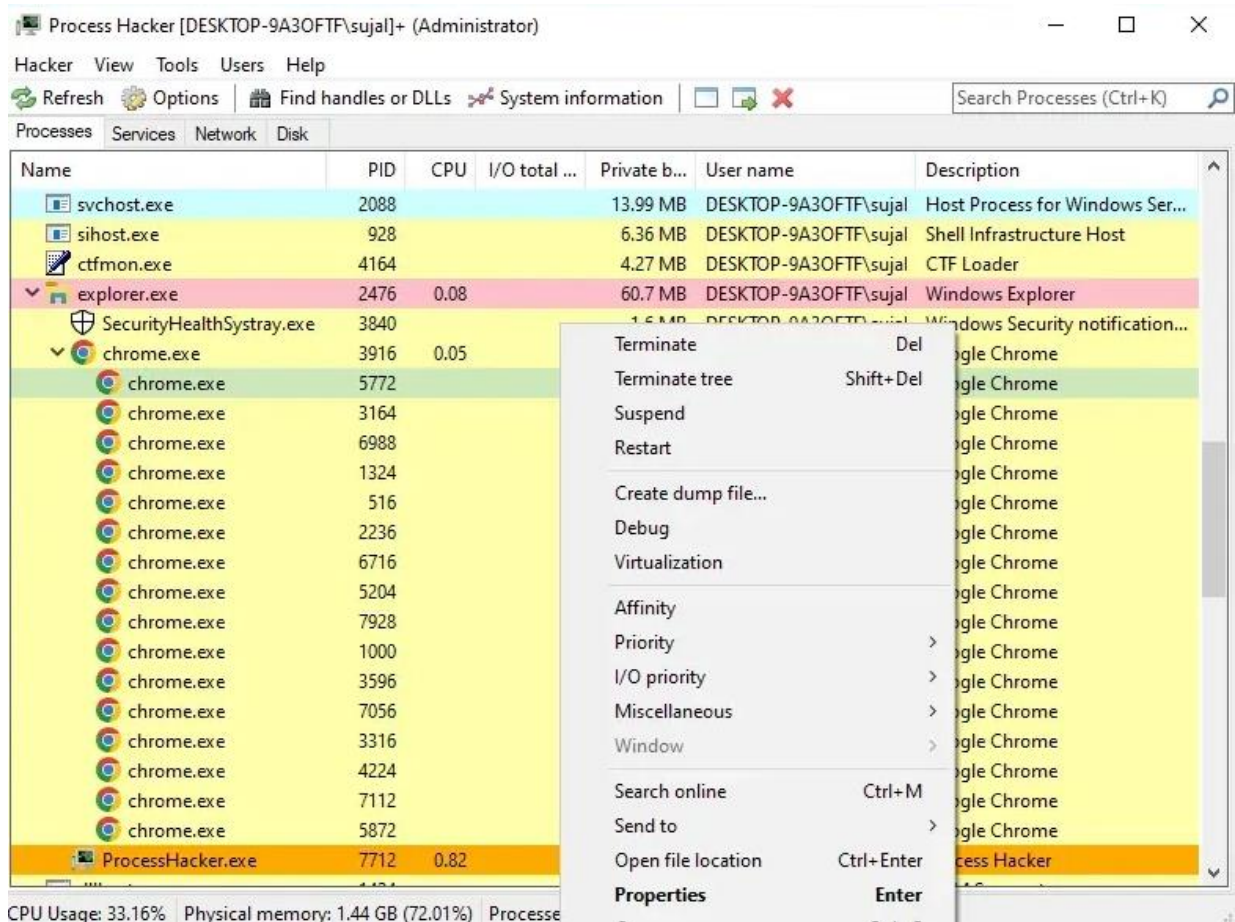


Рисунок 1.3 – Process Hacker

Оптимальний інструмент повинен мати зручний графічний інтерфейс, гнучкі механізми пошуку та можливість експортування результатів у формати CSV або Excel для спрощення складання звітів, уникаючи при цьому перевантаження користувача зайвою низькорівневою інформацією.

1.4. Постановка задачі розробки

Успішна реалізація інструментів діагностики операційної системи вимагає попереднього формування детального переліку технічних умов. Програмний засіб, що проектується для моніторингу, повинен забезпечувати комплексний контроль активних компонентів Windows, збирати телеметричні

дані та надавати користувачеві зручний графічний інтерфейс для подальшого аналізу. Основне завдання полягає у створенні надійного рішення, яке поєднуватиме можливості перегляду поточного стану операційного середовища з веденням журналу подій, уникаючи при цьому надлишкової складності, властивій професійним утилітам для кібербезпеки.

Відповідно до теоретичних засад розробки, система повинна виконувати визначений набір операцій, що забезпечать досягнення поставленої мети [9]. До основних функціональних вимог майбутнього додатка належать:

- а) отримання та відображення актуального списку процесів із їхніми детальними властивостями, серед яких ідентифікатор PID, час запуску, повний шлях до виконуваного файлу, інформація про компанію-розробника та версію продукту;

- б) відстеження подій запуску та зупинки програм на основі системних сповіщень (зокрема, через механізми підписки WMI);

- в) ведення журналу подій у пам'яті для ретроспективного аналізу змін у системі;

- г) забезпечення механізмів фільтрації та пошуку потрібної інформації за ідентифікатором або частковим збігом назви;

- д) експортування зібраних даних у зовнішні файли (у формати CSV з правильним кодуванням та електронні таблиці Excel) для зручного складання звітів.

Отже, формування функціональних вимог визначає напрямки для подальшого проектування програми. Реалізація заявлених можливостей, зокрема отримання розширених властивостей процесів та відстеження їхнього життєвого циклу за допомогою подій WMI, дозволить створити додаток на платформі .NET [10]. Важливою умовою є збереження зібраної історії та надання користувачеві зручних засобів фільтрації й експорту даних. Сформований перелік завдань описує необхідний функціонал і є достатнім для переходу до етапу проектування архітектури програми та розробки візуальних компонентів.

2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ОТРИМАННЯ ТА ОБРОБКИ ІНФОРМАЦІЇ ПРО ПРОЦЕСИ WINDOWS

2.1 Технологія System.Diagnostics для роботи з процесами

Простір імен System.Diagnostics у базовій бібліотеці класів .NET [11] забезпечує розробників набором інструментів для взаємодії із системними ресурсами, журналами подій та лічильниками продуктивності. Для проєктованого додатку цей простір імен стане базовим компонентом для реалізації основного функціоналу. Його інструментарій дозволяє організувати ефективний обмін даними між операційною системою та графічним інтерфейсом користувача без необхідності безпосереднього звернення до низькорівневих функцій Windows API.

Основним елементом для роботи з активними програмами в керованому коді є клас Process. Він надає доступ до локальних і віддалених процесів, дозволяючи зчитувати їхні параметри [12] та керувати їхнім виконанням. У розроблюваному програмному засобі об'єкти цього класу виступатимуть основним джерелом даних на рівні моделей у межах обраної архітектури. Дотримуючись принципів об'єктно-орієнтованого програмування, інформація, отримана з класу Process, буде інкапсульована у власні типи даних додатка для подальшої обробки та відображення.

Для отримання актуального переліку всіх програм, запущених у системі, використовується статичний метод Process.GetProcesses(), який повертає масив відповідних об'єктів. Під час програмної реалізації системи цей метод буде використано для первинного збору даних і заповнення головної таблиці. Оскільки зчитування масиву інформації про сотні компонентів може зайняти певний час, отримання списку процесів відбуватиметься у фоновому потоці.

Кожен екземпляр класу Process містить набір властивостей, що описують поточний стан відповідної програми. Проєктований додаток збиратиме таку інформацію: ідентифікатор PID, назву, час запуску та повний

шлях до виконуваного файлу на диску. Крім зчитування базових атрибутів, у системі буде реалізовано отримання розширених метаданих [13] за допомогою допоміжного класу FileVersionInfo. Це дозволить видобувати інформацію про оригінальну компанію-розробника та версію продукту.

Попри значні можливості, робота з системними класами має певні обмеження, безпосередньо пов'язані з політиками безпеки операційної системи. Спроба отримати доступ до властивостей системних або захищених процесів (наприклад, їхнього шляху або часу запуску) часто призводить до генерації виняткових ситуацій. Тому під час розробки інструментальних засобів буде імплементовано обробку винятків на рівні допоміжних класів (Helpers). При неможливості зчитати певний параметр програма не перериватиме свою роботу, а безпечно перехоплюватиме помилку, залишаючи відповідне поле порожнім або виводячи інформаційне повідомлення замість аварійного закриття додатка.

2.2 Технологія Windows Management Instrumentation (WMI)

Інструментарій управління Windows (WMI) – це базова інфраструктура для керування даними та операціями в операційних системах від компанії Microsoft. Її створено на основі загальноприйнятих стандартів адміністрування корпоративних мереж [14]. Головне призначення цієї технології полягає у наданні розробникам стандартизованого механізму для доступу до системних ресурсів, конфігурацій та телеметричних даних.

Архітектура цієї інфраструктури побудована на трьох основних компонентах: постачальниках даних (які збирають інформацію з обладнання або ядра операційної системи), диспетчері об'єктів, що координує запити, та споживачах – утилітах або скриптах, які обробляють зібрану інформацію [15]. Розроблювана система виступатиме саме в ролі споживача даних.

Уся інформація в межах інфраструктури управління організована у вигляді просторів імен, які діють як своєрідні каталоги для різних схем адміністрування. Найбільш поширеним є простір `root\CIMV2`, що містить

основні класи для роботи з апаратним забезпеченням та операційною системою. Інший важливий шлях – `root\WMI`, який часто використовується для зчитування специфічних подій із ядра. Проектована система для моніторингу процесів підключатиметься до стандартного простору `root\CIMV2` для конструювання запитів. Це забезпечить надійний доступ до необхідних об'єктів без застосування складних додаткових налаштувань політик безпеки.

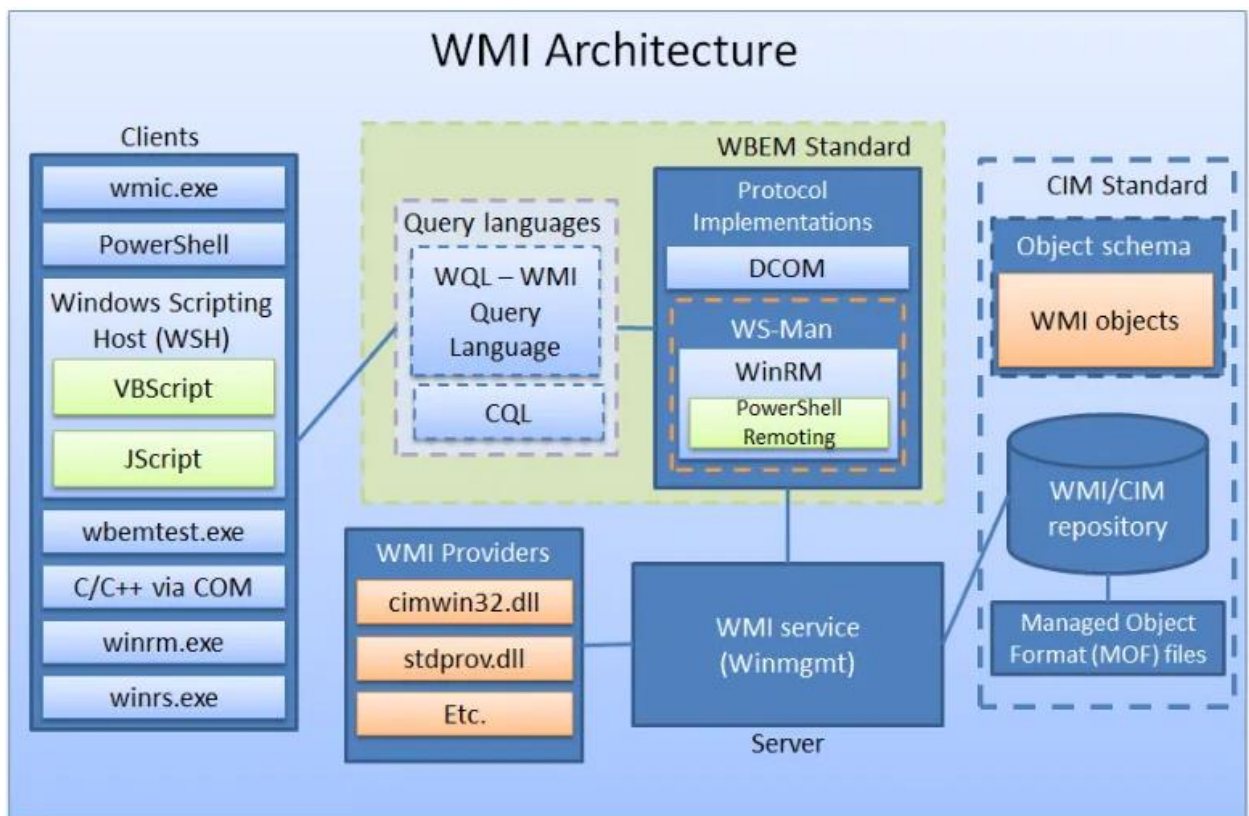


Рисунок 2.1 – Архітектура WMI [17]

Серед великої кількості класів системного інструментарію існують спеціалізовані рішення для роботи з програмами. Зокрема, виділяють базові об'єкти (наприклад, `Win32_Process`), а також спеціальні структури для відстеження подій у часі [18].

Оскільки сповіщення від операційної системи надходитимуть незалежно від дій користувача, їхня первинна обробка виконуватиметься у фоновому

потоці. Інформація про кожний новий запуск чи завершення роботи зберігатиметься в оперативній пам'яті, формуючи локальний журнал подій. Щоб уникнути конфліктів під час взаємодії з графічним інтерфейсом, передача цих записів до таблиці відображення здійснюватиметься через безпечне асинхронне оновлення віконних елементів за допомогою методів `Invoke` або `BeginInvoke`.

2.3 Використання WinAPI для отримання додаткової інформації про процеси

Windows Application Programming Interface (WinAPI) являє собою базовий набір інтерфейсів програмування, який надає прикладним програмам прямий доступ до функцій ядра операційної системи, системного реєстру, апаратного забезпечення та елементів інтерфейсу користувача. Хоча платформа .NET пропонує потужний інструментарій (зокрема простори імен для роботи з діагностикою), деякі специфічні системні дані або операції не мають повноцінних аналогів у стандартному керованому коді. Наприклад, видобування оригінальної піктограми виконуваного файлу, глибший аналіз внутрішніх дескрипторів або отримання специфічних параметрів вікон процесів вимагає звернення безпосередньо до низькорівневих системних бібліотек, таких як `kernel32.dll` або `user32.dll`.

Взаємодія сучасних додатків, написаних мовою C#, із нативними бібліотеками операційної системи реалізується через спеціальний механізм Platform Invocation Services (скорочено P/Invoke) [19]. Оскільки кероване середовище виконання .NET ізолює програміста від ручного управління пам'яттю, цей механізм бере на себе завдання правильного перетворення (маршалінгу) типів даних між безпечним керованим кодом та некерованими функціями, що написані мовами C або C++. Під час проектування архітектури майбутнього додатку взаємодія з нативним системним інтерфейсом буде інкапсульована в окремих класах на рівні допоміжних компонентів. Це дозволить зберегти загальну шарову структуру системи чистою,

дотримуючись принципу інверсії залежностей (DIP), та надійно відокремити логіку від специфіки платформи.

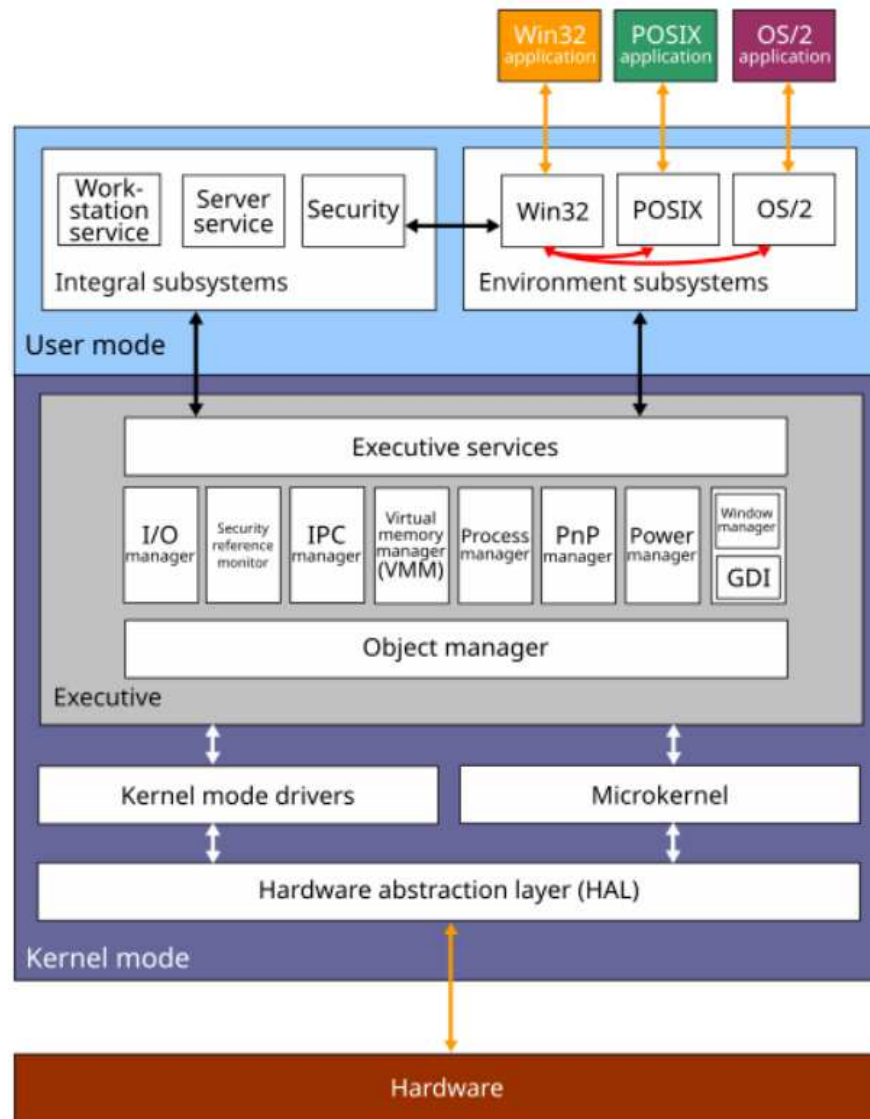


Рисунок 2.2 – Схема використання WinAPI [20]

Застосування зовнішніх системних функцій вимагає їхнього попереднього оголошення в коді за допомогою атрибута `DllImport`. Під час розробки програмного засобу звернення до таких низькорівневих процедур проектується з підвищеною обережністю. Оскільки звернення до нативного коду може спричинити помилки доступу на рівні ядра, додаток міститиме надійні механізми перехоплення виняткових ситуацій. Наприклад, специфічні

системні збої будуть оброблятися та перетворюватися у стандартний Win32Exception, уникаючи аварійного закриття програми.

Крім того, будь-які тривалі операції збору розширеної інформації через зовнішні бібліотеки будуть ініціюватися виключно у фоновому потоці. Після отримання та обробки результатів вони передаватимуться до графічного інтерфейсу через безпечне асинхронне оновлення віконних елементів, що гарантуватиме безперебійну чуйність системи під час активного моніторингу.

3 РОЗРОБКА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДОСЛІДЖЕННЯ ТА КОНТРОЛЮ АКТИВНИХ ПРОЦЕСІВ ОПЕРАЦІЙНОЇ СИСТЕМИ WINDOWS

3.1 Архітектура інструментальних засобів

Архітектура програмного забезпечення визначає його базову організацію, розподіл відповідальності між основними компонентами та правила їхньої взаємодії. Для розробленого програмного засобу було обрано багаторівневу структуру, яка є класичним рішенням для побудови додатків на платформі .NET.

Використання мови програмування C# дозволило реалізувати систему з дотриманням ключових принципів об'єктно-орієнтованого програмування, таких як інкапсуляція, принцип єдиної відповідальності, принцип відкритості/закритості та інверсія залежностей.

Структурно розроблений додаток організовано у вигляді кількох взаємопов'язаних логічних блоків, кожен з яких ізолює певну частину коду. Першим з них є блок представлення предметної області, де зосереджені класи для збереження зібраної інформації. Саме тут інкапсульовані структури, що описують поточний стан кожного процесу, його ідентифікатор PID, шлях до виконуваного файлу, а також додаткові метадані (зокрема, оригінальна версія та компанія-розробник), які отримуються за допомогою інструменту FileVersionInfo. Управління цими даними здійснює ядро бізнес-логіки. Цей центральний модуль відповідає за безпосередню взаємодію з операційною системою через простори імен діагностики та забезпечує обробку системних подій, що надходять від інфраструктури WMI.

Окрім основної логіки, архітектура містить інфраструктурний компонент, який об'єднує додатковий інструментарій. У ньому реалізовано алгоритми безпечної обробки системних винятків, а також утиліти для експорту накопичених даних у зовнішні файли за допомогою бібліотеки

ClosedXML версії 0.104.2. Зв'язок із кінцевим користувачем забезпечує візуальний блок, побудований на базі технології Windows Forms.

Цей модуль гарантує зручну взаємодію з користувачем завдяки застосуванню адаптивних компонентів для компоновання вікон.

3.2 Проєктування структури даних

Програмний продукт потребує організації інформації, оскільки архітектура зберігання даних безпосередньо впливає на продуктивність та надійність усієї системи.

На етапі проєктування розробленого додатка було визначено необхідність створення власних класів на рівні моделей. Вони слугуватимуть спеціалізованими контейнерами для збереження поточного стану компонентів операційної системи та історії їхньої активності.

3.2.1 Модель ProcessInfo

В архітектурі розробленого програмного засобу клас ProcessInfo виконує роль об'єкта передачі даних (Data Transfer Object).

Головне призначення цієї моделі полягає у забезпеченні логічної ізоляції графічного інтерфейсу від системних функцій платформи .NET. Замість прямої роботи з об'єктами System.Diagnostics.Process, рівень логіки зчитує потрібну інформацію і заповнює екземпляри ProcessInfo. Це дозволяє уникнути помилок доступу до ресурсів, коли візуальні компоненти намагаються відобразити властивості програм, які вже завершили свою роботу або були заблоковані політикою безпеки операційної системи.

Модель спроектована з використанням принципів інкапсуляції, тому всі атрибути реалізовані у вигляді властивостей.

Для кращого сприйняття та систематизації інформації про структуру класу, детальний опис його ключових полів наведено у таблиці 3.1.

Таблиця 3.1 – Опис полів моделі ProcessInfo

Назва поля	Тип даних	Призначення
Id	Цілочисельний (int)	Ідентифікатор PID, який операційна система призначила під час запуску
ProcessName	Рядковий (string)	Коротка назва виконуваного файлу без розширення
StartTime	Дата та час (DateTime)	Фіксує момент початку виконання активного компонента
FilePath	Рядковий (string)	Повний шлях до розташування виконуваного файлу на жорсткому диску
CompanyName	Рядковий (string)	Назва компанії-розробника, отримана з розширених метаданих
ProductVersion	Рядковий (string)	Оригінальна версія програмного продукту, отримана за допомогою FileVersionInfo

3.2.2 Модель EventLogEntry

Для організації моніторингу життєвого циклу процесів спроектовано модель EventLogEntry. Вона призначена для фіксації кожної зміни стану операційного середовища, про яку повідомляє технологія WMI. Її структура значно простіша за модель процесу і містить: мітку часу (момент надходження події), тип операції (наприклад, «Запущено» або «Завершено»), ідентифікатор PID та назву програми. Такий набір полів гарантує швидке створення нових екземплярів класу навіть за умови одночасної активності десятків компонентів.

Головною особливістю підсистеми журналювання є те, що всі записи зберігаються в оперативній пам'яті. Для цього використовується типізована колекція. Оскільки нові екземпляри додаються до колекції в результаті спрацювання підписок системних сповіщень (що відбувається у фоновому

потоці), операції запису вимагають обережної синхронізації. Щоб інформація з журналу могла відображатися на екрані, колекція передається до табличного елемента керування через безпечне асинхронне оновлення інтерфейсу. Це гарантує цілісність даних і запобігає блокуванню вікон додатка. Згодом накопичений масив об'єктів використовується рівнем допоміжних класів для експортування результатів у формати CSV (у кодуванні UTF-8) або в електронні таблиці Excel.

3.3 Функціональна схема програми

Для опису взаємодії компонентів у часі під час проєктування розробленого програмного засобу використано діаграму послідовності. Цей інструмент дозволяє проілюструвати обмін повідомленнями між різними логічними шарами архітектури під час виконання користувацьких та системних сценаріїв.

Функціональна логіка додатка базується на трьох основних сценаріях взаємодії. Першим із них є процес ініціалізації та первинного опитування операційного середовища. Коли користувач запускає програму, візуальний інтерфейс відправляє запит на формування списку до модулів логіки. Логічний шар здійснює звернення до системного інструментарію `System.Diagnostics.Process` у фоновому потоці. Отримавши масив даних від ядра ОС, система перехоплює можливі помилки доступу до захищених ресурсів та конвертує системні структури у власну колекцію об'єктів `ProcessInfo`. Завершується цей етап передачею підготовленої інформації назад до візуального шару через механізм оновлення таблиці.

Другий сценарій – це режим моніторингу подій. Його життєвий цикл починається з того, що інтерфейс активує відповідний функціонал, передаючи команду до служб. Шар логіки оформлює підписку на події інфраструктури WMI. Після цього додаток переходить у стан фонового очікування. Щойно операційна система генерує сповіщення про зміну стану якогось із процесів, логічний модуль опрацьовує цю інформацію, реєструючи подію в оперативній

пам'яті. Відразу після цього ініціюється оновлення візуального журналу, інформуючи користувача про зміни.

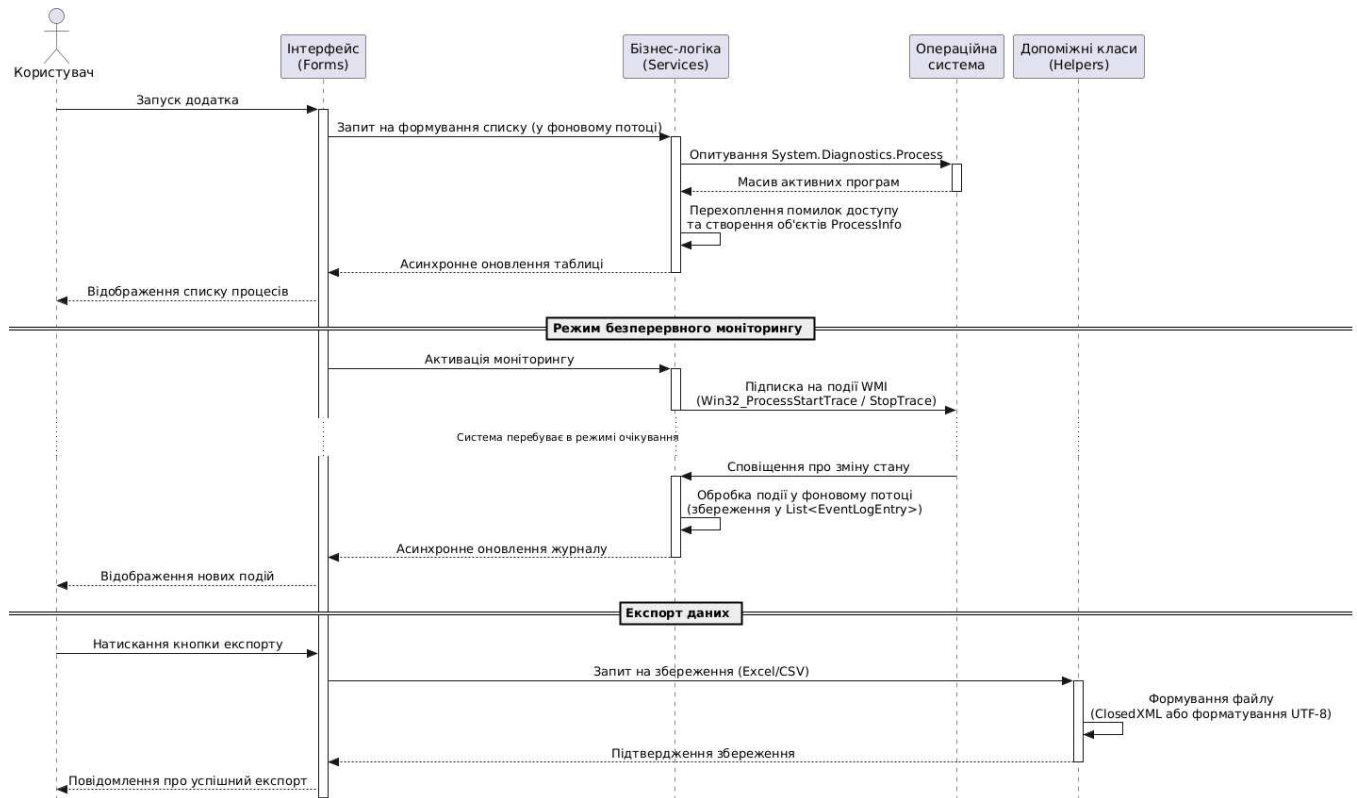


Рисунок 3.1 - Функціональна схема програми

Третій процес описує логіку експорту накопичених телеметричних даних. Ініціатором операції є користувач, який натискає кнопку збереження в інтерфейсі. Візуальний модуль перенаправляє цей запит до шару допоміжних класів. Залежно від обраного формату, допоміжний модуль виконує формування електронної таблиці Excel (за допомогою засобів ClosedXML з автоналаштуванням параметрів стовпців) або генерує текстовий файл CSV (із додаванням роздільників та застосуванням кодування UTF-8). Після запису файлу на жорсткий диск, допоміжний шар повертає підтвердження, а користувач отримує відповідне повідомлення в головному вікні додатка.

3.4 Алгоритм роботи програми

Після старту додатка відбувається ініціалізація базових налаштувань та візуальних компонентів. Наступним кроком є первинне опитування операційної системи для отримання списку наявних процесів. Отримані дані відфільтровуються від системних помилок та передаються до таблиць через оновлення інтерфейсу. Далі алгоритм передбачає активацію моніторингу: система оформлює підписку на події інструментарію WMI.

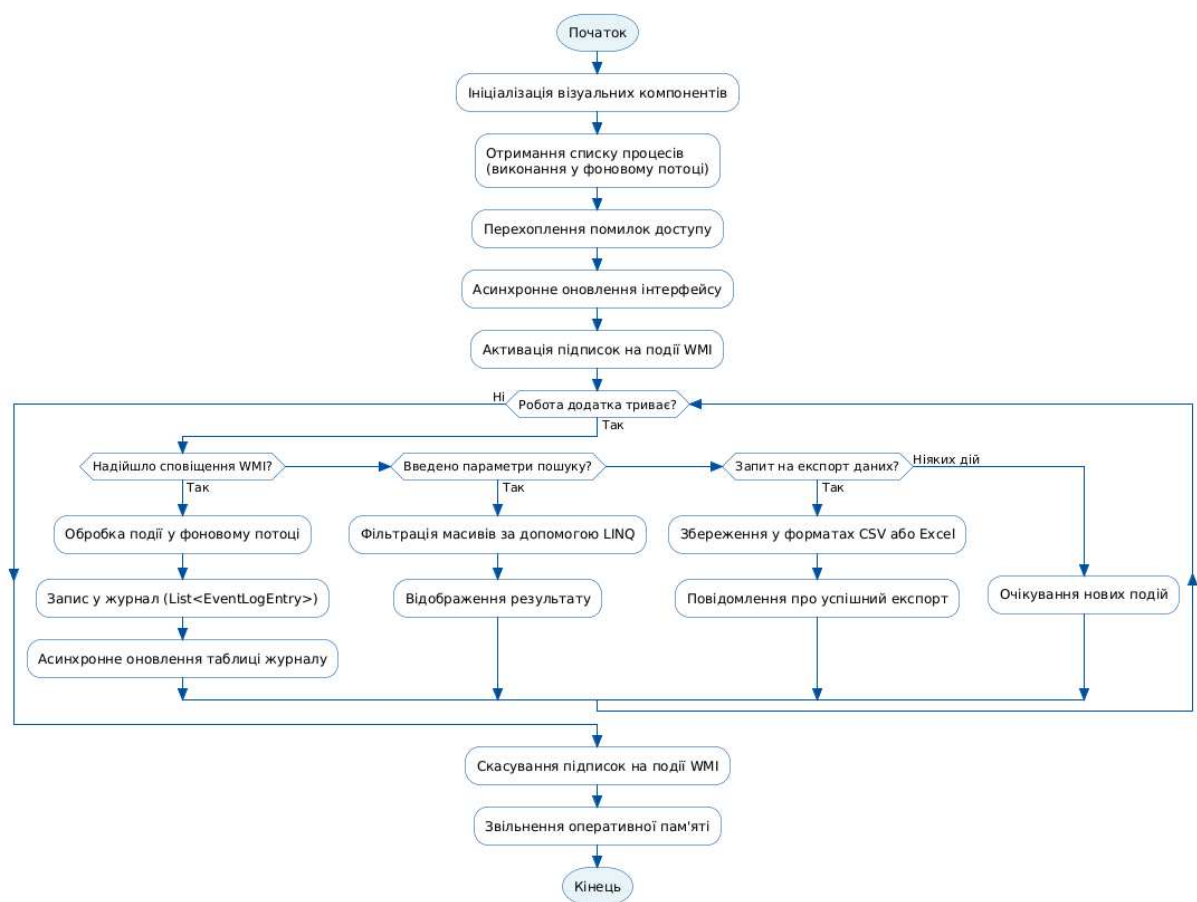


Рисунок 3.2 – Блок-схема алгоритму

Після успішного завантаження початкових даних додаток переходить до головного циклу обробки. Алгоритм розгалужується залежно від типу події. Перша гілка відповідає за моніторинг: якщо надходить сповіщення про запуск або завершення програми, алгоритм реєструє це у локальному списку пам'яті та оновлює таблицю журналу. Друга гілка реагує на дії користувача.

Наприклад, якщо користувач вводить параметри пошуку, застосовуються механізми LINQ для фільтрації відображеної інформації.

Якщо ж ініційовано операцію збереження, алгоритм перенаправляє виконання до модуля експорту, де формується файл у форматі CSV або таблиця Excel. Цей цикл триває доти, доки користувач не вирішить закрити головне вікно. Після цього алгоритм переходить до стадії завершення: скасовуються підписки на системні події, вивільняється оперативна пам'ять і процес додатка зупиняється.

3.5 Огляд роботи програми

Після запуску розробленого програмного засобу користувач потрапляє до головного робочого вікна (рисунок 3.3).

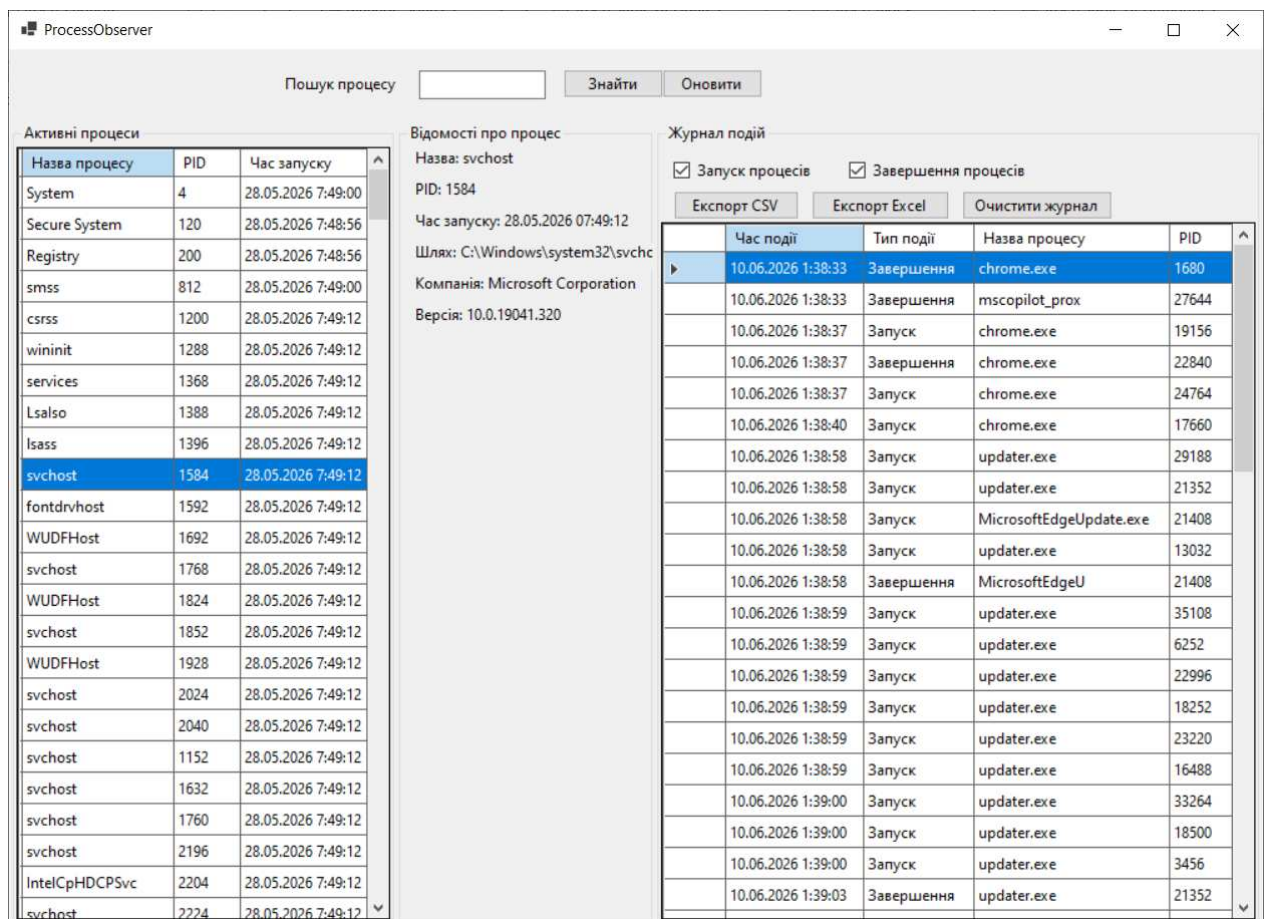
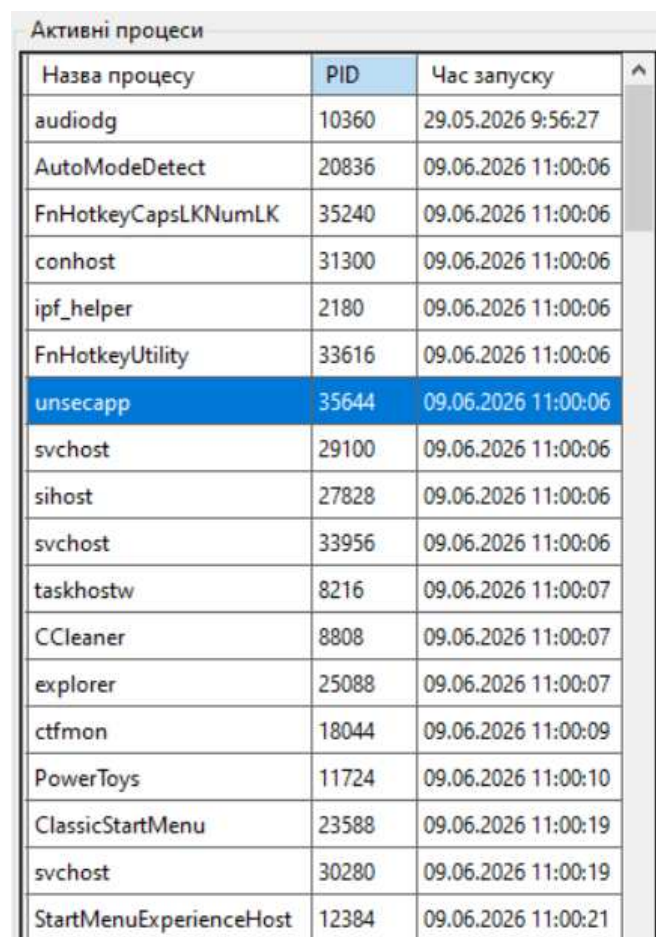


Рисунок 3.3 – Головне вікно інструментальних засобів дослідження та контролю активних процесів операційної системи Windows

Після ініціалізації розробленого програмного засобу користувач отримує доступ до головного вікна, яке слугує єдиним центром моніторингу та управління. Графічний інтерфейс для забезпечення гнучкого та адаптивного компонування візуальних елементів використовує контейнери `TableLayoutPanel` та `SplitContainer`. Це рішення дозволяє ефективно використовувати робочий простір монітора, масштабуючи таблиці та панелі незалежно від розміру екрана.

Головним елементом інтерфейсу є деталізована таблиця, що відображає масив активних програм. Під час первинного запуску система звертається до ядра операційної системи для збору базових атрибутів: ідентифікатора PID, часу старту та шляху до виконуваного файлу на диску. Для видобування розширених метаданих (інформації про компанію-розробника та версію продукту) застосовується системний компонент `FileVersionInfo`.



Назва процесу	PID	Час запуску
audiodg	10360	29.05.2026 9:56:27
AutoModeDetect	20836	09.06.2026 11:00:06
FnHotkeyCapsLKNumLK	35240	09.06.2026 11:00:06
conhost	31300	09.06.2026 11:00:06
ipf_helper	2180	09.06.2026 11:00:06
FnHotkeyUtility	33616	09.06.2026 11:00:06
unsecapp	35644	09.06.2026 11:00:06
svchost	29100	09.06.2026 11:00:06
sihost	27828	09.06.2026 11:00:06
svchost	33956	09.06.2026 11:00:06
taskhostw	8216	09.06.2026 11:00:07
CCleaner	8808	09.06.2026 11:00:07
explorer	25088	09.06.2026 11:00:07
ctfmon	18044	09.06.2026 11:00:09
PowerToys	11724	09.06.2026 11:00:10
ClassicStartMenu	23588	09.06.2026 11:00:19
svchost	30280	09.06.2026 11:00:19
StartMenuExperienceHost	12384	09.06.2026 11:00:21

Рисунок 3.4 – Перелік активних процесів

Для оперативної роботи з великими обсягами даних у верхній частині вікна передбачено панель інструментів із полем пошуку. Взаємодія з цим елементом ініціює алгоритми фільтрації на базі обробки масивів за допомогою технології LINQ. Це дає змогу адміністратору знаходити потрібні об'єкти як за точним числовим значенням ідентифікатора, так і за частковим збігом назви виконуваного файлу.

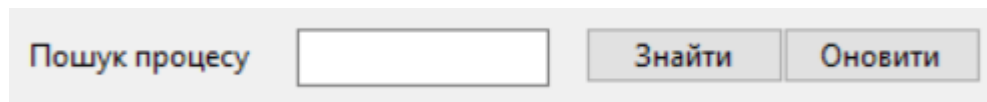


Рисунок 3.5 – Панель пошуку

Окрему секцію робочої області відведено під динамічний журнал подій. Тут у режимі реального часу накопичується інформація про зміну стану операційного середовища.

Журнал подій				
☒ Запуск процесів ☒ Завершення процесів				
Експорт CSV Експорт Excel Очистити журнал				
	Час події	Тип події	Назва процесу	PID
▶	10.06.2026 1:38:33	Завершення	chrome.exe	1680
	10.06.2026 1:38:33	Завершення	mscopilot_prox	27644
	10.06.2026 1:38:37	Запуск	chrome.exe	19156
	10.06.2026 1:38:37	Завершення	chrome.exe	22840
	10.06.2026 1:38:37	Запуск	chrome.exe	24764
	10.06.2026 1:38:40	Запуск	chrome.exe	17660
	10.06.2026 1:38:58	Запуск	updater.exe	29188
	10.06.2026 1:38:58	Запуск	updater.exe	21352
	10.06.2026 1:38:58	Запуск	MicrosoftEdgeUpdate.exe	21408
	10.06.2026 1:38:58	Запуск	updater.exe	13032
	10.06.2026 1:38:58	Завершення	MicrosoftEdgeU	21408
	10.06.2026 1:38:59	Запуск	updater.exe	35108
	10.06.2026 1:38:59	Запуск	updater.exe	6252
	10.06.2026 1:38:59	Запуск	updater.exe	22996

Рисунок 3.6 – Журнал подій

Цей механізм працює завдяки оформленій підписці на сповіщення інфраструктури WMI (зокрема, на класи подій Win32_ProcessStartTrace та Win32_ProcessStopTrace). Щойно генерується подія про запуск або зупинку компонента, додаток створює новий об'єкт у локальній колекції пам'яті та відображає його у візуальній таблиці. Це оновлення також виконується асинхронно, запобігаючи візуальним затримкам під час інтенсивних перетворень у системі.

Завершальним етапом аналізу є можливість експорту накопиченої телеметрії. За допомогою вбудованих функцій допоміжного шару архітектури користувач може згенерувати звіт у форматі CSV із застосуванням кодування UTF-8, що гарантує коректне збереження локалізованих символів.

	A	B	C	D
1	Дата	Тип	Процес	PID
2	10.06.2026 1:38	Завершення	chrome.exe	1680
3	10.06.2026 1:38	Завершення	mscopilot_prox	27644
4	10.06.2026 1:38	Запуск	chrome.exe	19156
5	10.06.2026 1:38	Завершення	chrome.exe	22840
6	10.06.2026 1:38	Запуск	chrome.exe	24764
7	10.06.2026 1:38	Запуск	chrome.exe	17660
8	10.06.2026 1:38	Запуск	updater.exe	29188
9	10.06.2026 1:38	Запуск	updater.exe	21352
10	10.06.2026 1:38	Запуск	MicrosoftEdgeUpda	21408
11	10.06.2026 1:38	Запуск	updater.exe	13032
12	10.06.2026 1:38	Завершення	MicrosoftEdgeU	21408
13	10.06.2026 1:38	Запуск	updater.exe	35108
14	10.06.2026 1:38	Запуск	updater.exe	6252
15	10.06.2026 1:38	Запуск	updater.exe	22996
16	10.06.2026 1:38	Запуск	updater.exe	18252
17	10.06.2026 1:38	Запуск	updater.exe	23220
18	10.06.2026 1:38	Запуск	updater.exe	16488
19	10.06.2026 1:39	Запуск	updater.exe	33264
20	10.06.2026 1:39	Запуск	updater.exe	18500
21	10.06.2026 1:39	Запуск	updater.exe	3456
22	10.06.2026 1:39	Завершення	updater.exe	21352
23	10.06.2026 1:39	Завершення	updater.exe	35108

Рисунок 3.7 – Результати експорту у формат CSV

Альтернативним варіантом є формування електронної таблиці Excel за допомогою інтегрованої бібліотеки ClosedXML. У цьому сценарії система автоматично виконує налаштування ширини стовпців та забезпечує правильне форматування дат, формуючи готовий до подальшого використання технічний документ.

ВИСНОВКИ

В ході виконання дипломної роботи було успішно вирішено задачу зі створення інструментального засобу для дослідження активних процесів операційної системи Windows. Поставлена мета повністю досягнута.

Основні результати роботи полягають у наступному:

а) Проведено аналіз існуючих програм-аналогів та теоретичної бази. Це дозволило визначити головні вимоги до розробки: створити зручний діагностичний інструмент, який не перевантажуватиме користувача зайвою низькорівневою інформацією.

б) Спроектовано архітектуру системи на базі платформи .NET (мова C#). Програмний код логічно розділено на моделі даних, системні служби та візуальний інтерфейс, що відповідає принципам об'єктно-орієнтованого програмування.

в) Реалізовано механізм збору системних даних. Розроблений додаток вміє отримувати як базові характеристики активних програм (ідентифікатор PID, час запуску, шлях до файлу), так і розширені дані, такі як версія продукту та назва компанії-розробника.

г) Налаштовано безперервний моніторинг системи за допомогою технології WMI. Додаток автоматично фіксує запуск та зупинку процесів. Усі ці дії виконуються у фоновому потоці, а надійна обробка помилок доступу гарантує стабільну роботу програми без збоїв.

д) Розроблено зручний адаптивний графічний інтерфейс з українською локалізацією.

е) Впроваджено функціонал для обробки зібраної інформації. Додано зручний пошук потрібних компонентів, а також інструменти для збереження історії подій у текстові файли CSV або в готові електронні таблиці Excel з автоматичним налаштуванням ширини колонок.

Створений програмний засіб повністю відповідає початковим вимогам. Він має практичну цінність і може ефективно використовуватися як для

повсякденного моніторингу стану комп'ютера, так і як наочний посібник під час вивчення архітектури ОС Windows. Для системних адміністраторів та досвідчених користувачів додаток стане надійним помічником у повсякденній діагностиці комп'ютера, відстеженні підозрілих програмних активностей та аналізі споживання ресурсів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Windows Management Instrumentation [Електронний ресурс] / Microsoft Learn. – 2023. – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page> (дата звернення: 06.05.2026).
2. Yosifovich P. and others. Windows Internals, Part 2. 7th ed. Redmond : Microsoft Press, 2021. 770 p.
3. Win32_ProcessStartTrace class [Електронний ресурс] / Microsoft Learn. – 2018. – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/previous-versions/windows/desktop/krnlprow/win32-processstarttrace> (дата звернення: 06.05.2026).
4. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. 10th ed. Hoboken : Wiley, 2018. 976 p.
5. Габрусєв, Валерій Юрійович. Основи операційних систем: ядро, процес, потік [Текст] / В. Ю. Габрусєв [и др.]. - Т. : Навчальна книга - Богдан, 2007. - 96 с.
6. Диспетчер завдань: інструкція для найпростішого запуску та аналізу [Електронний ресурс] / Fenix Industry. – 2024. – Режим доступу до ресурсу: <https://fnx.com.ua/ua/articles/publications/50> (дата звернення: 06.06.2026).
7. Process Explorer [Електронний ресурс] / Microsoft Learn. – 2026. – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer> (дата звернення: 06.06.2026).
8. Process Hacker: Overview [Електронний ресурс] / SourceForge. – Режим доступу до ресурсу: <https://processhacker.sourceforge.io/> (дата звернення: 06.06.2026).
9. Лавріщева К. М. Програмна інженерія : підручник. Київ : Академперіодика, 2008. 319 с.
10. Price M. J. C# 12 and .NET 8 – Modern Cross-Platform Development: Fundamentals, Web, and Cloud. 8th ed. Birmingham : Packt Publishing, 2023. 822 p.

11. Troelsen A., Japikse P. Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming. 11th ed. Berkeley : Apress, 2022. 1308 p.
12. Process Class (System.Diagnostics) [Электронный ресурс] / Microsoft Learn. – 2024. – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process> (дата звернення: 06.06.2026).
13. Albahari J. C# 12 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2023. 1088 p.
14. Holmes A. Understanding WMI Scripting: Exploiting Microsoft's Windows Management Instrumentation in Mission-Critical Computing Infrastructures. Burlington : Digital Press, 2003. 500 p.
15. WMI Architecture [Электронный ресурс] / Microsoft Learn. – 2021. – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-architecture> (дата звернення: 06.06.2026).
16. System.Management Namespace [Электронный ресурс] / Microsoft Learn. – 2024. – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/api/system.management> (дата звернення: 06.06.2026).
17. Windows Management Instrumentation cheatsheet [Электронный ресурс] / Режим доступа до ресурсу: <https://www.poco602.com/post/0080-wmi-cheatsheet/> (дата звернення: 06.06.2026).
18. Win32_ProcessStartTrace class [Электронный ресурс] / Microsoft Learn. – 2018. – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/previous-versions/windows/desktop/knlprov/win32-processstarttrace> (дата звернення: 06.06.2026).
19. Platform Invoke (P/Invoke) [Электронный ресурс] / Microsoft Learn. – 2023. – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke> (дата звернення: 06.06.2026).
20. Windows API [Электронный ресурс] – Режим доступа до ресурсу: https://www.wikiwand.com/en/Windows_API (дата звернення: 06.06.2026).

Додаток А – Фрагменти програмного коду

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Windows.Forms;
using ProcessObserver.Services;
using ProcessObserver.Models;

namespace ProcessObserver.Forms
{
    /*
     * Головна форма програми ProcessObserver.
     *
     * Це основне вікно програми, яке відображає інформацію
     * про активні процеси, журнал подій та забезпечує
     * взаємодію користувача з програмою.
     */
    public partial class Form1 : Form
    {
        /*
         * Сервіс для отримання списку процесів.
         * Використовується для завантаження інформації про
         * активні процеси системи.
         */
        private ProcessProvider _processProvider;

        /*
         * Сервіс для моніторингу запуску та завершення
 процесів.
         * Використовує WMI для відстеження подій системи в
 реальному часі.
         */
        private ProcessMonitorService _processMonitorService;

        /*
         * Сервіс для експорту журналу подій у різні формати.
         * Використовується для збереження даних у CSV та Excel.
         */
        private ExportService _exportService;

        /*
         * Список завантажених процесів.
         * Зберігається в пам'яті для швидкого доступу
         * та пошуку процесів.
         */
        private List<ProcessInfo> _processes;

        /*
```

```

        * Журнал подій запуску та завершення процесів.
        * Зберігає всі події для подальшого аналізу та
експорту.
    */
    private List<Models.EventLogEntry> _eventLog;

    /*
    * Конструктор форми.
    * Ініціалізує компоненти форми та виконує
    * початкове налаштування.
    */
    public Form1()
    {
        // Ініціалізуємо компоненти форми (створюється
дизайнером)
        InitializeComponent();

        // Створюємо екземпляр сервісу для роботи з
процесами
        _processProvider = new ProcessProvider();

        // Створюємо екземпляр сервісу для моніторингу
процесів
        _processMonitorService = new
ProcessMonitorService();

        // Створюємо екземпляр сервісу для експорту
        _exportService = new ExportService();

        // Ініціалізуємо список процесів
        _processes = new List<ProcessInfo>();

        // Ініціалізуємо журнал подій
        _eventLog = new List<Models.EventLogEntry>();

        // Підписуємося на події завантаження форми
Load += Form1_Load;

        // Підписуємося на подію закриття форми для
коректного завершення моніторингу
FormClosing += Form1_FormClosing;

        // Підписуємося на подію вибору рядка в таблиці
процесів
dgvProcesses.SelectionChanged +=
DgvProcesses_SelectionChanged;

        // Підписуємося на подію натискання кнопки оновлення
btnRefresh.Click += BtnRefresh_Click;

        // Підписуємося на подію натискання кнопки пошуку
btnSearch.Click += BtnSearch_Click;

```

```

        // Підписуємося на подію зміни тексту в полі пошуку
для очищення результатів
        txtSearch.TextChanged += TxtSearch_TextChanged;

        // Підписуємося на подію натискання кнопки очищення
журналу
        btnClearLog.Click += BtnClearLog_Click;

        // Підписуємося на подію зміни прапорців фільтрації
        chkStartEvents.CheckedChanged +=
        ChkFilter_CheckedChanged;
        chkStopEvents.CheckedChanged +=
        ChkFilter_CheckedChanged;

        // Підписуємося на подію експорту
        btnExportCsv.Click += BtnExportCsv_Click;
        btnExportExcel.Click += BtnExportExcel_Click;

        // Підписуємося на подію моніторингу процесів
        _processMonitorService.ProcessStarted +=
        OnProcessStarted;
        _processMonitorService.ProcessStopped +=
        OnProcessStopped;
    }

    /*
    * Обробник події завантаження форми.
    *
    * Викликається при відкритті форми.
    * Завантажує початковий список процесів та запускає
моніторинг.
    */
    private void Form1_Load(object sender, EventArgs e)
    {
        try
        {
            // Завантажуємо список процесів при старті
програми
            LoadProcesses();

            // Запускаємо моніторинг запуску та завершення
процесів
            _processMonitorService.StartMonitoring();
        }
        catch (Exception ex)
        {
            /*
            * Обробляємо помилки при завантаженні процесів.
            * Виводимо повідомлення користувачу про
проблему.
            */

```

```

        MessageBox.Show(
            $"Помилка при завантаженні процесів:
{ex.Message}",
            "Помилка",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    }
}

/*
 * Обробник події закриття форми.
 *
 * Викликається при закритті програми.
 * Коректно зупиняє моніторинг та звільняє ресурси.
 */
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    try
    {
        // Зупиняємо моніторинг процесів
        _processMonitorService.StopMonitoring();
    }
    catch (Exception ex)
    {
        /*
         * Обробляємо помилки при зупинці моніторингу.
         * Навіть при помилці продовжуємо закриття
програми.
         */
        Console.WriteLine($"Помилка при зупинці
моніторингу: {ex.Message}");
    }
}

/*
 * Завантажує список процесів та відображає його в
таблиці.
 *
 * Використовує ProcessProvider для отримання інформації
 * про всі активні процеси системи та відображає їх
 * в DataGridView dgvProcesses.
 */
private void LoadProcesses()
{
    try
    {
        // Отримуємо список процесів через сервіс
        _processes = _processProvider.GetProcesses();

        // Очищаємо таблицю перед завантаженням нових
даних

```

```

dgvProcesses.DataSource = null;
dgvProcesses.Rows.Clear();

// Заповнюємо таблицю даними про процеси
foreach (var process in _processes)
{
    dgvProcesses.Rows.Add(
        process.Name,
        process.ProcessId,
        process.StartTime);
}

/*
 * Налаштовуємо автоматичну ширину стовпців
 * для кращого відображення даних.
 */

dgvProcesses.AutoSizeColumns(DataGridViewAutoSizeColumnsMode.D
isplayedCells);
}
catch (Exception ex)
{
    /*
     * Обробляємо помилки при завантаженні процесів.
     */
    MessageBox.Show(
        $"Помилка при завантаженні процесів:
{ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}

/*
 * Обробник події вибору рядка в таблиці процесів.
 *
 * Викликається при зміні вибраного рядка в
dgvProcesses.
 * Відображає базову інформацію про вибраний процес
 * в правій панелі.
 */
private void DgvProcesses_SelectionChanged(object
sender, EventArgs e)
{
    try
    {
        // Перевіряємо, чи є вибраний рядок
        if (dgvProcesses.SelectedRows.Count == 0)
        {
            /*

```

```

        * Якщо рядок не вибрано, очищуємо панель
деталей.

        */
        ClearProcessDetails();
        return;
    }

    // Отримуємо індекс вибраного рядка
    int selectedIndex =
dgvProcesses.SelectedRows[0].Index;

    // Перевіряємо коректність індексу
    if (selectedIndex < 0 || selectedIndex >=
_processes.Count)
    {
        return;
    }

    // Отримуємо інформацію про вибраний процес
    var selectedProcess = _processes[selectedIndex];

    // Відображаємо базову інформацію про процес
    DisplayProcessDetails(selectedProcess);
}
catch (Exception ex)
{
    /*
    * Обробляємо помилки при відображенні деталей
процесу.

    */
    MessageBox.Show(
        {ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}

}

/*
* Відображає інформацію про процес в правій панелі.
*
* Отримує детальну інформацію про процес через
* System.Diagnostics.Process та FileVersionInfo.
* Відображає шлях до файлу, компанію-розробника
* та версію продукту.
*
* @param process Об'єкт ProcessInfo з інформацією про
процес
*/
private void DisplayProcessDetails(ProcessInfo process)
{

```

```

try
{
    // Відображаємо назву процесу
    lblDetailName.Text = $"Назва: {process.Name}";

    // Відображаємо ідентифікатор процесу
    lblDetailPid.Text = $"PID: {process.ProcessId}";

    // Відображаємо час запуску процесу
    lblDetailStartTime.Text = $"Час запуску:
{process.StartTime:dd.MM.yyyy HH:mm:ss}";

    /*
    * Отримуємо детальну інформацію про процес.
    * Використовуємо ProcessProvider для отримання
    * додаткових даних про процес.
    */
    var detailedProcess =
        _processProvider.GetProcessDetails(process.ProcessId);

    if (detailedProcess != null)
    {
        /*
        * Якщо вдалося отримати детальну
інформацію,
        * відображаємо шлях до виконуваного файлу.
        */
        lblDetailPath.Text = $"Шлях:
{detailedProcess.ExecutablePath ?? "Недоступно"}";

        /*
        * Отримуємо інформацію про версію файлу
через FileVersionInfo.
        * Це дозволяє отримати метадані
виконуваного файлу,
        * такі як компанія-розробник та версія
продукту.
        */
        try
        {
            if
(!string.IsNullOrEmpty(detailedProcess.ExecutablePath) &&
                detailedProcess.ExecutablePath !=
"Недоступно")
            {
                // Отримуємо версію файлу
                FileVersionInfo versionInfo =
                FileVersionInfo.GetVersionInfo(detailedProcess.ExecutablePath);

                // Відображаємо компанію-розробника
                lblDetailCompany.Text = $"Компанія:
{versionInfo.CompanyName ?? "Недоступно"}";

```



```

        // Відображаємо версію продукту
        lblDetailVersion.Text = $"Версія:
{versionInfo.ProductVersion ?? "Недоступно"}";
    }
    else
    {
        /*
        * Якщо шлях до файлу недоступний,
        * відображаємо повідомлення про
недоступність.

        */
        lblDetailCompany.Text = "Компанія:
Недоступно";
        lblDetailVersion.Text = "Версія:
Недоступно";
    }
}
catch (Exception ex)
{
    /*
    * Обробляємо помилки при отриманні
версії файлу.

    * Це може статися для системних

    * якщо файл недоступний для читання.
    */
    lblDetailCompany.Text = "Компанія:
Недоступно";
    lblDetailVersion.Text = "Версія:
Недоступно";
    Console.WriteLine($"Помилка при
отриманні версії файлу: {ex.Message}");
}
}
else
{
    /*
    * Якщо не вдалося отримати детальну
інформацію

    * (процес міг бути завершений),
    * відображаємо повідомлення про
недоступність.

    */
    lblDetailPath.Text = "Шлях: Недоступно";
    lblDetailCompany.Text = "Компанія:
Недоступно";
    lblDetailVersion.Text = "Версія:
Недоступно";
}
}
catch (Exception ex)

```

```

        {
            /*
             * Обробляємо помилки при відображенні деталей.
             */
            MessageBox.Show(
                {ex.Message}",
                "Помилка",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }

    /*
     * Очищає панель деталей процесу.
     * Використовується при скасуванні вибору рядка
     * або при виникненні помилок.
     */
    private void ClearProcessDetails()
    {
        lblDetailName.Text = "Назва: -";
        lblDetailPid.Text = "PID: -";
        lblDetailStartTime.Text = "Час запуску: -";
        lblDetailPath.Text = "Шлях: -";
        lblDetailCompany.Text = "Компанія: -";
        lblDetailVersion.Text = "Версія: -";
    }

    /*
     * Обробник події натискання кнопки "Оновити".
     * Перезавантажує список процесів з системи.
     * Це дозволяє побачити нові процеси, які запустилися
     * після останнього оновлення.
     */
    private void BtnRefresh_Click(object sender, EventArgs
e)
    {
        try
        {
            // Перезавантажуємо список процесів
            LoadProcesses();

            // Очищаємо панель деталей, оскільки вибір міг
змінитися
            ClearProcessDetails();
        }
        catch (Exception ex)
        {
            /*
             * Обробляємо помилки при оновленні списку.

```

```

        */
        MessageBox.Show(
            $"Помилка при оновленні: {ex.Message}",
            "Помилка",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    }
}

/*
 * Обробник події натискання кнопки "Знайти".
 *
 * Реалізує пошук процесів за назвою або PID.
 * Підсвічує знайдені записи в таблиці процесів.
 */
private void BtnSearch_Click(object sender, EventArgs e)
{
    try
    {
        // Отримуємо пошуковий запит з текстового поля
        string searchText = txtSearch.Text.Trim();

        // Перевіряємо, чи не порожній пошуковий запит
        if (string.IsNullOrEmpty(searchText))
        {
            MessageBox.Show(
                "Введіть назву процесу або PID для
пошуку.",
                "Пошук",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
            return;
        }

        // Скидаємо виділення перед пошуком
        dgvProcesses.ClearSelection();

        // Перевіряємо, чи є числове значення (пошук за
PID)
        bool isNumeric = int.TryParse(searchText, out
int searchPid);

        bool found = false;
        bool firstMatch = true;

        // Проходимо по всіх рядках таблиці процесів
        foreach (DataGridViewRow row in
dgvProcesses.Rows)
        {
            if (row.IsNewRow)
            {
                continue;

```

```

    }

    // Отримуємо назву процесу з першого стовпця
    string processName =
row.Cells[0].Value?.ToString() ?? "";

    // Отримуємо PID з другого стовпця
    int processId = 0;
    if (row.Cells[1].Value != null)
    {

int.TryParse(row.Cells[1].Value.ToString(), out processId);
    }

    // Перевіряємо відповідність пошуковому
запиту
    bool match = false;

    if (isNumeric)
    {
        // Пошук за PID
        if (processId == searchPid)
        {
            match = true;
        }
    }
    else
    {
        // Пошук за назвою (без урахування
регістру)
        if (processName.IndexOf(searchText,
StringComparison.OrdinalIgnoreCase) >= 0)
        {
            match = true;
        }
    }

    // Якщо знайдено відповідність, виділяємо
рядок
    if (match)
    {
        row.Selected = true;
        row.Visible = true;
        found = true;

        // Прокручуємо таблицю до першого
знайденого запису
        if (firstMatch)
        {

dgvProcesses.FirstDisplayedScrollingRowIndex = row.Index;
            firstMatch = false;

```

```

        }
    }
    else
    {
        // Знімаємо виділення з рядків, що не
відповідають пошуку
        row.Selected = false;
    }
}

// Виводимо повідомлення про результати пошуку
if (found)
{
    // Отримуємо кількість знайдених записів
    int foundCount =
dgvProcesses.SelectedRows.Count;

    // Виводимо повідомлення про успішний пошук
    MessageBox.Show(
        $"Знайдено {foundCount} процесів.",
        "Пошук",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
else
{
    // Виводимо повідомлення про відсутність
результатів
    MessageBox.Show(
        "Процеси не знайдено.",
        "Пошук",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
}
catch (Exception ex)
{
    /*
    * Обробляємо помилки при пошуку процесів.
    */
    MessageBox.Show(
        $"Помилка при пошуку: {ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}
}

/*
* Обробник події зміни тексту в полі пошуку.
*
* Викликається при зміні тексту в txtSearch.
*/

```

```

        * Якщо текст порожній, скидає виділення в таблиці
процесів.
        */
        private void TxtSearch_TextChanged(object sender,
EventArgs e)
        {
            try
            {
                // Перевіряємо, чи порожнє текстове поле
                if (string.IsNullOrEmpty(txtSearch.Text.Trim()))
                {
                    // Скидаємо виділення в таблиці процесів
                    dgvProcesses.ClearSelection();
                }
            }
            catch (Exception ex)
            {
                /*
                * Обробляємо помилки при зміні тексту пошуку.
                */
                Console.WriteLine($"Помилка при зміні тексту
пошуку: {ex.Message}");
            }
        }

        /*
        * Обробник події натискання кнопки "Експорт CSV".
        *
        * Відкриває діалог збереження файлу та експортує журнал
подій у формат CSV.
        */
        private void BtnExportCsv_Click(object sender, EventArgs
e)
        {
            try
            {
                // Перевіряємо, чи є записи в журналі
                if (_eventLog.Count == 0)
                {
                    MessageBox.Show(
                        "Журнал подій порожній. Немає даних для
експорту.",
                        "Експорт CSV",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Information);
                    return;
                }

                // Створюємо діалог збереження файлу
                using (var saveFileDialog = new
SaveFileDialog())
                {

```

```

        // Налаштовуємо діалог
        saveFileDialog.Filter = "CSV files
(*.csv)|*.csv|All files (*.*)|*.*";
        saveFileDialog.DefaultExt = "csv";
        saveFileDialog.FileName =
$"EventLog_{DateTime.Now:yyyyMMdd_HH:mm:ss}.csv";
        saveFileDialog.Title = "Експорт журналу
подій у CSV";

        // Показуємо діалог та перевіряємо результат
        if (saveFileDialog.ShowDialog() ==
DialogResult.OK)
        {
            // Отримуємо шлях до файлу
            string filePath =
saveFileDialog.FileName;

            // Експортуємо журнал через сервіс
            bool success =
_exportService.ExportToCsv(_eventLog, filePath);

            // Перевіряємо результат експорту
            if (success)
            {
                MessageBox.Show(
                    $"Журнал подій успішно
експортовано у файл:\n{filePath}",
                    "Експорт CSV",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
            }
            else
            {
                MessageBox.Show(
                    "Помилка при експорті журналу
подій.",
                    "Експорт CSV",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
            }
        }
    }
}
catch (Exception ex)
{
    /*
    * Обробляємо помилки при експорті в CSV.
    */
    MessageBox.Show(
        $"Помилка при експорті: {ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,

```

```

        MessageBoxIcon.Error);
    }
}

/*
 * Обробник події натискання кнопки "Експорт Excel".
 *
 * Відкриває діалог збереження файлу та експортує журнал
 подій у формат Excel (XLSX).
 */
private void BtnExportExcel_Click(object sender,
EventArgs e)
{
    try
    {
        // Перевіряємо, чи є записи в журналі
        if (_eventLog.Count == 0)
        {
            MessageBox.Show(
                "Журнал подій порожній. Немає даних для
експорту.",
                "Експорт Excel",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
            return;
        }

        // Створюємо діалог збереження файлу
        using (var saveFileDialog = new
SaveFileDialog())
        {
            // Налаштовуємо діалог
            saveFileDialog.Filter = "Excel files
(*.xlsx)|*.xlsx|All files (*.*)|*.*";
            saveFileDialog.DefaultExt = "xlsx";
            saveFileDialog.FileName =
$"EventLog_{DateTime.Now:yyyyMMdd_HH:mm:ss}.xlsx";
            saveFileDialog.Title = "Експорт журналу
подій у Excel";

            // Показуємо діалог та перевіряємо результат
            if (saveFileDialog.ShowDialog() ==
DialogResult.OK)
            {
                // Отримуємо шлях до файлу
                string filePath =
saveFileDialog.FileName;

                // Екпортуємо журнал через сервіс
                bool success =
_exportService.ExportToExcel(_eventLog, filePath);

```



```

        // Перевіряємо результат експорту
        if (success)
        {
            MessageBox.Show(
                $"Журнал подій успішно
експортовано у файл:\n{filePath}",
                "Експорт Excel",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
        }
        else
        {
            MessageBox.Show(
                "Помилка при експорті журналу
подій.",
                "Експорт Excel",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }
}
catch (Exception ex)
{
    /*
    * Обробляємо помилки при експорті в Excel.
    */
    MessageBox.Show(
        $"Помилка при експорті: {ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}

/*
* Обробник події запуску нового процесу.
*
* Викликається автоматично при запуску нового процесу в
системі.
* Додає запис у журнал та оновлює список процесів.
*
* @param processName Назва запущеного процесу
* @param processId Ідентифікатор запущеного процесу
*/
private void OnProcessStarted(string processName, int
processId)
{
    try
    {
        /*

```

```

        * Використовуємо Invoke для безпечного
оновлення інтерфейсу
        * з фонового потоку WMI. Це необхідно, оскільки
події WMI
        * приходять з іншого потоку.
        */
        if (InvokeRequired)
        {
            Invoke(new Action<string,
int>(OnProcessStarted), processName, processId);
            return;
        }

        /*
        * Додаємо запис у журнал подій про запуск
процесу.
        */
        AddEventToLog("Запуск", processName, processId);

        /*
        * Оновлюємо список процесів, щоб відобразити
новий процес.
        * Використовуємо BeginInvoke для асинхронного
оновлення,
        * щоб не блокувати потік WMI.
        */
        BeginInvoke(new Action(() =>
        {
            try
            {
                LoadProcesses();
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Помилка при
оновленні списку процесів: {ex.Message}");
            }
        }));
    }
    catch (Exception ex)
    {
        /*
        * Обробляємо помилки при обробці події запуску
процесу.
        */
        Console.WriteLine($"Помилка при обробці події
запуску: {ex.Message}");
    }
}

/*
* Обробник події завершення процесу.

```

```

        *
        * Викликається автоматично при завершенні процесу в
системі.
        * Додає запис у журнал та оновлює список процесів.
        *
        * @param processName Назва завершеного процесу
        * @param processId Ідентифікатор завершеного процесу
        */
private void OnProcessStopped(string processName, int
processId)
{
    try
    {
        /*
        * Використовуємо Invoke для безпечного
оновлення інтерфейсу
        * з фонового потоку WMI.
        */
        if (InvokeRequired)
        {
            Invoke(new Action<string,
int>(OnProcessStopped), processName, processId);
            return;
        }

        /*
        * Додаємо запис у журнал подій про завершення
процесу.
        */
        AddEventToLog("Завершення", processName,
processId);

        /*
        * Оновлюємо список процесів, щоб видалити
завершений процес.
        */
        BeginInvoke(new Action(() =>
        {
            try
            {
                LoadProcesses();
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Помилка при
оновленні списку процесів: {ex.Message}");
            }
        }));
    }
    catch (Exception ex)
    {
        /*

```

```

        * Обробляємо помилки при обробці події
завершення процесу.
        */
        Console.WriteLine($"Помилка при обробці події
завершення: {ex.Message}");
    }

    }

    /*
    * Додає запис у журнал подій.
    *
    * Створює новий запис EventLogEntry та додає його до
колекції.
    * Оновлює відображення журналу в DataGridView.
    *
    * @param eventType Тип події (Запуск або Завершення)
    * @param processName Назва процесу
    * @param processId Ідентифікатор процесу
    */
    private void AddEventToLog(string eventType, string
processName, int processId)
    {
        try
        {
            // Створюємо новий запис журналу
            var eventEntry = new Models.EventLogEntry
            {
                EventTime = DateTime.Now,
                EventType = eventType,
                ProcessName = processName,
                ProcessId = processId
            };

            // Додаємо запис до колекції журналу
            _eventLog.Add(eventEntry);

            // Оновлюємо відображення журналу
            RefreshEventLogDisplay();
        }
        catch (Exception ex)
        {
            /*
            * Обробляємо помилки при додаванні запису в
журнал.
            */
            Console.WriteLine($"Помилка при додаванні запису
в журнал: {ex.Message}");
        }
    }

    /*
    * Оновлює відображення журналу в DataGridView.

```

```

        *
        * Перезавантажує дані з колекції _eventLog у dgvEvents.
        * Використовується після додавання нових записів або
        фільтрації.
        */
private void RefreshEventLogDisplay()
{
    try
    {
        // Очищаємо таблицю журналу
        dgvEvents.DataSource = null;
        dgvEvents.Rows.Clear();

        // Отримуємо відфільтрований список подій
        var filteredEvents = GetFilteredEvents();

        // Заповнюємо таблицю даними про події
        foreach (var eventEntry in filteredEvents)
        {
            dgvEvents.Rows.Add(
                eventEntry.EventTime,
                eventEntry.EventType,
                eventEntry.ProcessName,
                eventEntry.ProcessId);
        }

        /*
        * Налаштовуємо автоматичну ширину стовпців
        * для кращого відображення даних.
        */

        dgvEvents.AutoSizeColumns(DataGridViewAutoSizeColumnsMode.DisplayedCells);
    }
    catch (Exception ex)
    {
        /*
        * Обробляємо помилки при оновленні відображення
        журналу.
        */
        Console.WriteLine($"Помилка при оновленні відображення журналу: {ex.Message}");
    }
}

/*
* Отримує відфільтрований список подій.
*
* Повертає події згідно з налаштуваннями фільтрів.
* Фільтрація виконується без втрати даних – повний
журнал

```

```

* зберігається в колекції _eventLog, а повертається
лише
* відфільтроване представлення.
*
* @return Відфільтрований список подій
*/
private List<Models.EventLogEntry> GetFilteredEvents()
{
    try
    {
        // Отримуємо стан прапорців фільтрації
        bool showStartEvents = chkStartEvents.Checked;
        bool showStopEvents = chkStopEvents.Checked;

        /*
        * Якщо обидва прапорці активні, повертаємо всі
        події.
        * Це оптимізація для уникнення непотрібної
        фільтрації.
        */
        if (showStartEvents && showStopEvents)
        {
            return _eventLog.ToList();
        }

        /*
        * Фільтруємо події згідно з налаштуваннями.
        * Використовуємо LINQ для зручної фільтрації.
        */
        var filteredEvents = _eventLog.Where(entry =>
        {
            // Перевіряємо, чи потрібно показувати подію
            запуску
            if (entry.EventType == "Запуск" &&
            showStartEvents)
            {
                return true;
            }

            // Перевіряємо, чи потрібно показувати подію
            завершення
            if (entry.EventType == "Завершення" &&
            showStopEvents)
            {
                return true;
            }

            // Подія не відповідає фільтрам
            return false;
        }).ToList();

        return filteredEvents;
    }
}

```

```

    }
    catch (Exception ex)
    {
        /*
         * Обробляємо помилки при фільтрації подій.
         */
        Console.WriteLine($"Помилка при фільтрації
подій: {ex.Message}");
        return new List<Models.EventLogEntry>();
    }
}

/*
 * Обробник події зміни прапорців фільтрації.
 *
 * Викликається при зміні стану chkStartEvents або
chkStopEvents.
 * Миттєво оновлює відображення журналу з новими
фільтрами.
 */
private void ChkFilter_CheckedChanged(object sender,
EventArgs e)
{
    try
    {
        // Оновлюємо відображення журналу з новими
фільтрами
        RefreshEventLogDisplay();
    }
    catch (Exception ex)
    {
        /*
         * Обробляємо помилки при зміні фільтрів.
         */
        Console.WriteLine($"Помилка при зміні фільтрів:
{ex.Message}");
    }
}

/*
 * Обробник події натискання кнопки "Очистити журнал".
 *
 * Очищає колекцію журналу та оновлює відображення.
 */
private void BtnClearLog_Click(object sender, EventArgs
e)
{
    try
    {
        // Очищаємо колекцію журналу
        _eventLog.Clear();
    }
}

```

```
// Оновлюємо відображення журналу
RefreshEventLogDisplay();

// Виводимо повідомлення про успішне очищення
MessageBox.Show(
    "Журнал подій очищено.",
    "Інформація",
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
}
catch (Exception ex)
{
    /*
     * Обробляємо помилки при очищенні журналу.
     */
    MessageBox.Show(
        {ex.Message}",
        "Помилка",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
}
}
}
```