

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту**

До захисту допущено:

В. о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«__» _____ 2026 р.

**Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи і методи штучного
інтелекту» спеціальності 122 «Комп'ютерні науки»
на тему: «Адаптивне управління процесами в ОС Linux з використанням
методів штучного інтелекту»**

Виконала:

Студентка IV курсу, групи КІ-22
Мороз Таїсія Миколаївна _____

Керівник:

доцент кафедри ШІ, к.т.н., доцент
Коваленко Анатолій Єпіфанович _____

Консультант з нормоконтролю:

фахівець першої категорії кафедри ШІ,
к. т. н., доцент, Комариста Богдана Миколаївна _____

Рецензент:

доцент кафедри ММСА, к.т.н.,
Зінченко Артем Юрійович _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студентка _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Системи і методи штучного інтелекту»

ЗАТВЕРДЖУЮ

В. о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«15» січня 2026 р.

ЗАВДАННЯ
на дипломну роботу студентці
Мороз Таїсії Миколаївні

1. Тема роботи «Адаптивне управління процесами в ОС Linux з використанням методів штучного інтелекту», керівник роботи Коваленко Анатолій Єпіфанович, к. т. н., доцент кафедри ШІ, затверджені наказом по НН ПСА від «27» травня 2026 р. № 1944-с.

2. Термін подання студентом роботи «05 » червня 2026 року.

3. Вихідні дані до роботи: операційна система Linux; стандартні механізми керування процесами та ресурсами Linux, зокрема cgroups v2, cpu.weight, cpu.max, nice/renice, /proc та sysfs; методи навчання з підкріпленням, зокрема Q-learning; експериментальний програмний прототип адаптивного контролера; результати тестових запусків, Q-таблиця та зібрані системні метрики.

4. Зміст роботи: аналіз механізмів керування процесами та ресурсами в ОС Linux; дослідження стандартних засобів user-space керування, зокрема cgroups v2, cpu.weight, cpu.max, nice/renice, /proc та sysfs; формалізація задачі адаптивного керування процесами як марковського процесу прийняття

рішень; розроблення математичної моделі адаптивного контролера на основі Q-learning; програмна реалізація експериментального user-space контролера; проведення експериментального дослідження, аналіз Q-table та отриманих результатів; функціонально-вартісний аналіз програмного продукту.

5. Перелік ілюстративного матеріалу: презентація до захисту роботи.

6. Дата видачі завдання «02» лютого 2026 року.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Підготовка даних до роботи	20.04.2026	Виконано
2	Вивчення літератури за темою роботи	28.04.2026	Виконано
3	Підготовка першого розділу	05.05.2026	Виконано
4	Підготовка другого розділу	12.05.2026	Виконано
5	Підготовка третього розділу	19.05.2026	Виконано
6	Підготовка четвертого розділу	26.05.2026	Виконано
7	Оформлення розділів відповідно до нормоконтролю	02.06.2026	Виконано
8	Оформлення дипломної роботи	09.06.2026	Виконано

Студент

Таїсія МОРОЗ

Керівник

Анатолій КОВАЛЕНКО

РЕФЕРАТ

Дипломна робота: 119 с., 4 рис., 13 табл., 18 джерел, 2 додатки.

АДАПТИВНЕ УПРАВЛІННЯ, LINUX, ПРОЦЕСИ, CGROUPS V2, CPU.WEIGHT, CPU.MAX, NICE, RENICE, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, Q-НАВЧАННЯ, Q-ТАБЛИЦЯ, USER-SPACE CONTROLLER.

Об'єктом дослідження є процеси керування обчислювальними ресурсами в операційній системі Linux в умовах динамічного навантаження.

Предметом дослідження є методи та програмні засоби адаптивного керування процесами в Linux на рівні простору користувача з використанням навчання з підкріпленням.

Метою роботи є дослідження та розроблення експериментального підходу до адаптивного керування процесами в ОС Linux на основі контролера простору користувача, який використовує модель марковського процесу прийняття рішень і метод Q-навчання.

У роботі проаналізовано механізми керування процесами та ресурсами Linux, зокрема nice/renice, cgroups v2, cpu.weight, cpu.max, /proc та sysfs, а також обґрунтовано доцільність застосування навчання з підкріпленням для адаптивного керування в умовах змінного навантаження.

Задачу адаптивного керування формалізовано як марковський процес прийняття рішень. Запропоновано математичну модель, що включає опис станів системи, множину керуючих дій і функцію винагороди, побудовану з урахуванням використання CPU, системного навантаження та ознак throttling. Для вибору дій використано табличний алгоритм Q-навчання.

Розроблено програмний прототип адаптивного контролера простору користувача, який здійснює збір системних метрик, формування стану середовища, вибір і застосування керуючих дій через стандартні механізми Linux та збереження результатів експериментів.

Проведене експериментальне дослідження підтвердило працездатність запропонованого підходу та можливість використання навчання з підкріпленням для адаптивного керування процесами без модифікації ядра Linux.

Практичне значення роботи полягає у створенні експериментального програмного прототипу, який може бути використаний як основа для подальших досліджень у сфері адаптивного керування системними ресурсами та розроблення інтелектуальних засобів підтримки керування процесами в ОС Linux.

ABSTRACT

Bachelor thesis: 119 pages, 4 figures, 13 tables, 18 references, 2 appendices.

ADAPTIVE MANAGEMENT, LINUX, PROCESSES, CGROUPS V2, CPU.WEIGHT, CPU.MAX, NICE, RENICE, REINFORCEMENT LEARNING, Q-LEARNING, Q-TABLE, USER-SPACE CONTROLLER.

The object of the study is the process of managing computational resources in the Linux operating system under dynamic workload conditions.

The subject of the study comprises methods and software tools for adaptive process management in Linux at the user-space level using reinforcement learning.

The aim of this thesis is to investigate and develop an experimental approach to adaptive process management in Linux based on a user-space controller employing a Markov decision process model and the Q-learning method for selecting control actions.

The thesis analyzes Linux process and resource management mechanisms, including nice/renice, cgroups v2, cpu.weight, cpu.max, /proc, and sysfs, and substantiates the feasibility of applying reinforcement learning to adaptive control under varying workloads.

The adaptive control problem is formalized as a Markov decision process. A mathematical model is proposed that includes a representation of system states, a set of control actions, and a reward function based on CPU utilization, system load, and throttling indicators. A tabular Q-learning algorithm is employed to select control actions.

A software prototype of a user-space adaptive controller has been developed. The prototype collects system metrics, constructs the environment state, selects and applies control actions using standard Linux mechanisms, and stores experimental results.

The conducted experimental study confirmed the feasibility of the proposed approach and demonstrated the potential of reinforcement learning for adaptive process management without modifying the Linux kernel.

The practical significance of the study lies in the development of an experimental software prototype that can serve as a basis for further research in adaptive resource management, automated workload monitoring, and the development of intelligent tools for process management support in Linux operating systems.

ЗМІСТ

РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ АДАПТИВНОГО	
КЕРУВАННЯ ПРОЦЕСАМИ В ОС LINUX	13
1.1 Особливості керування процесами в операційній системі Linux	13
1.2 Механізми керування ресурсами Linux	16
1.3 Проблема статичного керування ресурсами	21
1.4 Можливості застосування методів штучного інтелекту для адаптивного керування	25
1.5 Постановка задачі дослідження	29
Висновки до розділу 1	32
РОЗДІЛ 2 МАТЕМАТИЧНА МОДЕЛЬ АДАПТИВНОГО КЕРУВАННЯ	
ПРОЦЕСАМИ	35
2.1 Формалізація задачі керування як марковського процесу прийняття рішень	35
2.2 Модель стану системи	38
2.3 Простір керуючих дій	43
2.4 Функція винагороди	46
2.5 Алгоритм Q-learning для адаптивного вибору дій	51
2.6 Обмеження запропонованої математичної моделі	56
Висновки до розділу 2	58
РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ	
ЕКСПЕРИМЕНТАЛЬНИХ РЕЗУЛЬТАТІВ	61
3.1 Загальна архітектура програмного прототипу	61
3.2 Реалізація збору системних метрик	65
3.3 Реалізація середовища та агента навчання	68
3.4 Реалізація керуючих дій у Linux	73
3.5 Сценарій проведення експериментів	77

	7
3.6 Аналіз результатів експериментів	81
3.7 Аналіз Q-таблиці та поведінки агента.....	87
Висновки до розділу 3	91
ВИСНОВКИ.....	94
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	96
ДОДАТОК А ОСНОВНІ МОДУЛІ ПРОГРАМНОГО ПРОТОТИПУ	98
ДОДАТОК Б МОДУЛІ ВИКОНАННЯ КЕРУЮЧИХ ДІЙ ТА ОБРОБЛЕННЯ РЕЗУЛЬТАТІВ.....	105

ВСТУП

Сучасні операційні системи виконують велику кількість процесів, які одночасно конкурують за ресурси обчислення, оперативну пам'ять машини, доступ до файлової системи та інші системні можливості¹. Особливо актуальною ця проблема є для Linux-систем, що широко використовуються у серверній інфраструктурі, хмарних середовищах, контейнеризованих платформах, вбудованих системах і дослідницьких обчислювальних середовищах. В таких умовах навантаження на систему часто є нерівномірним і динамічним: одні процеси можуть тимчасово збільшувати споживання процесорного часу, інші – переходити у фоновий режим, а загальний стан системи змінюється залежно від кількості активних задач, їхніх пріоритетів і типу виконання.

Операційна система Linux надає низку механізмів для керування процесами та розподілу ресурсів між ними. До таких механізмів належать пріоритети `nice/renice`, `control groups`, параметри `cru.weight` та `cru.max`, а також системні інтерфейси `/proc` і `sysfs`. Вони дають змогу обмежувати або коригувати використання ресурсів окремими процесами чи групами процесів. Проте в реальності значна частина таких налаштувань задається заздалегідь і не завжди враховує поточний стан системи. У реальних сценаріях статичні правила можуть бути недостатньо ефективними, оскільки оптимальні параметри керування залежать від, поведінки процесів, поточного навантаження і цільових критеріїв роботи системи.

¹Тут і нижче використано такий інструмент штучного інтелекту як чат-бот з генеративним штучним інтелектом ChatGPT виключно для корегування та редагування тексту, створеного автором цієї дипломної роботи, на основі автоматизованої перевірки граматики, структури та стилю, що відповідає Політиці використання штучного інтелекту для академічної діяльності в КПП ім. Ігоря Сікорського (протокол №11 Вченої ради КПП ім. Ігоря Сікорського від 11 грудня 2023 р.).

У зв'язку з цим, дослідження підходів до адаптивного керування процесами, за яких керуючі дії змінюються відповідно до спостережуваного стану середовища є актуальним. Одним із перспективних напрямів цього підходу є використання методів штучного інтелекту, зокрема навчання з підкріпленням. На відміну від класичних алгоритмів із фіксованими правилами, навчання з підкріпленням дає змогу подати процес керування як послідовність рішень, у якій агент спостерігає стан системи, обирає дію, отримує оцінку її ефективності та поетапно коригує стратегію поведінки системи.

У межах даної роботи розглядається побудова експериментального адаптивного контролера простору користувача (user-space adaptive controller). Такий контролер збирає системні метрики, формує дискретний стан системи, обирає керуючу дію та застосовує її через стандартні механізми Linux. Основна ідея полягає в тому, щоб поєднати доступні засоби керування ресурсами операційної системи з адаптивною логікою прийняття рішень, яка враховує зміну навантаження під час роботи.

Математично задача адаптивного керування процесами формалізується як марковський процес прийняття рішень. У такій моделі стан відповідає дискретизованому опису поточного навантаження системи, дія визначає потенційний керуючий вплив, а винагорода відображає якість прийнятого рішення з погляду стабільності та ефективності використання ресурсів. Для адаптивного вибору дій у роботі використовується Q-навчання (Q-learning), що дає змогу формувати таблицю оцінок продуктивності дій і оновлювати її на основі взаємодії агента із середовищем.

Актуальність теми. Актуальність теми зумовлена необхідністю пошуку нових інтелектуальних методів керування обчислювальними ресурсами в Linux-середовищі, що здатні динамічно реагувати на непередбачувані профілі навантаження без реконфігурації системного ядра. Запропонований напрям дослідження є важливим для задач

автоматизованого моніторингу, адаптивного розподілу ресурсів і підвищення гнучкості керування процесами в контексті змінного навантаження.

Об'єкт дослідження. Об'єктом дослідження є процеси керування обчислювальними ресурсами в операційній системі Linux в умовах динамічного навантаження.

Предмет дослідження. Предметом дослідження є методи та програмні засоби адаптивного керування процесами в Linux на рівні простору користувача з використанням навчання з підкріпленням.

Мета роботи. Метою роботи є дослідження та розроблення експериментального підходу до адаптивного керування процесами в ОС Linux на основі контролера простору користувача, який використовує математичну модель марковського процесу прийняття рішень і метод Q-навчання для вибору керуючих дій.

Для досягнення поставленої мети необхідно виконати такі завдання:

- проаналізувати проблемну область керування процесами та ресурсами в операційній системі Linux;
- дослідити стандартні механізми Linux, які можуть використовуватися для впливу на виконання процесів, зокрема cgroups v2, cpu.weight, cpu.max, nice/renice, /proc та sysfs;
- проаналізувати можливості застосування методів штучного інтелекту до задачі адаптивного керування системними ресурсами;
- формалізувати задачу адаптивного керування процесами як марковський процес прийняття рішень;
- розробити математичну модель станів, дій, винагороди та стратегії керування для експериментального середовища;
- реалізувати прототип адаптивного контролера простору користувача для збору системних метрик, формування стану системи та вибору керуючих дій;

- провести експериментальне дослідження роботи прототипу в умовах змінного навантаження;
- проаналізувати отримані результати та визначити обмеження запропонованого підходу;
- провести функціонально-вартісний аналіз розробленого програмного рішення.

Методи дослідження. Методи дослідження включають аналіз наукових і технічних джерел з питань керування процесами в Linux, методи системного аналізу для визначення структури задачі та взаємозв'язків між її компонентами, методи математичного моделювання для формалізації задачі як марковського процесу прийняття рішень, методи навчання з підкріпленням для адаптивного вибору дій, а також експериментальні інженерні методи для оцінювання поведінки контролера під час взаємодії з динамічними профілями навантаження.

Елементи наукової новизни отриманих результатів. Елементи наукової новизни отриманих результатів полягають у подальшому розвитку підходів до адаптивного керування процесами в Linux шляхом поєднання стандартних механізмів керування ресурсами операційної системи з моделлю прийняття рішень на основі Q-навчання. У роботі запропоновано формалізацію задачі керування процесами як марковського процесу прийняття рішень, у межах якого стан системи формується на основі дискретизованих системних метрик, а вибір керуючих дій здійснюється з використанням таблиці Q-значень.

Практичне значення отриманих результатів. Практичне значення отриманих результатів полягає в розробленні експериментального прототипу адаптивного контролера простору користувача, який демонструє можливість використання навчання з підкріпленням для прийняття рішень щодо керування процесами в Linux. Запропонований підхід не є власним планувальником ядра Linux, а використовує стандартні системні механізми,

зокрема cgroups v2, cpu.weight, cpu.max, nice/renice, /proc та sysfs. Отримані результати можуть бути використані як основа для подальших досліджень у сфері адаптивного керування системними ресурсами, побудови інтелектуальних механізмів моніторингу навантаження та створення допоміжних інструментів для автоматизованого керування параметрами виконання процесів.

Структура та обсяг роботи. Пояснювальна записка складається зі вступу, трьох розділів, висновків, переліку використаних джерел із 18 найменувань та 2 додатків.

РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ АДАПТИВНОГО КЕРУВАННЯ ПРОЦЕСАМИ В ОС LINUX

1.1 Особливості керування процесами в операційній системі Linux

Операційна система Linux є багатозадачною системою, у якій одночасно можуть виконуватися багато процесів і потоків. Процес у Linux можна розглядати як екземпляр виконуваної програми, для якого ядро зберігає допоміжну інформацію: ідентифікатор процесу, стан виконання, пріоритет, дані про використання процесорного часу, пам'яті та інші параметри. Значна частина цієї інформації доступна з простору користувача через стандартні системні інтерфейси, зокрема через псевдофайлову систему `/proc` [1; 8–10].

Керування процесами в Linux здійснюється на рівні ядра операційної системи. Ядро відповідає за створення, виконання, призупинення, відновлення та завершення процесів, а також за розподіл процесорного часу між активними задачами [1; 2]. При цьому практичні програми та системні утиліти не взаємодіють безпосередньо з внутрішніми структурами планувальника, а використовують визначені системні виклики, файлові інтерфейси та утиліти. Такий підхід надає контрольований доступ до механізмів керування процесами та зменшує ризик порушення стабільної роботи операційної системи.

Одним із основних компонентів керування процесами є планувальник задач. Його завдання полягає у виборі процесу або потоку, якому буде розподілено частину процесорного часу у певний момент. У Linux рішення щодо планування приймаються з урахуванням правил планування, пріоритетів та поточного стану задачі [2]. Для процесів, які не належать до класу реального часу, важливу роль відіграють механізми справедливого розподілу процесорного часу між задачами.

Упродовж тривалого часу для звичайних процесів Linux використовувався повністю справедливий планувальник (Completely Fair Scheduler, CFS). Його принцип роботи базується на математичній моделі ідеально багатозадачного процесора, відповідно до якої процесорний час має розподілятися між задачами пропорційно та справедливо [3]. Для оцінювання того, скільки процесорного часу вже отримала задача, CFS використовує поняття віртуального часу виконання. Це дає змогу планувальнику надавати перевагу тим задачам, які отримали менше процесорного часу порівняно з іншими.

У сучасних версіях ядра Linux розвивається механізм найранішого придатного віртуального дедлайну (Earliest Eligible Virtual Deadline First, EEVDF). Цей підхід також спрямований на справедливе планування, але додатково використовує поняття придатності задачі до виконання, віртуального часу та віртуальних дедлайнів [4]. Перехід до таких механізмів свідчить про те, що планування процесів у Linux є складною задачею, яка враховує не лише факт наявності активних процесів, а й їхню історію виконання, пріоритети та відносні права на процесорний час.

Важливим параметром звичайних процесів є *нісе-рівень*. Він використовується для зміни відносного пріоритету процесу під час планування. У Linux *нісе-рівень* має діапазон від -20 до +19: менші значення відповідають вищому пріоритету, а більші – нижчому [6]. Для зміни *нісе-рівня* вже запущених процесів використовується команда *renice*, яка дає змогу змінювати пріоритет окремого процесу, групи процесів або процесів певного користувача [7]. Водночас *нісе-рівень* не задає жорсткого обмеження використання процесора, а лише впливає на відносне положення процесу серед інших задач під час планування.

У документації Linux окремо підкреслюється, що механізм *нісе* пов'язаний із загальною логікою справедливого розподілу процесорного часу [5]. Тому його доцільно розглядати як інструмент м'якого впливу на

виконання процесів. Він може бути корисним для зміни пріоритетності задач, але не забезпечує точного або гарантованого обмеження використання CPU. Для таких цілей у Linux застосовуються інші механізми, зокрема групове керування ресурсами.

Для отримання інформації про процеси та стан системи Linux надає псевдофайлову систему `/proc`. Вона є інтерфейсом до даних ядра та містить інформацію про активні процеси, параметри системи, використання ресурсів і стан окремих підсистем [8]. Наприклад, файли `/proc/[pid]/stat` та `/proc/[pid]/status` містять дані про стан процесу, його ідентифікатор, пріоритет, пісе-рівень, кількість потоків та інші характеристики [9; 10]. Завдяки цьому `/proc` може використовуватися як джерело метрик для аналізу дій процесів у часі.

Окрім `/proc`, у Linux використовується файлова система `sysfs`, яка експортує об'єкти ядра, їхні атрибути та зв'язки між ними у простір користувача [11]. На відміну від звичайної файлової системи, `sysfs` призначена не для зберігання користувацьких даних, а для представлення структур ядра у вигляді набору файлів і каталогів. Це надає можливість отримувати службову інформацію про пристрої, підсистеми та окремі параметри ядра без безпосередньої модифікації його коду.

Ще одним важливим механізмом керування ресурсами є контрольні групи (control groups, cgroups). З ними можна об'єднувати процеси у групи та застосовувати до них правила обліку, обмеження або розподілу ресурсів [12; 13]. У другій версії цього механізму, `cgroups v2`, для керування використанням процесора передбачено параметри `cpu.weight` та `cpu.max`. Параметр `cpu.weight` задає відносну вагу групи під час розподілу процесорного часу, а `cpu.max` дає змогу встановити верхню межу використання CPU для групи процесів [12].

Таким чином, керування процесами в Linux має багаторівневу структуру. На рівні ядра виконується планування задач і розподіл

процесорного часу між активними процесами. На рівні простору користувача доступні стандартні інтерфейси для спостереження за станом процесів і зміни окремих параметрів їх виконання. До таких механізмів належать `/proc`, `sysfs`, `nice/renice` та `cgroups v2` [6–13]. Сукупність цих засобів створює основу для подальшого аналізу механізмів керування ресурсами Linux і переходу до проблеми їх статичного налаштування.

1.2 Механізми керування ресурсами Linux

Операційна система Linux дає перелік базових механізмів для спостереження за процесами та керування використанням системних ресурсів. До них належать зміна пріоритетів процесів, отримання інформації через псевдофайлові системи, а також групове керування ресурсами за допомогою контрольних груп. Зазначені засоби не замінюють планувальник ядра, однак дозволяють впливати на поведінку процесів із простору користувача через визначені інтерфейси операційної системи [6–13].

Необхідність застосування таких механізмів зумовлена архітектурними особливостями багатозадачних середовищ. У межах однієї системи можуть одночасно виконуватися прикладні сервіси, фонові службові процеси, системні демони, обчислювальні задачі та процеси користувача. У разі надмірного використання ресурсів CPU одним із фонових процесів може погіршуватися час отримання відповіді більш важливих служб. Тому операційна система має надавати засоби не лише для запуску та завершення процесів, а й для регулювання їхньої пріоритетності, обмеження використання ресурсів і збору інформації про поточний стан системи.

Одним із базових механізмів впливу на виконання процесів є параметр `nice`, або `nice-рівень`. Він визначає відносну пріоритетність процесу під час

планування. У Linux nice-рівень має діапазон від -20 до +19: менші значення відповідають вищому пріоритету, а більші – нижчому [6]. Наприклад, фонову задачу архівації, резервного копіювання або обробки журналів доцільно виконувати з нижчим пріоритетом, щоб за наявності конкуренції за CPU перевагу отримували важливіші інтерактивні або серверні процеси.

Для зміни пріоритету вже запущених процесів використовується команда `renice`. Вона може застосовуватися до окремого процесу, групи процесів або процесів певного користувача [7]. Це дає змогу коригувати поведінку процесів без їхнього завершення. Наприклад, якщо службовий процес під час виконання починає активно використовувати CPU, адміністратор або автоматизований контролер може змінити його nice-рівень, зменшивши вплив цього процесу на інші задачі системи.

Разом із тим механізм `nice/renice` має обмеження. Він не встановлює точну межу використання процесора, а лише змінює відносну пріоритетність процесу серед інших задач. Якщо в системі відсутня конкуренція за CPU, процес із нижчим пріоритетом усе одно може використовувати значну частину процесорного часу. Отже, nice-рівень доцільно розглядати як засіб м'якого регулювання, а не як механізм жорсткого контролю та обмеження ресурсів [5–7].

Для прийняття рішень про керування процесами необхідно мати інформацію про їх поточний стан. Одним із основних джерел такої інформації в Linux є псевдофайлова система `/proc`. Вона надає доступ до даних ядра у вигляді файлів і каталогів, що дозволяє отримувати інформацію про активні процеси, параметри системи, використання ресурсів і стан окремих підсистем [8]. Практичне значення псевдофайлової системи `/proc` полягає у забезпеченні стандартизованого інтерфейсу доступу до внутрішніх структур даних ядра з простору користувача.

Каталог `/proc/[pid]` містить інформацію про конкретний процес із відповідним ідентифікатором процесу (Process ID, PID). Наприклад, файл `/proc/[pid]/stat` містить компактний набір числових параметрів процесу, а файл `/proc/[pid]/status` подає частину інформації у більш зручному для читання форматі [9; 10]. Через ці файли можна отримати дані про стан процесу, його ідентифікатор, пріоритет, пісе-рівень, кількість потоків та інші характеристики.

Дані з `/proc` можуть використовуватися для моніторингу процесів, побудови статистики та виявлення змін у поведінці навантаження. Наприклад, якщо протягом певного проміжку часу спостерігається зростання активності окремого процесу, система моніторингу може зафіксувати зміну відповідних показників і передати ці дані до модуля прийняття рішень. На основі таких даних може бути визначено подальшу дію: залишити параметри процесу без змін, знизити його пріоритет або застосувати до нього групове обмеження ресурсів.

Окрім `/proc`, у Linux використовується файлова система `sysfs`. Вона призначена для представлення об'єктів ядра, їхніх атрибутів і зв'язків між ними у просторі користувача [11]. На відміну від звичайних файлових систем, `sysfs` не використовується для зберігання прикладних даних. Її призначення полягає у наданні структурованого доступу до службової інформації про пристрої, драйвери, підсистеми ядра та окремі параметри системи.

У контексті керування процесами файлова система `sysfs` не є основним джерелом інформації про процеси за їхніми PID, як `/proc`. Водночас вона доповнює загальну інформаційну модель системи, оскільки дозволяє отримувати дані про окремі підсистеми, пристрої та параметри ядра. Це може бути корисним у складніших сценаріях моніторингу, де потрібно враховувати не лише характеристики процесів, а й стан системного середовища загалом.

Для більш гнучкого керування ресурсами Linux підтримує механізм контрольних груп (control groups, cgroups). Він дозволяє об'єднувати процеси у групи та застосовувати до них правила обліку, обмеження або розподілу ресурсів [12; 13]. На відміну від пісе-рівня, який впливає переважно на відносний пріоритет окремого процесу, cgroups дають змогу керувати ресурсами на рівні групи процесів. Це має особливе значення для серверних систем, контейнеризованих середовищ і сценаріїв, у яких потрібно ізолювати або обмежити окремі навантаження.

Наприклад, вебсервер і фоновий обчислювальний процес можуть бути розміщені в різних контрольних групах. Якщо фонове навантаження починає надмірно використовувати CPU, можна обмежити саме відповідну групу процесів, не змінюючи параметри інших служб у системі. Такий підхід є зручнішим, ніж індивідуальне налаштування кожного процесу, оскільки один сервіс у реальних умовах може складатися з кількох процесів або потоків.

У cgroups v2 використовується єдина ієрархія контрольних груп, що спрощує структуру керування ресурсами порівняно з попередньою версією. Для керування використанням процесора важливими є параметри `cpu.weight` та `cpu.max`. Параметр `cpu.weight` задає відносну вагу групи під час розподілу процесорного часу між групами. Чим більша вага, тим більшу частку CPU може отримати група порівняно з іншими групами за наявності конкуренції за ресурс [12].

Параметр `cpu.weight` можна сприймати як механізм пропорційного розподілу процесорного часу. Наприклад, якщо одна контрольна група має більшу вагу, ніж інша, то під час конкуренції за CPU їй надаватиметься більша частка процесорного часу. Водночас, якщо система не перевантажена, група з меншою вагою також може використовувати доступний ресурс. Тому `cpu.weight` доцільно застосовувати для відносного регулювання використання CPU між групами процесів.

Параметр `cpu.max` використовується для встановлення верхньої межі використання CPU групою процесів. На відміну від `cpu.weight`, який визначає відносний розподіл ресурсу, `cpu.max` дає змогу задіяти жорсткіше обмеження. Це корисно у випадках, коли потрібно запобігти надмірному використанню процесора окремою групою процесів або забезпечити більш передбачувану поведінку системи за умов високого навантаження [12].

Наприклад, якщо фонове навантаження не повинно перевищувати визначену частку процесорного часу, його можна розмістити в окремій контрольній групі та встановити відповідне обмеження через `cpu.max`. У результаті навіть за високої активності цієї групи вона не зможе перевищити задану межу. Такий механізм є важливим для реалізації підсистеми адаптивного керування, оскільки дозволяє не лише змінювати відносні пріоритети, а й прямо обмежувати використання CPU.

Основні механізми керування ресурсами Linux наведено в таблиці 1.1.

Таблиця 1.1 – Механізми керування ресурсами Linux

<i>Механізм</i>	<i>Призначення</i>	<i>Приклад використання</i>
<code>nice</code>	Задання початкового піс-рівня процесу	Запуск фонові задачі з нижчим пріоритетом
<code>renice</code>	Зміна піс-рівня вже запущеного процесу	Зниження пріоритету процесу, який надмірно використовує CPU
<code>/proc</code>	Отримання інформації про процеси та стан системи	Зчитування параметрів процесу через <code>/proc/[pid]/stat</code> або <code>/proc/[pid]/status</code>
<code>sysfs</code>	Доступ до атрибутів об'єктів ядра	Отримання службової інформації про підсистеми та пристрої
<code>cgroups v2</code>	Групове керування ресурсами процесів	Обмеження або розподіл ресурсів для групи процесів
<code>cpu.weight</code>	Задання відносної ваги групи процесів	Надання одній групі більшої частки CPU під час конкуренції
<code>cpu.max</code>	Встановлення верхньої межі використання CPU	Обмеження фонового навантаження заданою часткою CPU

Таким чином, Linux надає кілька рівнів керування ресурсами. Механізми `nice` та `renice` дозволяють змінювати відносний пріоритет окремих процесів, `/proc` і `sysfs` забезпечують доступ до системної інформації, а `cgroups v2` дають змогу керувати ресурсами на рівні груп процесів. Зазначені механізми можуть використовуватися разом: наприклад, `/proc` може бути джерелом метрик, а `renice` або `cgroups v2` – засобами впливу на виконання процесів.

Сукупність цих інструментів створює технічну основу для побудови адаптивного контролера, який може спостерігати за станом системи та змінювати параметри керування відповідно до поточного навантаження. При цьому всі дії виконуються через стандартні механізми Linux, що робить такий підхід придатним для дослідження адаптивного керування без модифікації ядра операційної системи.

1.3 Проблема статичного керування ресурсами

Традиційне керування ресурсами в операційних системах часто базується на попередньо заданих параметрах. Адміністратор або розробник може вручну визначити пріоритет процесу, обмежити використання CPU для групи процесів або задати відносну вагу умовного навантаження. У Linux для цього можуть застосовуватися такі механізми, як `nice`, `renice`, `cpu.weight` і `cpu.max` [6; 7; 12]. Такий підхід є технічно простим, однак його ефективність значною мірою залежить від того, наскільки точно початкові налаштування відповідають реальній поведінці системи.

Основна проблема статичного керування полягає в тому, що параметри ресурсів задаються наперед і не завжди змінюються відповідно до поточного стану навантаження. У багатозадачному середовищі кількість активних

процесів, інтенсивність використання CPU, обсяг фонових задач і характер взаємодії між сервісами можуть змінюватися в часі. Тому параметри, які є допустимі для одного сценарію роботи, можуть бути неефективними за інших умов.

Зокрема, фоновий процес обробки даних може не створювати суттєвого навантаження в нормальному режимі, але під час запуску інтенсивної обчислювальної задачі почати активно використовувати процесорний час. Якщо для такого процесу не задано відповідних обмежень, він може конкурувати з більш важливими службами. Водночас надто жорстке обмеження може призвести до ефекту CPU throttling, тобто примусового обмеження виділених квантів процесорного часу. У результаті процес може виконуватися повільніше, а затримки обробки задач можуть зростати навіть у випадках, коли частина ресурсів системи використовується не повністю [12; 18].

Подібна проблема виникає і при застосуванні пісе-рівнів. Зміна пісе-рівня дозволяє вплинути на відносну пріоритетність процесу, однак не гарантує точного обмеження використання CPU [5–7]. Якщо система перевантажена, підвищення або зниження пріоритету може впливати на розподіл процесорного часу між задачами. Водночас за відсутності конкуренції процес із нижчим пріоритетом може продовжувати використовувати значну частину CPU. Це обмежує застосування пісе-рівня як єдиного механізму керування ресурсами.

Механізм `cgroups v2` надає ширші можливості для керування групами процесів. Зокрема, параметр `cpu.weight` дозволяє визначати відносну частку CPU для групи процесів, а `cpu.max` – встановлювати верхню межу використання процесорного часу [12; 13]. Проте навіть ці механізми не усувають проблему вибору правильних значень параметрів. Якщо значення `cpu.max` є занадто низьким, група процесів може бути необґрунтовано обмежена, що підвищує ризик CPU throttling і збільшення затримок

виконання. Якщо ж обмеження занадто високе, воно може не запобігти надмірному використанню CPU під час пікового навантаження.

Статичні параметри особливо складно визначати для систем зі змінним або нерівномірним навантаженням. У таких системах один і той самий сервіс у різні моменти часу може виконувати різні типи роботи: обробляти запити користувачів, виконувати фонову синхронізацію, запускати періодичні задачі або взаємодіяти з іншими службами. У результаті фактична потреба в ресурсах змінюється, тоді як заздалегідь задані обмеження залишаються незмінними.

Окрему увагу в таких умовах слід приділяти затримкам виконання. Для інтерактивних і серверних систем важливим є не лише середній час обробки запитів, а й хвостові затримки (tail latency), тобто затримки для найповільніших запитів або операцій [18]. Навіть якщо середні показники продуктивності залишаються прийнятними, поява окремих тривалих затримок може негативно впливати на сприйняття роботи сервісу користувачем або на стабільність взаємодії між компонентами системи. У випадку жорстких статичних обмежень CPU такі затримки можуть зростати через примусове обмеження процесорного часу для відповідної групи процесів.

У дослідженнях з керування обчислювальними ресурсами зазначається, що ручне або фіксоване налаштування параметрів часто не забезпечує оптимального використання ресурсів у динамічних середовищах [16; 17]. Зокрема, у системах великого масштабу поширеною є ситуація, коли ресурси резервуються із запасом, щоб уникнути деградації продуктивності. Такий підхід підвищує стабільність, але може призводити до неефективного використання CPU та пам'яті [17]. З іншого боку, недостатнє виділення ресурсів може спричинити затримки, уповільнення роботи сервісів або нестабільну поведінку системи.

Проблема статичного керування також пов'язана з тим, що адміністратор не завжди може заздалегідь передбачити всі можливі стани системи. Навіть якщо початкові параметри підібрано на основі тестування, реальне навантаження може відрізнитися від тестового. Розглянемо ситуацію, коли змінюється кількість користувачів, інтенсивність запитів, характер фонових задач або співвідношення між процесами, які конкурують за CPU. За таких умов фіксовані налаштування можуть швидко втрачати актуальність.

Ще одним недоліком статичного підходу є необхідність ручного втручання. У разі деградації продуктивності виникає потреба локалізувати вузьке місце (bottleneck), проаналізувати метрики процесів і виконати реконфігурацію параметрів. Такий реактивний підхід потребує часу та є недостатньо ефективним в умовах швидких змін навантаження. Крім того, ручні зміни можуть бути суб'єктивними, оскільки залежать від досвіду адміністратора та повноти доступної інформації про стан системи.

З погляду побудови адаптивної системи, ключовим обмеженням статичного керування є відсутність зворотного зв'язку. Параметри задаються один раз або змінюються вручну, але система не аналізує результат попередніх дій і не використовує його для подальшого покращення керування. У динамічному середовищі більш доцільним є підхід, за якого рішення приймаються на основі поточних метрик, а подальші дії коригуються залежно від зміни стану системи.

Таким чином, статичне керування ресурсами є придатним для простих або передбачуваних сценаріїв, однак має суттєві обмеження в умовах змінного навантаження. Основними проблемами такого підходу є складність попереднього вибору параметрів, недостатня гнучкість, потреба в ручному втручанні, ризик CPU throttling, збільшення затримок виконання та відсутність механізму самостійної адаптації до поточного стану системи. Це обґрунтовує необхідність переходу до адаптивного керування, у якому

рішення щодо зміни параметрів процесів або груп процесів приймаються на основі системних метрик і результатів попередніх дій.

1.4 Можливості застосування методів штучного інтелекту для адаптивного керування

Обмеження статичного керування ресурсами зумовлюють потребу в підходах, здатних враховувати зміну стану системи в часі. У таких умовах доцільним є перехід від заздалегідь заданих параметрів до механізмів адаптивного прийняття рішень. Одним із напрямів, який може бути використаний для цього, є застосування методів штучного інтелекту, зокрема навчання з підкріпленням (Reinforcement Learning, RL).

Навчання з підкріпленням розглядає процес керування як послідовну взаємодію агента із середовищем. Агент спостерігає поточний стан середовища, обирає певну дію, отримує винагороду та переходить до нового стану [14]. На відміну від алгоритмів, які працюють за наперед визначеними правилами, RL-підхід дає змогу коригувати поведінку системи на основі результатів попередніх дій. Це робить його придатним для задач, у яких складно заздалегідь описати всі можливі стани середовища та оптимальні реакції на них.

У контексті керування ресурсами операційної системи середовищем може виступати сама система Linux або її окрема підсистема, станом – набір системних метрик, а діями – зміна параметрів керування процесами чи групами процесів. Стан може враховувати рівень використання CPU, наявність примусового обмеження процесорного часу, динаміку навантаження або інші показники, отримані через стандартні інтерфейси Linux. Діями можуть бути зміна пісе-рівня процесу, коригування

`cpu.weight` або встановлення нового значення `cpu.max` для контрольної групи [6; 7; 12].

Ключовим елементом RL є функція винагороди. Вона визначає, наскільки корисною була дія агента з погляду поставленої мети [14]. Для задачі керування ресурсами винагорода може враховувати кілька критеріїв: стабільність використання CPU, зменшення надмірного навантаження, уникнення CPU throttling, зниження затримок виконання або підтримання прийняттого рівня продуктивності. Такий підхід дає змогу формалізувати задачу керування не лише як обмеження процесів, а як пошук балансу між продуктивністю, стабільністю та ефективністю використання ресурсів.

Одним із базових алгоритмів навчання з підкріпленням є Q-навчання (Q-learning). Цей метод був запропонований як підхід до навчання оптимальної поведінки агента у дискретному середовищі без необхідності попередньо знати повну модель переходів між станами [15]. Основна ідея Q-навчання полягає в оцінюванні корисності виконання певної дії в певному стані. Для цього використовується функція якості дії Q, яка показує очікувану довгострокову вигоду від вибору дії.

У табличному варіанті Q-навчання значення функції Q зберігаються у вигляді Q-таблиці (Q-table). Рядки такої таблиці відповідають можливим станам середовища, а стовпці – доступним діям. Після виконання дії агент отримує винагороду та оновлює відповідне значення Q. Узагальнено правило оновлення можна подати так:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)],$$

де s – поточний стан,

a – виконана дія,

r – отримана винагорода,

s' – новий стан,

α – коефіцієнт навчання,

γ – коефіцієнт дисконтування майбутньої винагороди [14; 15].

Завдяки такому оновленню агент поступово накопичує інформацію про те, які дії є доцільнішими в певних станах.

Для задачі адаптивного керування процесами Q-таблиця має практичну перевагу, оскільки є відносно простою для реалізації та інтерпретації. На відміну від складних нейромережових моделей, табличний підхід дозволяє явно бачити відповідність між станами, діями та накопиченими оцінками корисності. Це є важливим для експериментального дослідження, у якому потрібно не лише отримати результат, а й пояснити логіку прийняття рішень.

Водночас застосування Q-навчання потребує дискретизації станів і дій. Системні метрики Linux, такі як використання CPU або рівень навантаження, зазвичай мають числовий і неперервний характер. Для використання Q-таблиці їх необхідно перетворити на обмежену кількість дискретних станів, наприклад “низьке навантаження”, “середнє навантаження” або “високе навантаження”. Аналогічно, дії контролера мають бути представлені як скінченний набір можливих варіантів: збільшити вагу CPU, зменшити вагу CPU, послабити обмеження, посилити обмеження або залишити параметри без змін.

Перевагою такого підходу є можливість реалізувати адаптивну поведінку без модифікації ядра операційної системи. Агент може працювати у просторі користувача, отримувати метрики через `/proc` або інші стандартні інтерфейси, а вплив на систему здійснювати через `renice` або параметри `cgroups v2` [6–13]. Таким чином, методи навчання з підкріпленням можуть бути поєднані з наявними механізмами Linux без створення власного планувальника задач.

У сучасних дослідженнях керування ресурсами розглядається як задача онлайн-прийняття рішень у змінному середовищі. Зокрема, у роботі DeepRM показано можливість застосування навчання з підкріпленням для керування

обчислювальними ресурсами та планування завдань у системах зі змінним навантаженням [16]. Такі підходи демонструють, що RL може бути корисним тоді, коли ручні евристики або фіксовані правила недостатньо гнучкі для різних сценаріїв роботи.

Досвід великих інфраструктур також підтверджує актуальність автоматизованого керування ресурсами. У роботі Autopilot, присвяченій автоматичному масштабуванню навантажень у Google, зазначається проблема ручного визначення ресурсних лімітів і потреба в автоматичному коригуванні параметрів відповідно до фактичного використання ресурсів [17]. Хоча такі системи мають інший масштаб і складність, вони підтверджують загальну тенденцію до переходу від ручного налаштування до автоматизованого й адаптивного керування.

Окреме значення для адаптивного керування має врахування затримок виконання. Для серверних та інтерактивних систем важливими є не лише середні показники продуктивності, а й хвостові затримки (tail latency), які можуть суттєво впливати на якість роботи системи [18]. Тому функція винагороди в адаптивному контролері може враховувати не лише рівень використання CPU, а й ознаки перевантаження, throttling або зростання затримок. Це дозволяє уникати рішень, які формально зменшують використання ресурсу, але погіршують фактичну якість виконання процесів.

Водночас застосування методів штучного інтелекту в керуванні ресурсами має певні обмеження. По-перше, якість рішень залежить від правильного вибору метрик, станів, дій і функції винагороди. По-друге, табличне Q-навчання схильне до проблеми вибуху простору станів (state-space explosion), що суттєво обмежує його масштабування в системах із великою кількістю неперервних системних метрик. По-третє, під час навчання агент може виконувати неефективні дії, тому в системах реального часу або промислових середовищах експлуатації (production environments) потрібні додаткові механізми безпеки та контролю.

Отже, методи навчання з підкріпленням не замінюють стандартні механізми Linux, а можуть використовуватися як надбудова для адаптивного вибору дій. У межах задачі адаптивного керування процесами RL дозволяє формалізувати взаємозв'язок між станом системи, дією контролера та отриманим результатом. Застосування Q-навчання і Q-таблиці є доцільним для експериментального прототипу, оскільки забезпечує зрозумілу модель прийняття рішень і може бути реалізоване поверх наявних інтерфейсів Linux.

1.5 Постановка задачі дослідження

Проведений аналіз особливостей керування процесами в Linux показав, що операційна система вже має розвинені механізми планування задач і розподілу процесорного часу на рівні ядра [2–4]. Водночас Linux надає низку інтерфейсів простору користувача, які можуть застосовуватися для спостереження за станом процесів і зміни окремих параметрів їх виконання. До таких інтерфейсів належать псевдофайлова система `/proc`, файлова система `sysfs`, механізми `nice/renice` та контрольні групи `cgroups v2` [6–13].

З урахуванням цього задача дослідження не полягає у розробленні власного планувальника ядра Linux або модифікації механізмів CFS чи EEVDF. Предметом подальшого розгляду є адаптивне керування процесами на рівні простору користувача. Такий підхід передбачає створення програмного контролера, який не втручається у внутрішню логіку планувальника ядра, а використовує наявні механізми операційної системи для збирання метрик і впливу на параметри процесів або груп процесів.

Основна задача дослідження полягає у розробленні експериментального адаптивного контролера простору користувача (`user-`

space adaptive controller) для ОС Linux, який виконує спостереження за системними метриками, формує дискретний стан системи, приймає рішення щодо зміни параметрів керування та застосовує обрану дію через стандартні засоби Linux. Як метод прийняття рішень у межах такого контролера доцільно використати Q-навчання, оскільки воно дозволяє оцінювати корисність дій у певних станах і поступово коригувати політику керування на основі отриманої винагороди [14; 15].

У межах дослідження стан системи може бути представлений через сукупність метрик, що характеризують поточне використання ресурсів і поведінку процесів. До таких метрик можуть належати рівень використання CPU, ознаки перевантаження, факт або інтенсивність CPU throttling, а також інші показники, доступні через стандартні інтерфейси Linux. Такі дані можуть бути отримані з `/proc`, `cgroups v2` або інших системних джерел, що не потребують модифікації ядра операційної системи [8–13].

Дії контролера мають відповідати реальним механізмам впливу, які підтримуються Linux. Зокрема, до множини можливих дій можуть належати зміна пісе-рівня процесу, коригування відносної ваги групи процесів через `cpu.weight`, зміна обмеження використання CPU через `cpu.max` або збереження поточних параметрів без змін [6; 7; 12]. Такий вибір дій забезпечує зв'язок між алгоритмом прийняття рішень і практичними механізмами керування ресурсами.

Функція винагороди в адаптивному контролері має відображати бажану поведінку системи. Її доцільно формувати так, щоб заохочувати стабільне використання CPU, зменшення надмірного навантаження, уникнення примусового обмеження процесорного часу та запобігання зростанню затримок виконання. Це узгоджується з проблемами, розглянутими в попередніх підрозділах: статичні обмеження можуть призводити до CPU throttling, збільшення затримок і неефективного використання ресурсів [12; 18].

Узагальнену постановку задачі адаптивного керування процесами наведено в табл. 1.2.

Таблиця 1.2 – Узагальнена постановка задачі адаптивного керування процесами

<i>Компонент задачі</i>	<i>Зміст у межах дослідження</i>
Середовище	Операційна система Linux та процеси, що виконуються в ній
Джерела даних	/proc, sysfs, cgroups v2, системні метрики процесів і груп процесів
Стан	Дискретизований опис поточного навантаження та поведінки системи
Дії	Зміна nice-рівня, коригування <code>cpu.weight</code> , зміна <code>cpu.max</code> , відсутність змін
Метод прийняття рішень	Q-навчання з використанням Q-таблиці
Винагорода	Оцінка результату дії з урахуванням використання CPU, throttling, стабільності та затримок
Результат	Адаптивна зміна параметрів керування без модифікації ядра Linux

Особливістю сформульованої задачі є поєднання стандартних механізмів Linux із методом навчання з підкріпленням. З одного боку, система спирається на наявні засоби операційної системи, що забезпечує реалістичність і технічну коректність підходу. З іншого боку, використання Q-навчання дозволяє перейти від наперед заданих правил до адаптивного вибору дій на основі поточного стану та попереднього досвіду [14–16].

Водночас запропонований підхід слід розглядати як експериментальний прототип, а не як готове рішення для промислових середовищ експлуатації. Табличне Q-навчання потребує дискретизації станів і дій, а також має обмеження масштабування, пов'язані з проблемою вибуху простору станів. Тому в межах дослідження доцільно зосередитися не на

створенні універсальної системи керування ресурсами, а на дослідженні можливості застосування Q-навчання для прийняття адаптивних рішень у контрольованому середовищі.

Отже, постановка задачі дослідження полягає у створенні та перевірці експериментального адаптивного контролера процесів в ОС Linux, який працює у просторі користувача, використовує системні метрики для формування стану, застосовує Q-таблицю для вибору дії та впливає на параметри виконання процесів через стандартні механізми Linux. Такий підхід дозволяє дослідити можливості адаптивного керування ресурсами без втручання у планувальник ядра операційної системи.

Таким чином, математична постановка задачі зводиться до знаходження такої оптимальної стратегії керування π , яка максимізує очікувану сумарну дисконтовану винагороду агента впродовж усього процесу взаємодії із середовищем. Детальну формалізацію відповідного математичного апарату наведено у розділі 2.

Висновки до розділу 1

У першому розділі було проаналізовано проблемну область адаптивного керування процесами в операційній системі Linux. Розглянуто особливості керування процесами, роль планувальника задач, механізми пріоритетів, а також інтерфейси простору користувача, які можуть застосовуватися для спостереження за станом системи та зміни параметрів виконання процесів.

Встановлено, що Linux уже має розвинені механізми планування задач на рівні ядра, зокрема CFS та EEVDF. Тому розроблення власного планувальника ядра не є метою цього дослідження. Натомість доцільним є

використання наявних інтерфейсів операційної системи, таких як `/proc`, `sysfs`, `nice/renice` та `cgroups v2`. Вони забезпечують можливість отримання системних метрик і впливу на параметри процесів або груп процесів без модифікації ядра Linux.

Проаналізовано механізми `cpu.weight` і `cpu.max`, які застосовуються у `cgroups v2` для відносного розподілу процесорного часу та встановлення верхньої межі використання CPU. Показано, що ці механізми є придатними для побудови адаптивного контролера, оскільки дозволяють керувати ресурсами на рівні груп процесів. Водночас статичне задання таких параметрів не завжди є ефективним у системах зі змінним навантаженням.

Обґрунтовано проблему статичного керування ресурсами. Основними недоліками такого підходу є складність попереднього вибору параметрів, відсутність зворотного зв'язку, потреба в ручному втручанні, ризик CPU throttling і збільшення затримок виконання. Це особливо важливо для багатозадачних систем, у яких профіль навантаження може змінюватися в часі, а фіксовані значення параметрів швидко втрачають ефективність.

Розглянуто можливість застосування методів штучного інтелекту, зокрема навчання з підкріпленням, для задач адаптивного керування ресурсами. Показано, що Q-навчання може бути використане для вибору дій на основі поточного стану системи, отриманої винагороди та попереднього досвіду. Для експериментального прототипу доцільним є використання Q-таблиці, оскільки такий підхід є достатньо простим для реалізації, інтерпретації та подальшого аналізу результатів.

У результаті сформульовано задачу дослідження: створити експериментальний адаптивний контролер процесів в ОС Linux, який працює у просторі користувача, збирає системні метрики, формує дискретний стан, обирає керуючу дію на основі Q-таблиці та застосовує її через стандартні механізми Linux. Математично задача зводиться до знаходження

оптимальної стратегії керування, яка максимізує очікувану сумарну дисконтовану винагороду агента під час взаємодії із середовищем.

Сформульовані висновки та визначені обмеження створюють концептуальне підґрунтя для розроблення формальної математичної моделі адаптивного контролера, що є предметом розгляду в наступному розділі.

РОЗДІЛ 2 МАТЕМАТИЧНА МОДЕЛЬ АДАПТИВНОГО КЕРУВАННЯ ПРОЦЕСАМИ

2.1 Формалізація задачі керування як марковського процесу прийняття рішень

Задача адаптивного керування процесами в операційній системі Linux у межах цієї роботи розглядається як задача послідовного прийняття рішень. Адаптивний контролер періодично отримує інформацію про поточний стан системи, обирає керуючу дію та оцінює її наслідки за допомогою числової винагороди. Такий підхід дає змогу формалізувати процес керування як марковський процес прийняття рішень.

У цій роботі важливо підкреслити, що запропонована модель не передбачає створення або модифікацію планувальника ядра Linux. Керування виконується на рівні простору користувача. Контролер лише спостерігає за системними метриками та впливає на процеси через стандартні механізми Linux, зокрема `cgroups v2`, параметри `cpu.weight` і `cpu.max`, `nice/renice`, а також дані з `/proc` і `sysfs`.

Марковський процес прийняття рішень можна подати у вигляді такої п'ятірки:

$$M = (S, A, P, R, \gamma), \quad (2.1)$$

де M – модель марковського процесу прийняття рішень;

S – множина можливих станів системи;

A – множина доступних керуючих дій;

P – функція переходів між станами;

R – функція винагороди;

γ – коефіцієнт дисконтування майбутніх винагород.

Оскільки внутрішні механізми перемикавання контексту в ядрі Linux та інтенсивність надходження запитів є стохастичними і заздалегідь невідомими, точний аналітичний вигляд функції переходів P визначити неможливо. Саме тому в роботі обґрунтовано застосування підходу *model-free* (навчання без моделювання середовища), за якого агент оптимізує свою політику виключно на основі результатів взаємодії з операційною системою.

У межах цієї задачі середовищем є операційна система Linux або її спрощена модель у режимі симуляції (*simulation mode*). Агентом виступає *user-space adaptive controller*, який не втручається безпосередньо в роботу планувальника ядра, а приймає рішення на основі зовнішнього спостереження за станом системи.

На кожному дискретному кроці часу t агент отримує стан s_t , обирає дію a_t , після чого система переходить у новий стан s_{t+1} , а агент отримує винагороду r_t . Загальна схема взаємодії може бути описана так:

$$s_t \in S, \quad (2.2)$$

$$a_t \in A, \quad (2.3)$$

$$s_{t+1} \sim P(s_{t+1} \vee s_t, a_t), \quad (2.4)$$

$$r_t = R(s_t, a_t, s_{t+1}), \quad (2.5)$$

де s_t – стан системи на кроці t ;

a_t – дія, обрана контролером на кроці t ;

s_{t+1} – новий стан системи після виконання дії;

$P(s_{t+1} \vee s_t, a_t)$ – ймовірність переходу до стану s_{t+1} за умови поточного стану s_t і дії a_t ;

r_t – винагорода, отримана після виконання дії;

$R(s_t, a_t, s_{t+1})$ – функція, яка оцінює якість прийнятого рішення.

Теоретично функція переходів P описує, як саме змінюється стан системи після виконання певної керуючої дії. Однак для реальної операційної системи Linux аналітичний вигляд цієї функції заздалегідь невідомий. Це пов'язано з тим, що внутрішні механізми перемикавання контексту, активність процесів, інтенсивність запитів до додатків, поведінка планувальника ядра та зміни навантаження мають стохастичний характер. User-space контролер не має повної інформації про внутрішній стан ядра і не будує точну математичну модель усіх процесів, що відбуваються в системі.

Саме тому в роботі доцільним є використання підходу model-free, тобто підходу без явного моделювання функції переходів середовища. У цьому випадку агент не потребує попередньо заданої моделі P , а навчається на основі досвіду взаємодії із середовищем: спостережуваних станів, виконаних дій та отриманих винагород. Такий підхід відповідає алгоритму Q-learning, який буде розглянуто в наступних підрозділах.

Основною метою агента є вибір такої стратегії керування, яка забезпечує найбільше очікуване значення сумарної дисконтованої винагороди. Політика керування визначає, яку дію потрібно обрати в певному стані системи:

$$\pi(a \vee s) = P(a_t = a \vee s_t = s) \quad (2.6)$$

де $\pi(a \vee s)$ – політика агента;

s – поточний стан системи;

a – можлива керуюча дія;

$P(a_t = a \vee s_t = s)$ – ймовірність вибору дії a у стані s .

Оптимальна політика визначається як така, що максимізує очікувану сумарну дисконтовану винагороду:

$$\pi = \underset{\pi}{\operatorname{argmax}} J(\pi), \quad (2.7)$$

$$J(\pi) = E_{\pi}[\sum_{t=0}^T \gamma^t r_t], \quad (2.8)$$

де π – оптимальна політика керування;

$J(\pi)$ – цільова функція для політики π ;

E_{π} – математичне сподівання за умови використання політики π ;

T – кількість кроків взаємодії агента із середовищем;

γ – коефіцієнт дисконтування, який визначає важливість майбутніх винагород;

r_t – винагорода на кроці t .

У контексті цієї роботи максимізація винагороди означає не досягнення абсолютного максимуму продуктивності системи, а пошук прийнятної адаптивної поведінки контролера. Така поведінка має зменшувати перевантаження, уникати надмірного throttling, підтримувати стабільніше використання CPU та не виконувати зайвих керуючих втручань. Отже, математична модель використовується як основа для побудови експериментального прототипу адаптивного контролера, а не як кінцева модель промислового рівня (production-ready).

2.2 Модель стану системи

Для застосування алгоритму навчання з підкріпленням необхідно визначити, яку саме інформацію агент отримує про середовище. У цій роботі стан системи є спрощеним дискретизованим описом поточного навантаження Linux-середовища або його спрощеної моделі у режимі симуляції (simulation mode). Такий підхід обрано тому, що використовується табличний Q-learning, для якого неперервний простір станів потрібно звести до скінченної кількості дискретних комбінацій.

На практиці операційна система Linux має велику кількість параметрів, які можуть характеризувати її поточний стан: використання CPU, кількість активних процесів, значення load average, рівень throttling, поточні параметри cgroups, пріоритети процесів тощо. Однак включення всіх можливих метрик зробило б модель надмірно складною для досліджуваного експериментального прототипу. Тому в роботі використовується обмежений набір ознак, які безпосередньо пов'язані з керуванням процесами та навантаженням на CPU.

Початковий вектор спостереження можна подати у вигляді:

$$x_t = (u_t, l_t, \theta_t), \quad (2.9)$$

де x_t – вектор спостережуваних параметрів системи на кроці t ;

u_t – поточний або усереднений рівень використання CPU;

l_t – характеристика системного навантаження;

θ_t – показник throttling або ознака обмеження CPU;

Оскільки табличний Q-learning працює зі скінченною множиною станів, числові значення метрик перетворюються на дискретні категорії. У досліджуваному прототипі функція дискретизації D відображає поточні метрики системи у набір категоріальних ознак. Узагальнено цей процес можна записати так:

$$s_t = D(x_t), \quad (2.10)$$

де s_t – дискретний стан системи на кроці t ;

D – функція дискретизації;

x_t – вектор поточних спостережуваних параметрів.

Після дискретизації стан може бути представлений як комбінація категоріальних ознак:

$$s_t = (u'_t, l'_t, \theta'_t), \quad (2.11)$$

де u'_t – дискретний клас використання CPU;

l'_t – дискретний клас системного навантаження;

θ'_t – дискретний клас throttling;

Параметри `cpu.weight`, `cpu.max` та `nice/renice` у цій реалізації не включаються до вектора стану. Вони розглядаються як параметри, на які впливає агент через простір керуючих дій. Такий підхід дозволяє уникнути надмірного зростання кількості можливих станів і зробити табличний Q-learning придатним для експериментального прототипу.

У реалізованій моделі утилізація процесорної потужності дискретизується на три чіткі класи: низька (*low*), середня (*medium*) та висока (*high*). Це дозволяє контролеру не працювати з неперервними відсотковими значеннями CPU, а приймати рішення на основі узагальненого класу навантаження.

Формально дискретизацію використання CPU можна подати так:

$$D_{u(u_t)} = low, \text{ якщо } 0 \leq u_t < \tau_{u1} \quad (2.12)$$

$$D_{u(u_t)} = medium, \text{ якщо } \tau_{u1} \leq u_t < \tau_{u2} \quad (2.13)$$

$$D_{u(u_t)} = high, \text{ якщо } u_t \geq \tau_{u2} \quad (2.14)$$

де D_u – функція дискретизації використання CPU;

u_t – значення використання CPU на кроці t ;

τ_{u1} та τ_{u2} – порогові значення, які відокремлюють низьке, середнє та високе навантаження;

low – низький рівень використання CPU;

medium – середній рівень використання CPU;

high – високий рівень використання CPU.

Системне навантаження також подається не як довільне числове значення, а як дискретний стан. У прототипі для цього використовуються категорії, що описують загальний характер завантаження системи: нормальний стан, зайнятий стан (*busy*) та перевантажений стан (*overloaded*). Така класифікація дозволяє відокремити ситуації, коли система працює в допустимому режимі, від ситуацій, коли потрібне активніше керуюче втручання.

Дискретизацію системного навантаження можна подати у вигляді:

$$D_{l(l_t)} = normal, \text{ якщо } l_t < \tau_{l1} \quad (2.15)$$

$$D_{l(l_t)} = busy, \text{ якщо } \tau_{l1} \leq l_t < \tau_{l2} \quad (2.16)$$

$$D_{l(l_t)} = overloaded, \text{ якщо } l_t \geq \tau_{l2} \quad (2.17)$$

де

D_l – функція дискретизації системного навантаження;

l_t – показник навантаження системи на кроці t ;

τ_{ul1} та τ_{ul2} – порогові значення для переходу між класами навантаження;

normal – нормальний стан системи;

busy – стан підвищеної зайнятості системи;

overloaded – стан перевантаження системи.

Окремо враховується стан *throttling*, тобто ситуація, коли використання CPU обмежується відповідними параметрами *cgroups*. Для цього показник θ_t перетворюється на дискретну ознаку. У найпростішому випадку використовуються класи *not_throttled* та *throttled*:

$$D_{\theta(\theta_t)} = not_throttled, \text{ якщо } \theta_t = 0, \quad (2.18)$$

$$D_{\theta(\theta_t)} = throttled, \text{ якщо } \theta_t > 0 \quad (2.19)$$

де D_θ – функція дискретизації стану throttling;

θ_t – показник наявності або інтенсивності throttling на кроці t ;

$not_{throttled}$ – стан, у якому обмеження CPU не спостерігається;

$throttled$ – стан, у якому наявне обмеження використання CPU.

Отже, у спрощеній моделі стан системи може набувати, наприклад, таких категоріальних описів, як *busy*, *overloaded* або *not_throttled*. Наприклад, стан виду:

$$s_t = (high, overloaded, not_{throttled}). \quad (2.20)$$

означає, що система має високе використання CPU, перебуває у стані перевантаження, але при цьому явний throttling не зафіксовано. Для агента такий стан є сигналом, що система працює під значним навантаженням, однак обмеження CPU ще не активувалися або не проявляються у поточному інтервалі спостереження.

Параметри *cpu.weight*, *cpu.max* та *nice/renice* також можуть бути подані у вигляді дискретних класів, оскільки контролер працює не з нескінченною кількістю можливих значень, а з обмеженим набором допустимих керуючих режимів. Це узгоджується з ідеєю табличного Q-learning, де кожній парі «стан – дія» відповідає окреме Q-значення.

Загальну функцію дискретизації можна подати як композицію окремих функцій:

$$D(x_t) = (D_{u(u_t)}, D_{l(l_t)}, D_{\{\theta\}(\theta_t)}), \quad (2.21)$$

де $D(x_t)$ – загальна функція перетворення вектора спостережень у дискретний стан; $D_{u(u_t)}$ – дискретизація використання CPU; $D_{l(l_t)}$ – дискретизація системного навантаження; $D_{\{\theta\}(\theta_t)}$ – дискретизація стану throttling.

Важливо, що стан системи в цій моделі не є повним внутрішнім станом ядра Linux. Агент працює лише з тими метриками, які доступні з простору користувача через стандартні інтерфейси. Тому модель стану має наближений характер. Вона відображає не всю складність роботи операційної системи, а лише ті характеристики, які є корисними для прийняття керуючих рішень у межах запропонованого user-space adaptive controller.

Такий підхід дозволяє зберегти баланс між формальною математичною постановкою задачі та практичною реалізованістю прототипу. З одного боку, стан системи має достатню структуру для застосування Q-learning. З іншого боку, модель залишається простою, інтерпретованою та придатною для реалізації без втручання у планувальник ядра Linux.

2.3 Простір керуючих дій

Після визначення множини станів системи необхідно формалізувати множину керуючих дій, які може виконувати адаптивний контролер. У межах цієї роботи дія агента не означає пряме втручання в алгоритм планування процесів ядра Linux. Контролер працює у просторі користувача та використовує лише стандартні механізми керування ресурсами, доступні в операційній системі.

Простір керуючих дій можна подати як скінченну множину:

$$A = \{a_0, a_1, a_2, \dots, a_m\}, \quad (2.22)$$

де A – множина доступних керуючих дій;

a_i – окрема дія агента;

m – кількість можливих дій без урахування нульового індексу.

У досліджуваній моделі кожна дія відповідає зміні одного з параметрів керування процесами або групами процесів. До таких параметрів належать *cpu.weight*, *cpu.max* та *nice/renice*. Окремо враховується дія *noop*, яка означає відсутність втручання з боку контролера на поточному кроці.

Узагальнений простір дій для запропонованого контролера можна записати так:

$$A = \left\{ no_{action}, increase_{weight}, decrease_{weight}, increase_{quota}, decrease_{quota}, increase_{nice}, decrease_{nice} \right\}, \quad (2.23)$$

де no_{action} – відсутність керуючого втручання;

$increase_{weight}$ – збільшення значення *cpu.weight*;

$decrease_{weight}$ – зменшення значення *cpu.weight*;

$increase_{quota}$ – збільшення доступної квоти CPU через *cpu.max*;

$decrease_{quota}$ – зменшення доступної квоти CPU через *cpu.max*;

$increase_{nice}$ – збільшення значення *nice*, що знижує пріоритет процесу;

$decrease_{nice}$ – зменшення значення *nice*, що підвищує пріоритет процесу.

Параметр *cpu.weight* у *cgroups v2* визначає відносну вагу групи процесів під час розподілу процесорного часу. Збільшення цього параметра означає, що відповідна група процесів може отримувати більшу частку CPU порівняно з іншими групами за умов конкуренції за ресурс. Зменшення *cpu.weight*, навпаки, знижує відносну пріоритетність групи під час розподілу процесорного часу.

Параметр *cpu.max* використовується для обмеження максимально доступної частки CPU для певної групи процесів. Збільшення квоти CPU може бути доцільним тоді, коли система не перебуває у стані критичного перевантаження, а процес або група процесів потребує більшої обчислювальної потужності. Зменшення квоти може застосовуватися для

стримування надмірного споживання CPU та запобігання погіршенню загального стану системи.

Механізм `nice/renice` дає змогу змінювати пріоритет окремого процесу на рівні простору користувача. У Linux більше значення `nice` відповідає нижчому пріоритету процесу, а менше значення `nice` – вищому пріоритету. Тому дія `increase_nice` фактично зменшує пріоритет процесу, а дія `decrease_nice` підвищує його відносну пріоритетність.

На кожному кроці часу t контролер обирає одну дію з множини доступних дій:

$$a_t \in A, \quad (2.24)$$

де a_t – дія, обрана агентом на кроці t ;

– множина всіх допустимих дій.

У випадку детермінованої політики вибір дії можна подати так:

$$a_t = \pi(s_t), \quad (2.25)$$

де:

π – політика керування;

s_t – поточний дискретний стан системи;

a_t – дія, яку обирає агент у цьому стані.

У випадку стохастичної політики дія може вибиратися з певною ймовірністю:

$$a_t \pi(a \vee s_t) \quad (2.26)$$

де $\pi(a \vee s_t)$ – ймовірність вибору дії a у стані s_t ;

a_t – дія, фактично виконана на кроці t .

Для табличного Q-learning кожній парі «стан – дія» ставиться у відповідність числове Q-значення. Воно показує очікувану корисність виконання певної дії в певному стані системи:

$$Q = Q(s, a), s \in S, a \in A, \quad (2.27)$$

де $Q(s, a)$ – оцінка корисності виконання дії a у стані s ;

S – множина можливих станів системи;

A – множина доступних дій.

У практичній реалізації контролер не повинен змінювати параметри системи надто часто, оскільки надмірні керуючі втручання можуть погіршити стабільність роботи середовища. Тому дія поор є важливою частиною простору дій. Вона дозволяє агенту не змінювати параметри керування, якщо поточний стан системи є прийнятним або якщо додаткове втручання не дає очікуваної користі.

Таким чином, простір керуючих дій у запропонованій моделі є скінченним, дискретним і сумісним із табличним Q-learning. Він охоплює базові механізми впливу на процеси Linux з простору користувача, але не передбачає модифікації планувальника ядра операційної системи. Це відповідає загальній концепції роботи, у якій адаптивне керування реалізується як зовнішній user-space controller, що приймає рішення на основі спостережуваних метрик системи.

2.4 Функція винагороди

Функція винагороди є одним із ключових елементів математичної моделі навчання з підкріпленням. Вона визначає, наскільки корисною або

небажаною була дія агента з погляду подальшого стану системи. У межах цієї роботи винагорода використовується для того, щоб адаптивний контролер поступово формував поведінку, спрямовану на підтримання стабільного використання CPU, уникнення перевантаження, зменшення небажаного throttling та збереження прийняттого рівня системного навантаження.

У запропонованій моделі винагорода на кроці (t) залежить від стану системи до виконання дії, обраної керуючої дії та стану системи після її виконання:

$$r_t = R(s_t, a_t, s_{t+1}) \quad (2.28)$$

де r_t – винагорода, отримана агентом на кроці (t);

R – функція винагороди;

s_t – стан системи до виконання дії;

a_t – дія, обрана контролером;

s_{t+1} – стан системи після виконання дії.

У практичній реалізації адаптивного контролера винагорода формується як сума позитивного бонусу за стабільне використання CPU та штрафів за небажані стани системи. Узагальнено її можна подати так:

$$r_t = B_{\{cpu\}(t)} - C_{\{overload\}(t)} - C_{\{idle\}(t)} - C_{\{load\}(t)} - C_{\{throttle\}(t)}, \quad (2.29)$$

де $B_{\{cpu\}(t)}$ – бонус за перебування утилізації CPU у стабільному робочому діапазоні; $C_{\{overload\}(t)}$ – штраф за перевищення верхнього порогу використання CPU;

$C_{\{idle\}(t)}$ – штраф за надто низьке використання CPU;

$C_{\{load\}(t)}$ – штраф за підвищене системне навантаження;

$C_{\{throttle\}(t)}$ – штраф за збільшення часу throttling.

У програмній реалізації стабільним вважається діапазон використання CPU від 35 % до 75 %. Якщо поточне значення CPU перебуває в цьому інтервалі, агент отримує додатний бонус:

$$B_{\{cpu\}(t)} = 0.30 \cdot I(35 \leq u_t \leq 75) \quad (2.30)$$

де $B_{\{cpu\}(t)}$ – бонус за стабільне використання CPU;

u_t – відсоток використання CPU на кроці (t);

$I(\cdot)$ – індикаторна функція, яка дорівнює 1, якщо умова істинна, і 0, якщо умова хибна.

Штраф за перевантаження CPU активується тоді, коли використання процесорного ресурсу перевищує 75 %. Для цього використовується функція максимуму, яка не допускає появи штрафу за відсутності перевантаження:

$$C_{\{overload\}(t)} = \frac{\max(0, u_t - 75)}{25} \quad (2.31)$$

де $C_{\{overload\}(t)}$ – штраф за перевищення допустимого рівня використання CPU;

u_t – поточне використання CPU у відсотках;

75 – верхній поріг стабільного діапазону CPU;

25 – нормувальний коефіцієнт.

Штраф за надто низьке використання CPU враховує ситуації, коли система працює недостатньо активно. У реалізованій моделі нижнім порогом для такого штрафу є 20 %:

$$\frac{C_{\{idle\}(t)} = \max(0, 20 - u_t)}{100}, \quad (2.32)$$

де $C_{\{idle\}(t)}$ – штраф за надто низьке використання CPU;

u_t – поточне використання CPU у відсотках;

20 – нижній поріг, нижче якого система вважається недостатньо завантаженою;

100 – нормувальний коефіцієнт.

Окремо враховується штраф за підвищене системне навантаження. У програмній реалізації для цього використовується показник load average за одну хвилину. Якщо його значення перевищує 2, агент отримує штраф:

$$C_{\{load\}(t)} = \frac{\max(0, l_t - 2)}{4} \quad (2.33)$$

де $C_{\{load\}(t)}$ – штраф за підвищене системне навантаження;

l_t – значення load average за одну хвилину на кроці t ;

2 – порогове значення, після якого навантаження починає штрафуватися;

4 – нормувальний коефіцієнт.

Штраф за throttling враховує збільшення сумарного часу обмеження CPU. Якщо значення лічильника throttled_usec після виконання дії стало більшим, ніж до її виконання, це означає, що в поточному інтервалі спостереження було зафіксовано throttling:

$$C_{\{throttle\}(t)} = I(\theta_t > \theta_{t-1}) \quad (2.34)$$

де $C_{\{throttle\}(t)}$ – штраф за throttling;

θ_t – значення лічильника throttled_usec після виконання дії;

θ_{t-1} – значення цього лічильника до виконання дії;

$I(\cdot)$ – індикаторна функція.

З урахуванням наведених компонентів фактичну функцію винагороди, використану в експериментальному прототипі, можна записати так:

$$r_t = 0.30 \cdot I(35 \leq u_t \leq 75) - \frac{\max(0, u_t - 75)}{25} - \frac{\max(0, 20 - u_t)}{100} - \frac{\max(0, l_t - 2)}{4} - I(\theta_t > \theta_{t-1}) \quad (2.35)$$

де r_t – підсумкова винагорода на кроці (t);

u_t – поточне використання CPU у відсотках;

l_t – значення load average за одну хвилину;

θ_t і θ_{t-1} – значення лічильника throttled_usec після та до виконання дії відповідно.

У більш загальному вигляді цю функцію можна подати із ваговими коефіцієнтами:

$$r_t = \alpha \cdot I(35 \leq u_t \leq 75) - \beta \cdot \frac{\max(0, u_t - 75)}{25} - \delta \cdot \frac{\max(0, 20 - u_t)}{100} - \mu \cdot \frac{\max(0, l_t - 2)}{4} - \nu \cdot I(\theta_t > \theta_{t-1}). \quad (2.36)$$

Ці коефіцієнти є гіперпараметрами ШІ-агента та визначаються емпіричним або експертним шляхом. Вони задають компроміс між продуктивністю операційної системи, стабільністю її роботи, допустимим рівнем навантаження та небажаністю throttling. У реалізованому прототипі фактично використано значення .

Важливо зазначити, що в цій реалізації не використовується окрема цільова утилізація $u_{\{target\}}$. Замість цього стабільна зона CPU задається інтервалом від 35 % до 75 %, а відхилення за межі цього інтервалу штрафуються за допомогою функції $\max(0, \cdot)$. Такий підхід усуває проблему неконтрольованого зростання від’ємного внеску, оскільки штрафи активуються лише тоді, коли відповідний поріг справді порушено. Якщо $u_t > 75$, збільшується штраф $C_{\{overload\}}(t)$; якщо $u_t < 20$, збільшується штраф

$C_{\{idle\}}(t)$; якщо $20u_t < 35$, штраф за простій уже не застосовується, але бонус за стабільну зону ще не нараховується.

Дія (noop) у просторі керуючих дій означає відсутність зміни параметрів системи на поточному кроці. У поточній функції винагороди вона не має окремого штрафу або бонусу. Її доцільність оцінюється опосередковано через подальший стан системи: якщо після вибору (noop) система залишається у стабільному режимі, агент отримує вищу винагороду; якщо стан погіршується, винагорода зменшується через відповідні штрафні компоненти.

Отже, функція винагороди в цій роботі не є повною моделлю продуктивності операційної системи Linux. Вона є практичним критерієм для навчання табличного Q-learning агента в межах експериментального user-space adaptive controller. Її компоненти побудовані на метриках, доступних із простору користувача, і узгоджуються з ідеєю керування процесами без модифікації планувальника ядра Linux.

2.5 Алгоритм Q-learning для адаптивного вибору дій

Для адаптивного вибору керуючих дій у роботі використовується табличний алгоритм Q-learning. Цей метод належить до підходів навчання з підкріпленням без явного моделювання середовища, тобто до model-free методів. Його застосування є доцільним у межах цієї задачі, оскільки аналітичний вигляд функції переходів між станами операційної системи Linux заздалегідь невідомий.

Основна ідея Q-learning полягає в тому, що агент поступово оцінює корисність виконання кожної доступної дії в кожному дискретному стані системи. Для цього використовується таблиця Q-значень, або Q-table. Кожному стану $s \in S$ та кожній дії $a \in A$ ставиться у відповідність числове значення $Q(s, a)$, яке характеризує очікувану корисність вибору дії a у стані s .

Формально Q-функцію можна подати як відображення:

$$Q: S \times A \rightarrow R, \quad (2.37)$$

де Q – функція корисності стану та дії;

S – множина дискретних станів системи;

A – множина доступних керуючих дій;

R – множина дійсних чисел, якими оцінюється корисність дій.

На початку навчання Q-значення є невідомими, тому в табличній реалізації вони ініціалізуються нульовими значеннями:

$$Q_{0(s,a)} = 0, s \in S, a \in A, \quad (2.38)$$

де $Q_{0(s,a)}$ – початкове значення Q-функції для стану s та дії a ;

s – дискретний стан системи;

a – керуюча дія.

У реалізованому прототипі стан системи формується з трьох компонентів: дискретного рівня використання CPU, дискретного рівня системного навантаження та дискретного стану throttling:

$$s_t = (u'_t, l'_t, \theta'_t), \quad (2.39)$$

де s_t – дискретний стан системи на кроці t ;

u'_t – дискретний клас використання CPU;

l'_t – дискретний клас системного навантаження;

θ'_t – дискретний клас стану throttling.

Наприклад, стан $s_t = (\text{high}, \text{overloaded}, \text{not_throttled})$ означає, що система має високий рівень використання CPU, перебуває у стані перевантаження, але в поточному інтервалі спостереження throttling не

зафіксовано. Такий запис узгоджується з моделлю стану, наведеною у підрозділі 2.2.

На кожному кроці часу агент має обрати одну дію з множини доступних дій. У роботі використовується ε -жадібна стратегія, яка поєднує використання вже накопичених знань і дослідження альтернативних дій. Спочатку визначається дія з найбільшим поточним Q-значенням:

$$a^{(s)} = \operatorname{argmax} Q(s, a'), a' \in A, \quad (2.40)$$

де $a^{(s)}$ – дія, яка має найбільше Q-значення у стані s ;

a' – довільна дія з множини A ;

$Q(s, a')$ – оцінка корисності виконання дії a' у стані s .

Ймовірність вибору дії за ε -жадібною політикою можна подати так:

$$\pi(a \vee s) = \varepsilon / |A| + (1 - \varepsilon) \cdot I, \quad (2.41)$$

де $\pi(a \vee s)$ – ймовірність вибору дії a у стані s ;

ε – ймовірність дослідження випадкової дії;

$|A|$ – кількість доступних дій агента;

$I(\cdot)$ – індикаторна функція, що дорівнює 1, якщо умова істинна, і 0,

якщо умова хибна;

$a^{(s)}$ – дія з найбільшим Q-значенням у стані s .

У межах цієї роботи кількість доступних дій дорівнює семи, тобто $|A| = 7$. Отже, найкраща за поточними Q-значеннями дія отримує ймовірність $1 - \varepsilon + \varepsilon/7$, а кожна інша дія – ймовірність $\varepsilon/7$. Така стратегія дозволяє агенту не лише використовувати вже знайдені корисні дії, а й періодично перевіряти альтернативні варіанти поведінки.

Після виконання дії a_t контролер отримує новий стан $s_{\{t+1\}}$ та винагороду r_t . На основі цієї інформації оновлюється відповідне Q-значення.

Щоб не змішувати швидкість навчання з ваговими коефіцієнтами функції винагороди, у математичному описі швидкість навчання позначимо через η . У програмному коді цьому параметру відповідає змінна alpha.

Формула оновлення Q-значення має вигляд:

$$Q_{(t+1)(s_t, a_t)} = Q_{t(s_t, a_t)} + \eta \cdot \left[r_t + \gamma \cdot \max_{a' \in A} Q_{t(s_{t+1}, a')} - Q_{t(s_t, a_t)} \right], \quad (2.42)$$

де $Q_{(t+1)(s_t, a_t)}$ – оновлене Q-значення для поточного стану та дії;

$Q_{t(s_t, a_t)}$ – попереднє Q-значення;

η – швидкість навчання;

r_t – винагорода, отримана після виконання дії;

γ – коефіцієнт дисконтування майбутніх винагород;

$s_{\{t+1\}}$ – наступний стан системи;

$\max_{a' \in A} Q_{t(s_{t+1}, a')}$ – максимальне очікуване значення функції якості серед

усіх можливих дій a' у наступному стані середовища.

У програмній реалізації використано такі значення основних гіперпараметрів Q-learning:

$$\eta = 0.20, \gamma = 0.85, \varepsilon = 0.25, |A| = 7 \quad (2.43)$$

де η – швидкість навчання;

γ – коефіцієнт дисконтування;

ε – імовірність випадкового вибору дії для дослідження середовища;

$|A|$ – кількість доступних дій агента.

Значення $\eta = 0.20$ означає, що новий досвід частково коригує вже накопичене Q-значення, але не замінює його повністю. Значення $\gamma = 0.85$ показує, що агент враховує не лише негайну винагороду, а й очікувану користь майбутніх станів. Значення $\varepsilon = 0.25$ забезпечує періодичне

дослідження альтернативних дій, що важливо для навчання агента в умовах неповної інформації про середовище.

У реалізованому адаптивному контролері цикл навчання має такий загальний вигляд. Спочатку контролер збирає системні метрики та перетворює їх у дискретний стан s_t . Потім агент обирає дію a_t відповідно до ε -жадібної політики. Далі дія застосовується через відповідний backend, який може працювати у режимі симуляції або взаємодіяти зі стандартними механізмами Linux. Після цього контролер знову збирає метрики, формує наступний стан s_{t+1} , обчислює винагороду r_t та оновлює Q-table.

Важливо, що навчання відбувається не шляхом модифікації планувальника ядра Linux, а через зовнішній user-space controller. Агент не змінює внутрішню логіку kernel scheduler, а лише навчається обирати одну з допустимих дій: `noop`, `increase_weight`, `decrease_weight`, `increase_quota`, `decrease_quota`, `increase_nice` або `decrease_nice`. Ці дії реалізуються через доступні механізми керування ресурсами, зокрема `cgroups v2`, `cpu.weight`, `cpu.max` та `nice/renice`.

Результатом навчання є Q-table, у якій зберігаються оцінки корисності для пар «дискретний стан – дія». У програмній реалізації ця таблиця може бути експортована та збережена у локальний файл конфігурації формату JSON, а саме `q_table.json`. Наявність такої таблиці дозволяє аналізувати, які дії агент вважає доцільними для різних станів системи, а також використовувати накопичені значення під час наступних запусків експериментального контролера.

Таким чином, Q-learning у цій роботі виконує роль базового ШІ-механізму адаптивного прийняття рішень. Він дозволяє контролеру не використовувати наперед жорстко задані правила для всіх ситуацій, а поступово накопичувати досвід взаємодії із середовищем і коригувати вибір дій відповідно до отриманої винагороди.

2.6 Обмеження запропонованої математичної моделі

Запропонована математична модель адаптивного керування процесами має експериментальний характер і призначена для побудови контролера простору користувача. Вона дає змогу формалізувати задачу керування процесами Linux у вигляді марковського процесу прийняття рішень та застосувати табличний Q-learning для вибору керуючих дій. Водночас під час інтерпретації результатів необхідно враховувати низку методологічних і технічних обмежень цієї моделі.

Насамперед модель використовує спрощений опис стану системи. У реалізованому прототипі стан формується на основі трьох основних компонентів: рівня використання CPU, системного навантаження та стану throttling. Такий вибір дає змогу обмежити розмір простору станів і зробити табличний Q-learning придатним для практичної реалізації. Водночас модель не охоплює всіх характеристик, які можуть впливати на поведінку операційної системи Linux. Зокрема, у стані явно не враховуються обсяг використаної оперативної пам'яті, інтенсивність операцій введення-виведення, мережева активність, кількість потоків, особливості конкретних застосунків та інші системні параметри.

Наступний методологічний компроміс пов'язаний із дискретизацією неперервних метрик. Значення використання CPU, load average та показників throttling перетворюються на скінченну кількість категорій, наприклад, "low", "medium", "high", "busy", "overloaded", "not_throttled". Такий підхід спрощує навчання агента, але водночас призводить до втрати частини інформації. Два відмінні чисельні показники можуть потрапити до одного дискретного класу, хоча з погляду реальної поведінки системи вони можуть мати різні наслідки.

Окрему увагу слід звернути на обмеження, пов'язані з використанням саме табличного Q-learning. Цей алгоритм є простим, інтерпретованим і

придатним для невеликого дискретного простору станів і дій. Проте він погано масштабується у випадку значного збільшення кількості станів. Якщо до моделі додати більше системних метрик або деталізувати існуючі класи, розмір Q-таблиці може швидко зрости, що ускладнить навчання та подальше використання агента.

Архітектурною особливістю моделі, що одночасно виступає її обмеженням, є відсутність явної функції переходів середовища. У роботі використовується model-free підхід, тому агент не будує аналітичну модель переходів між станами. Зазначене рішення є науково обґрунтованим для стохастичного середовища ОС Linux, оскільки поведінка системи залежить від багатьох чинників: активності процесів, перемикання контексту, інтенсивності запитів, особливостей планувальника ядра та поточного навантаження. Водночас відсутність явної функції переходів означає, що агент навчається лише на основі спостережуваного досвіду і не має повного прогнозного опису середовища.

Крім того, слід зважити на чутливість функції винагороди до вибраних порогових значень і гіперпараметрів. У моделі використовуються фіксовані межі стабільного діапазону CPU, порогові значення для load average та штраф за зростання throttling. Такі параметри задають бажану поведінку агента, але можуть потребувати переналаштування для інших апаратних конфігурацій, типів навантаження або сценаріїв експлуатації системи.

Важливим технічним обмеженням є те, що запропонований контролер не є заміною планувальника ядра Linux. Він працює у просторі користувача та використовує стандартні механізми керування ресурсами, зокрема cgroups v2, cpu.weight, cpu.max та nice/renice. Тому вплив агента на систему є опосередкованим. Контролер може змінювати доступні параметри керування, однак не змінює внутрішню логіку kernel scheduler і не має повного доступу до внутрішнього стану ядра.

Результати навчання також залежать від конкретного сценарію експерименту. Матриця Q-значень формується на основі тих станів, дій і винагород, які агент спостерігає під час певного запуску. Якщо характер навантаження зміниться, раніше отримані Q-значення можуть бути не повністю оптимальними. Тому результати навчання слід розглядати як адаптацію до заданого експериментального середовища, а не як універсальну політику керування для всіх Linux-систем.

У межах цієї роботи не використовуються глибокі нейронні мережі або складні методи апроксимації функції якості. Це зроблено свідомо, оскільки метою є побудова інтерпретованого експериментального прототипу адаптивного керування, а не розробка складної системи промислового рівня (production-ready environment). Табличний Q-learning дозволяє прозоро показати зв'язок між станами, діями, винагородами та результатом навчання.

Таким чином, запропонована математична модель є спрощеною, але достатньою для дослідження базового принципу адаптивного керування процесами Linux із використанням методів штучного інтелекту. Вона забезпечує формальну основу для реалізації контролера простору користувача, однак не претендує на побудову всеохопної аналітичної моделі низькорівневих підсистем операційної системи або на універсальну оптимальність у довільних умовах експлуатації.

Висновки до розділу 2

У другому розділі було побудовано математичну модель адаптивного керування процесами в операційній системі Linux. Задачу керування формалізовано як марковський процес прийняття рішень, у межах якого середовищем виступає ОС Linux або її спрощена симуляційна модель,

агентом – контролер простору користувача, станом – дискретизований опис поточного навантаження, а діями – допустимі зміни параметрів керування ресурсами.

Було визначено модель стану системи, що ґрунтується на трьох основних компонентах: рівні використання CPU, системному навантаженні та стані throttling. Такий підхід є математично обґрунтованим, оскільки дозволяє мінімізувати розмірність простору станів, запобігаючи проблемі вибуху простору станів (state-space explosion), і водночас врахувати ключові метрики, які характеризують поточне навантаження системи.

У розділі також формалізовано простір керуючих дій. Показано, що адаптивний контролер не втручається в роботу планувальника ядра Linux, а використовує стандартні механізми операційної системи: `cgroups v2`, `cpu.weight`, `cpu.max` та `nice/renice`. До простору дій включено зміну ваги CPU, зміну квоти CPU, зміну пріоритету процесів, а також нульову дію `noop`.

Окремо було формалізовано функцію винагороди, яку спроектовано у вигляді системи нелінійного компромісу з використанням індикаторних функцій та оператора $\max(0, \cdot)$ для коректного опрацювання крайових умов утилізації CPU та системного навантаження. Винагорода враховує стабільний діапазон використання CPU, штрафи за перевантаження, надто низьке використання CPU, підвищене системне навантаження та збільшення часу throttling. Така функція не є повною моделлю продуктивності Linux, однак вона задає практичний критерій для навчання агента в межах експериментального контролера.

Для адаптивного вибору дій обґрунтовано використання табличного алгоритму Q-learning. Оскільки аналітичний вигляд функції переходів середовища є невідомим, застосування model-free підходу є доцільним для стохастичного середовища ОС Linux. Результатом навчання є Q-таблиця, у якій накопичуються оцінки корисності для пар «стан – дія», що забезпечує

високу інтерпретованість, математичну прозорість та можливість явного аналізу сформованої політики адаптивного керування.

Також було визначено основні обмеження запропонованої математичної моделі. Вони пов'язані зі спрощеним описом стану, дискретизацією неперервних метрик, залежністю функції винагороди від порогових значень і гіперпараметрів, а також із тим, що контролер працює у просторі користувача та не замінює планувальник ядра Linux.

Отже, у цьому розділі сформовано строгу математичну основу для подальшої програмної реалізації адаптивного контролера та побудовано аналітичний апарат, експериментальна валідація та аналіз результатів якого детально розглядаються у розділі 3.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ ЕКСПЕРИМЕНТАЛЬНИХ РЕЗУЛЬТАТІВ

3.1 Загальна архітектура програмного прототипу

У межах дипломної роботи було розроблено програмний прототип контролера для адаптивного керування процесорними ресурсами в операційній системі Linux. Запропонована система не є власним планувальником ядра Linux і не замінює стандартні механізми планування процесів. Її призначення полягає у демонстрації підходу до адаптивного керування на рівні простору користувача з використанням методів навчання з підкріпленням.

Загальна логіка роботи програмного прототипу побудована як послідовний контур адаптивного керування. Він включає збір системних метрик, перетворення отриманих значень у дискретний стан, вибір керуючої дії інтелектуальним агентом, виконання цієї дії через відповідний програмний механізм та збереження результатів експерименту для подальшого аналізу. Така структура дозволяє відокремити етапи моніторингу, прийняття рішень, взаємодії з операційною системою та аналізу результатів.

Після опису загальної логіки доцільно навести схему адаптивного контуру керування, яка відображає взаємозв'язок основних компонентів системи.

Першим компонентом системи є модуль збору системних метрик, реалізований у файлі *monitoring/metrics.py*. Він отримує поточні характеристики стану системи, зокрема рівень використання процесора, значення усередненого системного навантаження, кількість активних процесів та, за наявності шляху до ресурсної групи, показник *throttled_usec* із

файлу *cpu.stat*. Зібрані метрики описують поточне навантаження системи і надалі використовуються для формування стану середовища.

Другим компонентом є модуль дискретизації стану, реалізований у файлі *environment/state.py*. Оскільки у прототипі використовується табличне Q-навчання, неперервні числові метрики не передаються агенту безпосередньо. Замість цього вони перетворюються на дискретні категорії. Наприклад, рівень використання процесора може бути віднесений до категорій «low», «medium» або «high», рівень навантаження – до «normal», «busy» або «overloaded», а стан обмеження ресурсів – до «throttled» або «not_throttled». У результаті формується текстовий ключ стану, наприклад «medium|busy|not_throttled».

Компонент штучного інтелекту реалізовано у модулі *agent/q_learning.py*. У цьому файлі міститься клас *QLearningAgent*, який відповідає за вибір керуючої дії та оновлення Q-таблиці. Агент використовує табличне Q-навчання та ϵ -жадібну стратегію вибору дії. Це означає, що частина рішень приймається випадково з метою дослідження середовища, а частина – на основі вже накопичених Q-значень. Саме цей модуль є основним інтелектуальним компонентом роботи, оскільки він реалізує механізм адаптивного прийняття рішень на основі досвіду попередніх взаємодій із середовищем.

Центральним координуючим елементом системи є модуль *controller/adaptive_controller.py*. Він містить клас *AdaptiveController*, що реалізує основний цикл керування. На кожній ітерації контролер збирає метрики, формує дискретний стан, передає його агенту, отримує обрану дію, передає її відповідному модулю застосування, після чого повторно збирає метрики, обчислює винагороду та оновлює Q-таблицю. Таким чином, контролер поєднує між собою моніторинг, модель середовища, агента навчання, механізми впливу на систему та журналювання результатів.

Застосування керуючих дій винесено в окремий модуль *controller/backends.py*. Такий підхід дозволяє розділити логіку прийняття рішень і конкретний механізм впливу на операційну систему. У проєкті реалізовано три режими роботи. Режим симуляції «simulation» є безпечним демонстраційним режимом: він змінює лише внутрішні значення змодельованих параметрів і не впливає на операційну систему. Режим «nice» використовує механізм *nice/renice* для зміни пріоритету конкретного процесу за його ідентифікатором процесу (PID), однак така дія залежить від прав доступу користувача. Режим реального керування «real» призначений для середовищ із підтримкою *cgroups v2* і може записувати значення у файли *cru.weight* та *cru.max*, якщо заданий шлях до ресурсної групи існує та доступний для запису.

Додатково в системі реалізовано модулі для збереження й аналізу експериментальних результатів. Модуль *experiments/exporter.py* формує журнал експерименту у форматах CSV та XLSX, а також створює текстовий файл *summary.txt* із коротким підсумком запуску. Модуль *experiments/plots.py* будує графіки динаміки використання процесора, зміни винагороди та розподілу обраних дій. Після завершення експерименту Q-таблиця агента зберігається у файл *q_table.json*, що дозволяє надалі аналізувати сформовану поведінку агента або продовжити навчання з уже накопичених значень.

Для узагальнення функцій основних програмних компонентів у таблиці 3.1 наведено структуру модулів проєкту.

Таблиця 3.1 – Основні модулі програмного прототипу

Модуль	Призначення
<i>main.py</i>	Головна точка запуску експерименту, створення основних об'єктів системи та запуск циклу керування.
<i>agent/q_learning.py</i>	Реалізація Q-learning агента, вибір дій, оновлення та збереження Q-таблиці.
<i>controller/adaptive_controller.py</i>	Основний цикл адаптивного керування та координація взаємодії між компонентами.
<i>controller/backends.py</i>	Реалізація механізмів застосування керуючих дій у режимах симуляції, «nice» та реального керування через <i>cgroups v2</i> .
<i>monitoring/metrics.py</i>	Збір системних метрик, необхідних для оцінювання поточного стану системи.
<i>environment/state.py</i>	Дискретизація числових метрик і формування стану для Q-learning агента.
<i>environment/reward.py</i>	Обчислення винагороди за результатами виконання керуючої дії.
<i>experiments/exporter.py</i>	Експорт результатів експерименту у файли CSV, XLSX та <i>summary.txt</i> .
<i>experiments/plots.py</i>	Побудова графіків для аналізу динаміки експерименту.
<i>load_generator.py</i>	Генерація штучного CPU-навантаження для перевірки роботи контролера.

Отже, архітектура програмного прототипу побудована за модульним принципом. Збір метрик, формування стану, прийняття рішень, застосування дій і збереження результатів реалізовано як окремі компоненти. Це спрощує аналіз системи, дозволяє запускати її в безпечному режимі симуляції, а також забезпечує можливість експериментальної перевірки специфічних механізмів Linux у середовищах із підтримкою *cgroups v2*. Водночас система залишається експериментальним прототипом простору користувача і не

претендує на заміну стандартних механізмів планування процесів у ядрі Linux.

3.2 Реалізація збору системних метрик

Збір системних метрик у програмному прототипі реалізовано в модулі *monitoring/metrics.py*. Основним класом цього модуля є *MetricsCollector*, який формує поточний знімок стану системи у вигляді об'єкта *SystemMetrics*. Такий підхід дозволяє відокремити етап моніторингу від логіки прийняття рішень і забезпечити єдиний формат даних для подальшої дискретизації стану.

Об'єкт *SystemMetrics* містить набір характеристик, які описують поточний стан системи з погляду використання процесорних ресурсів. До них належать час вимірювання, відсоток використання процесора, усереднене системне навантаження за одну хвилину, кількість процесів, значення *throttled_usec* та назва платформи. Не всі ці показники безпосередньо входять до дискретного стану агента, однак вони є корисними для журналювання, аналізу експерименту та перевірки поведінки системи.

Для отримання частини системних характеристик використовується бібліотека *psutil*. З її допомогою програма визначає відсоток використання процесора та кількість процесів у системі. Показник використання процесора є одним із ключових параметрів, оскільки саме він безпосередньо відображає поточне завантаження обчислювальних ресурсів. У межах прототипу цей показник надалі перетворюється на дискретний рівень «low», «medium» або «high», що дозволяє використовувати його в табличному алгоритмі Q-навчання.

Другим важливим показником є усереднене системне навантаження за одну хвилину. У програмній реалізації воно отримується за допомогою функції *os.getloadavg()*, якщо така функція доступна в операційній системі. Значення *load average* використовується як узагальнений індикатор навантаження системи. У модулі дискретизації цей показник класифікується як «normal», «busy» або «overloaded». Така категоризація дозволяє агенту враховувати не лише миттєве використання процесора, а й загальний рівень системного навантаження.

Кількість процесів у системі визначається за допомогою функції *psutil.pids()*. У поточній реалізації цей показник зберігається в журналі експерименту та використовується для подальшого аналізу результатів. Він не входить безпосередньо до ключа дискретного стану інтелектуального агента, однак є корисним діагностичним параметром, оскільки дозволяє оцінити загальний контекст запуску експерименту.

Окремо в модулі передбачено зчитування показника *throttled_usec* із віртуальної файлової системи *cgroups v2*. Якщо під час запуску програми передано параметр командного рядка *--cgroup-path*, контролер намагається прочитати файл *cpu.stat* у відповідній директорії ресурсної групи. У цьому файлі шукається значення *throttled_usec*, яке відображає сумарний час обмеження процесорного ресурсу для відповідної групи. Якщо шлях до ресурсної групи не передано, файл відсутній або значення неможливо прочитати, програма повертає нульове значення. Це зроблено для того, щоб прототип міг працювати не лише в середовищах із підтримкою *cgroups v2*, а й у безпечному режимі симуляції.

Показник *throttled_usec* використовується для визначення того, чи зросло обмеження процесорного часу між двома послідовними вимірюваннями. У модулі *environment/state.py* він перетворюється на одну з двох категорій: «throttled» або «not_throttled». Якщо поточне значення *throttled_usec* більше за попереднє, система вважає, що відбулося примусове

обмеження процесорного часу. Такий підхід дозволяє включити до стану середовища не лише рівень завантаження, а й ознаку втручання з боку механізмів квотування ядра.

Для узагальнення ролі кожної метрики в роботі прототипу доцільно навести таблицю 3.2.

Таблиця 3.2 – Системні метрики, що використовуються у програмному прототипі

Метрика	Джерело отримання	Призначення
<i>cpu_percent</i>	<i>psutil.cpu_percent()</i>	Визначення поточного рівня використання процесора та формування категорій «low», «medium», «high».
<i>load_avg_1m</i>	<i>os.getloadavg()</i>	Оцінювання усередненого системного навантаження та формування категорій «normal», «busy», «overloaded».
<i>process_count</i>	<i>psutil.pids()</i>	Діагностичний показник для журналювання й аналізу умов проведення експерименту.
<i>throttled_usec</i>	файл <i>cpu.stat</i> у <i>cgroups v2</i>	Визначення наявності обмеження процесорного часу та формування категорій «throttled» або «not_throttled».
<i>timestamp</i>	системний час	Фіксація моменту отримання знімка стану системи.
<i>platform_name</i>	інформація про платформу	Додаткова характеристика середовища запуску.

Зібрані метрики використовуються у двох основних частинах системи. По-перше, вони передаються до функції *discretize_metrics()*, де формують поточний стан для агента Q-навчання. У цій функції відсоток використання процесора, значення *load average* та зміна показника *throttled_usec* перетворюються на дискретні категорії. По-друге, метрики використовуються у функції *calculate_reward()* для оцінювання наслідків

виконаної керуючої дії. Винагорода залежить від поточного рівня використання процесора, показника *load average* і зміни статусу обмеження процесорного часу.

Таким чином, модуль збору системних метрик є початковою ланкою всього адаптивного контуру керування. Саме на основі отриманих ним даних формується стан середовища, який передається інтелектуальному агенту для вибору дії. Реалізований механізм моніторингу працює у просторі користувача і не потребує модифікації ядра Linux. У режимі симуляції він використовується для перевірки логіки адаптивного циклу без зміни системних параметрів, а в режимі реального керування може застосовуватися для аналізу стану ресурсних груп *cgroups v2* за умови наявності відповідних прав доступу.

3.3 Реалізація середовища та агента навчання

Реалізація середовища та агента навчання є центральною частиною програмного прототипу, оскільки саме в цих компонентах формується зв'язок між поточним станом операційної системи, керуючими діями та механізмом навчання з підкріпленням. У межах розробленої системи середовище не є окремим симулятором у класичному розумінні, а формується на основі фактичних або змодельованих системних метрик, які надходять із модуля збору даних. Агент, у свою чергу, аналізує дискретний стан середовища, обирає дію та оновлює Q-таблицю на основі отриманої винагороди.

Формування стану середовища реалізовано в модулі *environment/state.py*. У цьому файлі визначено структуру *DiscreteState*, яка містить три основні компоненти: рівень використання процесора, рівень

системного навантаження та ознаку наявності обмеження процесорного часу. Для перетворення числових метрик у дискретний стан використовується функція *discretize_metrics()*. Вона приймає поточний об'єкт *SystemMetrics* і попереднє значення *throttled_usec*, після чого формує категоріальний опис стану системи.

У програмній реалізації рівень використання процесора поділяється на три категорії. Якщо значення CPU менше ніж 35 %, стан позначається як «low». Якщо воно перебуває в межах від 35 % до 75 %, використовується категорія «medium». Якщо рівень CPU дорівнює або перевищує 75 %, стан класифікується як «high». Такий поділ дозволяє спростити неперервний простір значень і зробити його придатним для табличного Q-навчання.

Аналогічно обробляється показник усередненого системного навантаження за одну хвилину. Якщо значення *load_avg_1m* менше ніж 1,0, стан навантаження визначається як «normal». Якщо значення перебуває в діапазоні від 1,0 до 3,0, використовується категорія «busy». Якщо показник дорівнює або перевищує 3,0, система класифікує навантаження як «overloaded». Це дає змогу агенту враховувати не тільки миттєве використання процесора, а й загальний тренд завантаженості системи.

Третім компонентом дискретного стану є ознака обмеження процесорного часу. Вона визначається шляхом порівняння поточного значення *throttled_usec* із попереднім. Якщо поточне значення більше за попереднє, стан позначається як «throttled». В іншому випадку використовується категорія «not_throttled». Таким чином, агент може враховувати, чи відбулося примусове обмеження процесорного ресурсу з боку механізмів квотування.

Після визначення всіх трьох компонентів формується ключ стану. Він має текстовий вигляд і складається з категорії CPU, категорії навантаження та категорії обмеження ресурсів. Наприклад, стан може бути поданий як

«medium|busy|not_throttled». Саме такий ключ використовується агентом для пошуку та оновлення відповідних Q-значень у таблиці.

Компонент агента навчання реалізовано в модулі `agent/q_learning.py`. Основним класом цього модуля є `QLearningAgent`. Він містить набір допустимих дій, параметри навчання, Q-таблицю, механізм вибору дії та функцію оновлення Q-значень. У поточній реалізації агент працює з табличним представленням, тобто для кожної пари «стан – дія» зберігається окреме числове Q-значення.

Простір дій агента є наперед визначеним і включає такі дії: «noop», «increase_weight», «decrease_weight», «increase_quota», «decrease_quota», «increase_priority» та «decrease_priority». Дія «noop» означає відсутність зміни параметрів керування. Інші дії відповідають абстрактним керуючим впливам, пов'язаним зі зміною ваги CPU, квоти CPU або пріоритету процесу. Конкретний спосіб виконання цих дій залежить від обраного режиму роботи системи та реалізується окремо в модулі `controller/backends.py`.

Вибір дії здійснюється методом `choose_action()`, у ньому використовується

ε - жадібна стратегія. З імовірністю ε агент обирає випадкову дію, що забезпечує дослідження середовища. В іншому випадку агент використовує накопичені Q-значення та обирає одну з дій із найбільшим відомим значенням для поточного стану. Якщо декілька дій мають однакове максимальне Q-значення, вибір між ними виконується випадково. Це дозволяє уникнути систематичного зміщення на користь першої дії у списку.

Оновлення Q-таблиці реалізовано методом `update()`. Після виконання дії агент отримує поточний стан, обрану дію, значення винагороди та наступний стан. На основі цих даних оновлюється Q-значення для відповідної пари «стан – дія». У реалізації використовуються три основні параметри:

	швидкість	навчання
є,	коефіцієнт дисконтування	та імовірність дослідження

є. За замовчуванням у програмі встановлено значення $\eta = 0,20$, $\gamma = 0\{, \}85$ та $\epsilon = 0,25$. Такі параметри є достатніми для демонстраційного прототипу, однак у складніших експериментальних сценаріях вони можуть бути змінені.

Для збереження результатів навчання в агенті передбачено роботу з Q-таблицею. У пам'яті програми вона зберігається як набір пар, де ключем є комбінація стану та дії, а значенням – відповідне Q-значення. Після завершення експерименту таблиця експортується у файл *q_table.json*. Також у програмі передбачено можливість завантаження попередньо збереженої таблиці, якщо під час запуску використано відповідний параметр. Це дозволяє не лише аналізувати результати окремого запуску, а й продовжувати навчання агента на основі вже накопиченого досвіду.

Функція винагороди, яка використовується агентом для оцінювання наслідків дії, реалізована в модулі *environment/reward.py*. Вона враховує поточний рівень використання процесора, перевантаження системи, надто низьке використання CPU, значення усередненого навантаження та зростання показника *throttled_usec*. Позитивна складова винагороди нараховується тоді, коли використання процесора перебуває в стабільному робочому діапазоні від 35 % до 75 %. Натомість штрафи застосовуються у випадках перевантаження, простою, високого навантаження або збільшення часу обмеження. Такий підхід дозволяє спрямувати поведінку агента на підтримання стабільного стану системи.

Для узагальнення параметрів агента Q-навчання доцільно навести таблицю 3.3.

У загальному циклі роботи системи агент не взаємодіє з операційною системою безпосередньо. Він лише отримує дискретний стан і повертає назву обраної дії. Подальше застосування цієї дії виконується контролером через відповідний модуль реалізації керуючих впливів. Такий поділ є важливим, оскільки дозволяє відокремити інтелектуальну частину системи від Linux-специфічних механізмів керування ресурсами.

Таблиця 3.3 – Основні параметри Q-learning агента

<i>Параметр</i>	<i>Значення за замовчуванням</i>	<i>Призначення</i>
η	0,20	Визначає швидкість оновлення Q-значень після отримання винагороди.
γ	0,85	Визначає вагу майбутніх винагород під час оновлення Q-таблиці.
ϵ	0,25	Визначає імовірність випадкового вибору дії для дослідження середовища.
<i>seed</i>	42	Забезпечує відтворюваність вибору дій у демонстраційних експериментах.
q_{table}	формується під час роботи	Зберігає оцінки корисності пар «стан – дія».

У загальному циклі роботи системи агент не взаємодіє з операційною системою безпосередньо. Він лише отримує дискретний стан і повертає назву обраної дії. Подальше застосування цієї дії виконується контролером через відповідний модуль реалізації керуючих впливів. Такий поділ є важливим, оскільки дозволяє відокремити інтелектуальну частину системи від Linux-специфічних механізмів керування ресурсами.

Отже, у програмному прототипі середовище описується через дискретизовані системні метрики, а агент навчання реалізує механізм адаптивного прийняття рішень на основі Q-таблиці. Така архітектурна реалізація є обчислювально ефективною, не створює надмірного навантаження на систему і водночас повноцінно демонструє ключові елементи навчання з підкріпленням: стан, дію, винагороду, оновлення політики поведінки та накопичення досвіду. При цьому система залишається експериментальним контролером простору користувача і не виконує функцій планувальника ядра Linux.

3.4 Реалізація керуючих дій у Linux

Реалізація керуючих дій у програмному прототипі винесена в окремий модуль *controller/backends.py*. Такий підхід дозволяє відокремити алгоритм прийняття рішень від конкретних механізмів впливу на операційну систему. Q-learning агент не змінює системні параметри безпосередньо, а лише обирає назву дії. Подальше застосування цієї дії виконується відповідним модулем реалізації, який залежить від режиму запуску програми.

У системі передбачено три режими роботи: режим симуляції «simulation», режим зміни пріоритету процесу «nice» та режим реального керування «real» для середовищ із підтримкою *cgroups v2*. Вибір режиму здійснюється під час запуску програми за допомогою параметра командного рядка *--mode*. Створення відповідного об'єкта реалізації виконується функцією *create_backend()*, яка повертає один із класів: *SimulationBackend*, *NiceReniceBackend* або *LinuxCgroupsV2Backend*.

Спільним форматом результату виконання дії є структура *ControlSnapshot*. Вона містить інформацію про режим роботи, поточні значення змодельованої ваги CPU, квоти CPU, періоду квотування, зміни пріоритету, ознаку фактичного застосування дії, текстову примітку, цільовий PID та поточне значення *nice*, якщо воно доступне. Завдяки цьому кожна ітерація експерименту може бути зафіксована в журналі незалежно від того, чи було реально змінено системний параметр, чи дія була лише змодельована.

У режимі симуляції використовується клас *SimulationBackend*. Цей режим є безпечним і призначений для демонстраційних запусків, зокрема в середовищах, де немає доступу до механізмів *cgroups v2* або небажано змінювати параметри операційної системи. У цьому режимі дії агента змінюють лише внутрішні змодельовані значення, зокрема *cpu_weight*,

cpu_quota та *nice_delta*. Фактичні системні файли або параметри процесів при цьому не змінюються, а поле *applied* у структурі *ControlSnapshot* має значення *False*.

Для дій «*increase_weight*» та «*decrease_weight*» у режимі симуляції змінюється внутрішнє значення *cpu_weight*. Збільшення виконується на 10 % із верхнім обмеженням 10000, а зменшення – на 10 % із нижнім обмеженням 1. Аналогічно для дій «*increase_quota*» та «*decrease_quota*» змінюється значення *cpu_quota*. У реалізації воно обмежується діапазоном від 10000 до 400000, а значення *cpu_period* залишається сталим і дорівнює 100000. Для дій «*increase_priority*» та «*decrease_priority*» змінюється лише внутрішній параметр *nice_delta*, який не впливає на реальний пріоритет процесу в режимі симуляції.

Другим режимом є режим «*nice*», реалізований класом *NiceReniceBackend*. Він використовує стандартні Unix-подібні механізми *os.getpriority()* та *os.setpriority()* для читання й зміни значення *nice* конкретного процесу. Для роботи цього режиму необхідно передати цільовий ідентифікатор процесу (PID) за допомогою параметра *--target-pid*. Нижче значення *nice* відповідає вищому пріоритету процесу, а вище значення *nice* – нижчому пріоритету.

У режимі «*nice*» фактично застосовуються лише дії, пов'язані зі зміною пріоритету: «*increase_priority*» та «*decrease_priority*». Дія «*increase_priority*» намагається зменшити значення *nice* на одиницю, але не нижче за -20. Дія «*decrease_priority*» збільшує значення *nice* на одиницю, але не вище за 19. Інші дії, пов'язані з вагою або квотою CPU, у цьому режимі лише оновлюють внутрішні змодельовані значення і не змінюють операційну систему. Це обмеження є важливим, оскільки механізм *nice/renice* не керує квотами або вагами CPU, а впливає лише на пріоритет планування процесу.

Зміна значення *nice* залежить від прав доступу користувача. Зазвичай користувач може знизити пріоритет власного процесу, тобто збільшити

значення *nice*, але підвищення пріоритету через зменшення значення *nice* може потребувати розширених прав. У програмі це враховано через обробку винятків *PermissionError*, *ProcessLookupError* та інших помилок операційної системи. Якщо зміну виконати неможливо, відповідна інформація фіксується в полі *note*, а ознака *applied* залишається хибною.

Третім режимом є режим реального керування «real», реалізований класом *LinuxCgroupsV2Backend*. Він призначений для запуску в операційній системі Linux із підтримкою *cgroups v2*. Для роботи цього режиму необхідно передати шлях до вже наявної ресурсної групи за допомогою параметра *--cgroup-path*. Програма не створює ресурсну групу автоматично, а працює з уже існуючою директорією, у якій мають бути доступні відповідні файли керування CPU.

У режимі реального керування дії «*increase_weight*» та «*decrease_weight*» застосовуються через запис нового значення у файл *cpu.weight*. Перед записом програма перевіряє наявність цього файлу та права на запис. Якщо файл існує і доступний для запису, значення *cpu_weight* оновлюється, а дія вважається застосованою. Якщо файл відсутній або недоступний, дія не виконується, а в журналі фіксується відповідна примітка.

Дії «*increase_quota*» та «*decrease_quota*» у режимі «real» застосовуються через файл *cpu.max*. У цьому файлі записується пара значень: квота CPU та період квотування. У програмному прототипі період має стале значення 100000, а квота змінюється залежно від обраної дії. Як і у випадку з *cpu.weight*, перед записом перевіряється наявність файлу та можливість запису. Це дозволяє уникнути неконтрольованих помилок у середовищах, де поточний користувач не має необхідних прав доступу.

Дії, пов'язані зі зміною пріоритету процесу, у режимі реального керування можуть виконуватися через внутрішнє використання механізму *NiceReniceBackend*. Якщо під час запуску було передано параметр *--target-pid*, дії «*increase_priority*» та «*decrease_priority*» застосовуються до

відповідного процесу. Якщо цільовий PID не передано, зміна пріоритету пропускається, а в журналі експерименту зберігається повідомлення про те, що дія *nice/renice* не була виконана.

Для узагальнення особливостей режимів роботи доцільно навести таблицю 3.4.

Таблиця 3.4 – Режими застосування керуючих дій

<i>Режим</i>	<i>Клас реалізації</i>	<i>Необхідні параметри</i>	<i>Реальний вплив на систему</i>
«simulation»	<i>SimulationBackend</i>	Не потребує додаткових параметрів	Не змінює операційну систему; оновлює лише внутрішні змодельовані значення.
«nice»	<i>NiceReniceBackend</i>	<i>--target-pid</i>	Може змінювати значення <i>nice</i> вибраного процесу, якщо користувач має відповідні права.
«real»	<i>LinuxCgroupsV2Backend</i>	<i>--cgroup-path</i> , опційно <i>--target-pid</i>	Може змінювати <i>cpu.weight</i> та <i>cpu.max</i> у <i>cgroups v2</i> ; дії пріоритету можливі лише за наявності PID.

Важливо зазначити, що режим «real» не перетворює програму на власний планувальник ядра Linux. Контролер не втручається у внутрішню логіку планувальника, не модифікує ядро і не замінює стандартні механізми керування процесами. Він лише використовує наявні інтерфейси операційної

системи, зокрема *cgroups v2* та *nice/renice*, для експериментального застосування рішень, які обирає Q-learning агент.

Таким чином, реалізація керуючих дій у програмному прототипі побудована як багаторежимний механізм взаємодії з операційною системою. Режим симуляції забезпечує безпечну перевірку логіки адаптивного керування, режим «*nice*» дозволяє експериментувати зі зміною пріоритету конкретного процесу, а режим реального керування забезпечує можливість роботи з параметрами *cgroups v2* у Linux-середовищі. Такий підхід відповідає експериментальному характеру роботи та дозволяє дослідити адаптивне керування процесорними ресурсами без розроблення власного планувальника ядра.

3.5 Сценарій проведення експериментів

Для перевірки працездатності програмного прототипу було передбачено експериментальний сценарій, який дозволяє оцінити роботу адаптивного контуру керування без втручання в ядро операційної системи. Основна мета експерименту полягає у перевірці того, чи здатна система послідовно збирати метрики, формувати дискретний стан середовища, обирати керуючу дію за допомогою агента Q-навчання, обчислювати винагороду та зберігати результати для подальшого аналізу.

Експериментальний сценарій реалізовано через головний файл запуску *main.py*. Під час старту програми користувач задає режим роботи, тривалість експерименту, інтервал між ітераціями, шлях до директорії результатів, а також додаткові параметри для окремих режимів. Зокрема, для режиму «*nice*» може передаватися ідентифікатор цільового процесу (PID), а для режиму реального керування – шлях до ресурсної групи *cgroups v2*.

Базовим і найбільш безпечним сценарієм перевірки є запуск у режимі симуляції «simulation». У цьому режимі програма виконує повний цикл адаптивного керування, однак не змінює реальні параметри операційної системи. Це дозволяє перевірити логіку роботи контролера, агента навчання, функції винагороди, журналювання та побудови графіків навіть у середовищах, де відсутня підтримка *sgroups v2* або немає прав доступу до системних файлів керування процесором.

Типовий запуск експерименту в режимі симуляції виконується з тривалістю 60 секунд та інтервалом між ітераціями 1 секунда. Для забезпечення відтворюваності вибору дій використовується параметр `--seed`, який задає початкове значення генератора випадкових чисел. У межах демонстраційного сценарію доцільно використовувати значення `seed = 42`. Такий підхід дозволяє отримувати порівнювані результати під час повторних запусків експерименту.

Загальний вигляд команди запуску в режимі симуляції має такий вигляд:

```
python3 main.py --mode simulation --duration 60 --interval 1 --seed 42
```

Для створення додаткового навантаження на процесор у проєкті передбачено окремий модуль *load_generator.py*. Він запускає задану кількість робочих процесів, які виконують обчислювальне навантаження протягом визначеного часу. Такий підхід дозволяє сформувати більш динамічні умови експерименту, за яких значення використання CPU та *load average* змінюються впродовж запуску. Генератор навантаження запускається окремо від основного контролера, зазвичай в іншому вікні терміналу.

Приклад запуску генератора навантаження:

```
python3 load_generator.py --workers 4 --duration 60
```

У разі перевірки режиму «nice» генератор навантаження може додатково зберігати ідентифікатори створених процесів у файл. Це дає змогу обрати один із PID як ціль для керування пріоритетом. Після цього основний

контролер запускається з параметром `--target-pid`. У цьому сценарії агент може обирати дії, пов'язані зі зміною пріоритету процесу, однак фактичне застосування таких дій залежить від прав доступу користувача та політик операційної системи.

Приклад запуску генератора навантаження зі збереженням PID:

```
python3 load_generator.py --workers 2 --duration 90 --pid-file
results/load_pids.txt
```

Приклад запуску контролера в режимі «nice»:

```
python3 main.py --mode nice --target-pid 1234 --duration 60 --interval 1 --
seed 42
```

Окремим експериментальним сценарієм є режим реального керування «real», який призначений для середовищ Linux із підтримкою *cgroups* v2. У цьому випадку програма може записувати значення у файли *cpu.weight* та *cpu.max* у вже створеній ресурсній групі. Для такого запуску необхідно заздалегідь створити відповідну директорію *cgroup*, передати шлях до неї через параметр `--cgroup-path` і забезпечити права доступу на запис до файлів керування. Якщо ці умови не виконані, режим реального керування не може бути коректно перевірений.

Приклад запуску в режимі реального керування:

```
python3 main.py --mode real --cgroup-path /sys/fs/cgroup/adaptive_demo --
duration 60 --interval 1 --seed 42
```

У межах цієї дипломної роботи основним безпечним сценарієм валідації є режим симуляції, оскільки він дозволяє перевірити повний алгоритмічний цикл без ризику некоректного впливу на операційну систему. Режими «nice» та «real» розглядаються як додаткові експериментальні можливості, які демонструють інтеграцію прототипу зі стандартними механізмами Unix-подібних систем і Linux. При цьому режим «real» потребує саме Linux-середовища з підтримкою *cgroups* v2 та відповідних прав доступу.

Під час кожної ітерації експерименту контролер виконує однакову послідовність дій. Спочатку модуль збору метрик отримує поточний стан системи. Далі ці метрики перетворюються на дискретний стан, який передається агенту Q-навчання. Агент обирає одну з допустимих дій, після чого відповідний модуль застосування намагається виконати її згідно з обраним режимом роботи. Після паузи, що визначається параметром --interval, система повторно збирає метрики, обчислює винагороду та оновлює Q-таблицю. Кожна ітерація записується до журналу експерименту.

У журналі експерименту зберігаються номер кроку, час від початку запуску, дискретні категорії стану, ключ стану, обрана дія, значення винагороди, фактичні метрики системи, режим роботи, ознака застосування дії, параметри керування CPU та службові примітки. Така структура журналу дозволяє не лише оцінити загальну динаміку експерименту, а й простежити, які рішення приймав агент у різних станах середовища.

Після завершення запуску програма формує набір файлів результатів у директорії *results* або в іншій директорії, заданій параметром --results-dir. До основних результатів належать файл *experiment_results.csv*, файл *experiment_results.xlsx*, текстовий звіт *summary.txt*, графіки *cpu_usage.png*, *reward.png*, *actions.png* та файл *q_table.json*. Якщо експеримент було перервано комбінацією клавіш *Ctrl+C*, програма зберігає часткові результати, зібрані до моменту переривання.

Для узагальнення основних параметрів експериментального сценарію доцільно навести таблицю 3.5.

Таблиця 3.5 – Основні параметри сценарію експерименту

<i>Параметр</i>	<i>Типове значення</i>	<i>Призначення</i>
<i>--mode</i>	«simulation»	Визначає режим роботи контролера.
<i>--duration</i>	60	Визначає тривалість експерименту в секундах.
<i>--interval</i>	1,0	Визначає паузу між ітераціями керування в секундах.
<i>--seed</i>	42	Забезпечує відтворюваність випадкового вибору дій.
<i>--target-pid</i>	задається за потреби	Використовується в режимі «nise» для вибору процесу.
<i>--cgroup-path</i>	задається за потреби	Використовується в режимі «real» для доступу до ресурсної групи <i>cgroups v2</i> .
<i>--results-dir</i>	<i>results</i>	Визначає директорію для збереження результатів експерименту.

Таким чином, сценарій проведення експериментів побудований так, щоб послідовно перевірити всі ключові компоненти програмного прототипу: збір метрик, дискретизацію стану, роботу агента Q-навчання, застосування керуючих дій, обчислення винагороди та збереження результатів. Основний акцент зроблено на безпечній і відтворюваній перевірці адаптивного циклу, тоді як режими «nise» та «real» використовуються для демонстрації можливості інтеграції з реальними механізмами керування ресурсами в Unix-подібних системах і Linux.

3.6 Аналіз результатів експериментів

Після завершення експериментального запуску програмний прототип формує набір результатів, які дають змогу оцінити роботу адаптивного контуру керування. Основними файлами для аналізу є *experiment_results.csv*, *experiment_results.xlsx*, *summary.txt*, графіки *cpu_usage.png*, *reward.png*, *actions.png* та файл *q_table.json*. У цьому підрозділі розглянуто результати запуску контролера в режимі симуляції «simulation», оскільки цей режим є

безпечним і дозволяє перевірити повний алгоритмічний цикл без зміни параметрів операційної системи.

Файл *experiment_results.csv* містить покроковий журнал роботи контролера. Кожен рядок відповідає одній ітерації адаптивного циклу. У журналі зберігаються номер кроку, час від початку запуску, дискретні категорії стану, ключ стану, обрана дія, значення винагороди, фактичне використання процесора, значення *load_avg_1m*, кількість процесів, показник *throttled_usec*, режим роботи, ознака фактичного застосування дії та службова примітка. Така структура журналу дозволяє простежити, як змінювався стан системи та які рішення приймав агент упродовж експерименту.

Файл *summary.txt* містить узагальнені показники експерименту. Зокрема, у ньому подано кількість виконаних кроків, середнє та максимальне використання процесора, середню, найкращу й найгіршу винагороду, режим роботи, кількість дій, фактично застосованих до операційної системи, а також розподіл обраних агентом дій. Ці показники є зручними для швидкої оцінки результатів без необхідності переглядати повний журнал експерименту.

Для узагальнення основних числових результатів доцільно навести таблицю 3.6.

Оскільки основний експеримент виконувався в режимі симуляції, значення показника фактичного застосування дій до операційної системи дорівнює нулю. Це є очікуваним результатом, оскільки режим «simulation» призначений для безпечної перевірки логіки адаптивного керування. У цьому режимі змінюються лише внутрішні змодельовані параметри, а реальні значення *cpu.weight*, *cpu.max* або *nice* не змінюються.

Таблиця 3.6 – Узагальнені результати експериментального запуску

Показник	Значення
Кількість кроків експерименту	60
Середнє використання CPU, %	40,35
Максимальне використання CPU, %	48,00
Середнє значення винагороди	0,0233
Найкраще значення винагороди	0,1582
Найгірше значення винагороди	-0,0833
Найчастіше обрана дія	increase_weight
Режим роботи	simulation
Кількість дій, застосованих до ОС	0

Динаміку використання процесора відображено на рисунку 3.1. Графік *cpu_usage.png* показує зміну показника CPU упродовж експерименту. За цим графіком можна оцінити, чи перебувало навантаження в стабільному діапазоні, чи спостерігалися періоди низького використання процесора або перевантаження. Якщо значення CPU переважно перебуває в межах від 35 % до 75 %, це відповідає цільовому діапазону, який використовується у функції винагороди. Якщо ж значення часто опускається нижче 35 % або перевищує 75 %, це призводить до зменшення винагороди та впливає на подальше оновлення Q-таблиці.

Динаміку винагороди наведено на рисунку 3.2. Графік *reward.png* дозволяє оцінити, наскільки успішними були рішення агента з погляду заданої функції винагороди. Додатні або вищі значення винагороди відповідають станам, у яких використання процесора перебувало ближче до стабільного робочого діапазону. Від’ємні або нижчі значення винагороди свідчать про наявність небажаних станів, зокрема перевантаження, простою, високого системного навантаження або зростання показника *throttled_usec*.

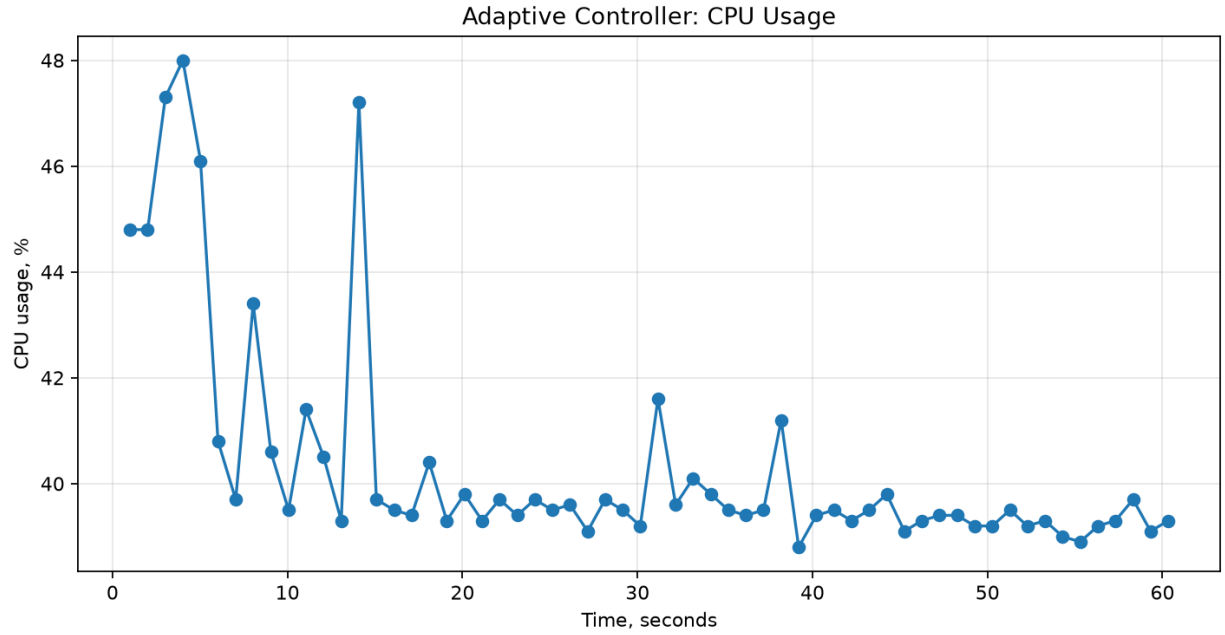


Рисунок 3.1 – Динаміка використання CPU під час експерименту

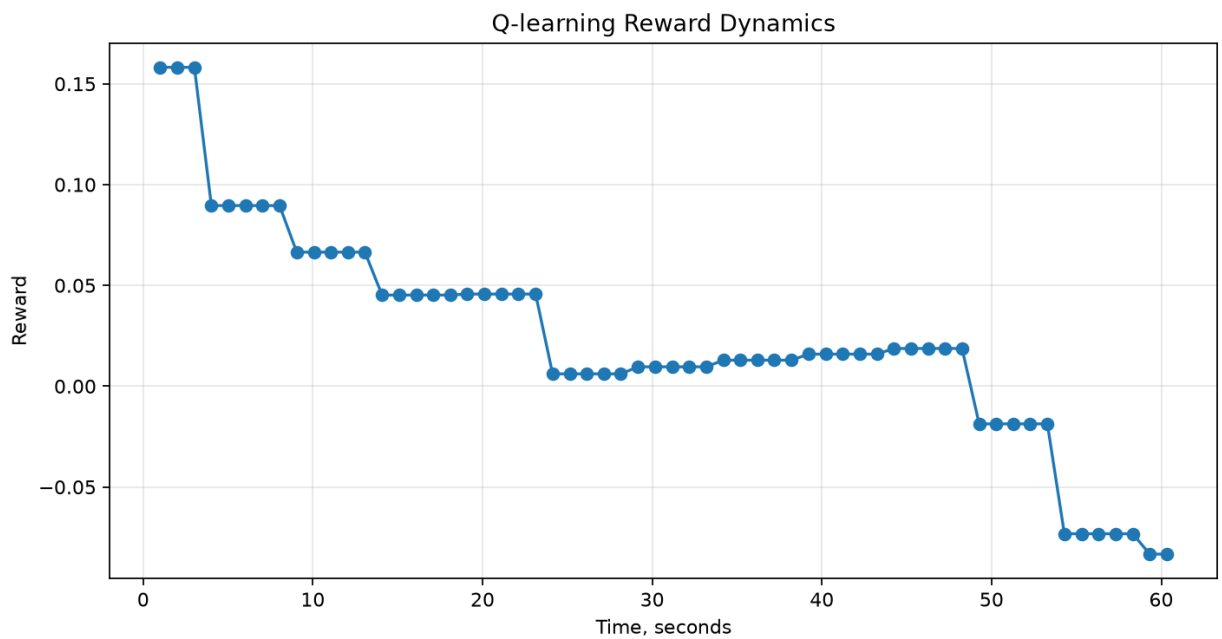


Рисунок 3.2 – Динаміка винагороди Q-learning агента

Окрему увагу слід приділити розподілу обраних дій, який подано на рисунку 3.3. Графік *actions.png* показує, які дії агент обирав найчастіше. У режимі симуляції та на початкових етапах навчання значну роль відіграє ϵ -

жадібна стратегія, тому частина дій обирається випадково з метою дослідження середовища. Це пояснює можливу нерівномірність розподілу дій і наявність як активних керуючих дій, так і дії «noop», яка означає відсутність зміни параметрів керування.

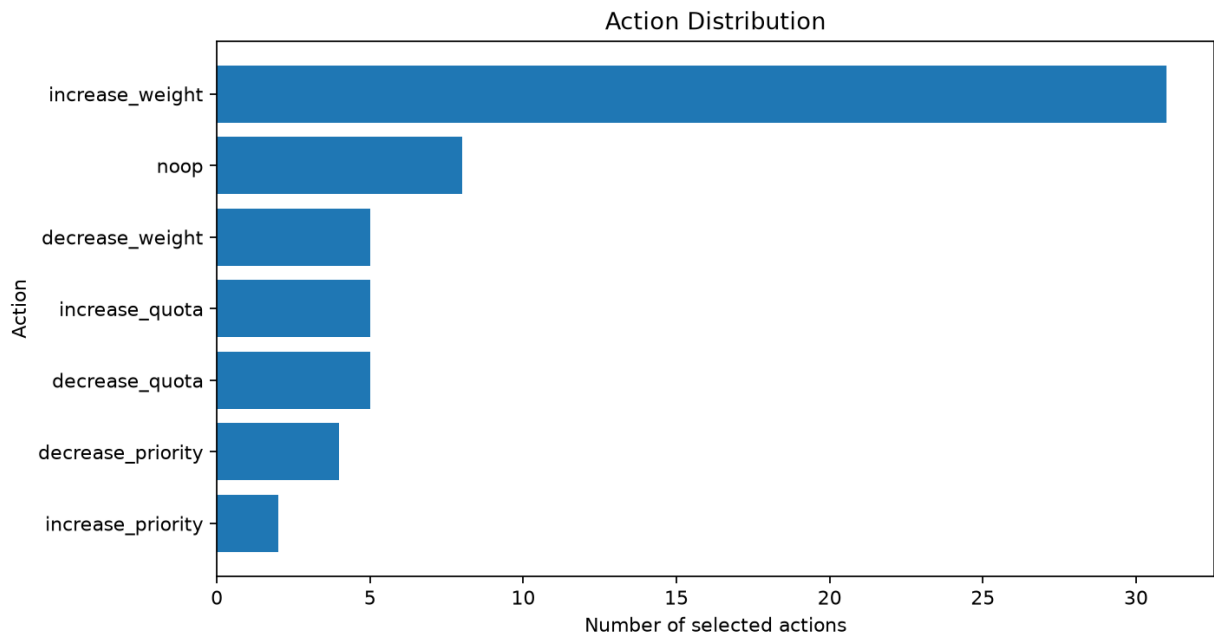


Рисунок 3.3 – Розподіл дій, обраних агентом під час експерименту

Аналіз розподілу дій дозволяє оцінити, чи агент використовував різні варіанти керування, чи переважно зосереджувався на одній дії. Якщо найчастіше обраною дією є «noop», це може свідчити про те, що в частині станів агент не мав достатніх підстав для активного впливу або ще не накопичив достатнього досвіду для стабільного вибору інших дій. Якщо ж переважають дії, пов'язані зі збільшенням або зменшенням ваги, квоти чи пріоритету, це означає, що агент активно досліджував простір керуючих впливів.

Для більш детального аналізу результатів доцільно використовувати файл *experiment_results.xlsx*. Він містить як повний журнал експерименту, так і окремий аркуш із підсумковими метриками. Формат XLSX є зручним для ручного перегляду результатів, фільтрації рядків за діями або станами, а

також для перевірки окремих ітерацій, у яких було отримано найкраще або найгірше значення винагороди.

Отримані результати підтверджують, що програмний прототип виконує повний цикл адаптивного керування. Система збирає метрики, перетворює їх у дискретний стан, передає стан агенту Q-навчання, обирає керуючу дію, обчислює винагороду та оновлює Q-таблицю. У режимі симуляції ці дії не змінюють параметри операційної системи, що робить такий запуск придатним для безпечної перевірки алгоритмічної частини роботи.

Водночас результати експерименту слід інтерпретувати з урахуванням обмежень прототипу. Тривалість запуску є відносно невеликою, а табличний Q-learning потребує накопичення досвіду в різних станах середовища. Тому результати окремого запуску не слід розглядати як доказ оптимального керування ресурсами. Вони демонструють працездатність запропонованої архітектури, коректність взаємодії між модулями та можливість використання навчання з підкріпленням для адаптивного вибору керуючих дій.

Таким чином, експериментальні результати підтверджують функціональність розробленого програмного прототипу як адаптивного контролера простору користувача. Система не виконує функцій планувальника ядра Linux, однак демонструє можливість побудови надбудови, яка аналізує поточні системні метрики та приймає рішення на основі Q-learning агента. Отримані файли результатів і графіки є достатніми для подальшого аналізу поведінки агента та оцінювання ефективності запропонованого підходу.

3.7 Аналіз Q-таблиці та поведінки агента

Одним із ключових результатів роботи агента Q-навчання є сформована Q-таблиця, яка зберігається у файлі *q_table.json*. Вона відображає накопичений агентом досвід взаємодії із середовищем і містить числові оцінки корисності для окремих пар «стан – дія». Саме на основі цих значень агент надалі може обирати керуючі дії не лише випадково, а й з урахуванням результатів попередніх ітерацій експерименту.

У програмній реалізації Q-таблиця формується в модулі *agent/q_learning.py* у класі *QLearningAgent*. У пам'яті програми вона зберігається як словник, у якому ключем є пара, що складається з дискретного стану та дії, а значенням – відповідна оцінка корисності. Після завершення експерименту таблиця експортується у файл *q_table.json*. Для зручності збереження ключі у файлі мають текстовий формат, у якому стан і дія розділені службовим розділювачем. Наприклад, запис може мати вигляд «medium|busy|not_throttled::increase_quota».

Дискретний стан, який використовується в Q-таблиці, формується в модулі *environment/state.py*. Він складається з трьох компонентів: рівня використання процесора, рівня системного навантаження та ознаки обмеження процесорного часу. Кожен стан записується як текстовий ключ, наприклад «low|normal|not_throttled», «medium|busy|not_throttled» або «high|overloaded|throttled». Такий формат є зручним для аналізу, оскільки з нього одразу видно, за яких умов агент приймав певне рішення.

Загальна кількість можливих дискретних станів у поточній реалізації є обмеженою. Рівень використання процесора має три категорії: «low», «medium» і «high». Рівень системного навантаження також має три категорії: «normal», «busy» і «overloaded». Ознака обмеження процесорного часу має дві категорії: «throttled» і «not_throttled». Отже, теоретично можливими є 18

дискретних станів. Оскільки агент має 7 доступних дій, максимальна кількість пар «стан – дія» у повністю заповненій Q-таблиці становить 126.

Проте під час короткого експериментального запуску не всі можливі стани обов’язково виникають. Q-таблиця заповнюється лише для тих пар «стан – дія», з якими агент фактично зіткнувся під час роботи. Якщо для певної пари значення ще не було створено, у процесі вибору дії воно розглядається як нульове. Це є типовою особливістю табличного Q-навчання: агент поступово накопичує оцінки корисності лише для відвіданих станів і виконаних дій.

Оновлення Q-значень відбувається після кожної ітерації адаптивного циклу. Агент отримує поточний стан, обрану дію, значення винагороди та наступний стан. На основі цих даних метод *update()* змінює оцінку відповідної пари «стан – дія». Якщо виконана дія призвела до кращого результату з погляду функції винагороди, її Q-значення зростає. Якщо ж після виконання дії система перейшла в небажаний стан, наприклад у стан перевантаження або зростання обмеження процесорного часу, відповідна оцінка може зменшуватися.

Для аналізу Q-таблиці доцільно розглядати не лише окремі числові значення, а й найкращу дію для кожного стану. Це дозволяє оцінити, яку поведінку агент починає формувати внаслідок навчання. Наприклад, для кожного стану можна порівняти Q-значення всіх доступних дій і визначити дію з найбільшим значенням. Якщо для стану «high|overloaded|not_throttled» найвищу оцінку має дія, пов’язана зі зменшенням навантаження або обмеженням ресурсу, це може свідчити про формування логічної реакції на перевантаження. Якщо ж для стабільного стану найвищу оцінку має дія «поор», це може означати, що агент не вбачає потреби в активному втручанні.

Для узагальнення структури аналізу Q-таблиці доцільно навести таблицю 3.7.

Таблиця 3.7 – Приклад аналізу записів Q-таблиці

Стан	Найкраща дія	Q-значення	Інтерпретація
low busy not_throttled	noop	0,068265	Для стану з невисоким рівнем CPU та зайнятим навантаженням найкращою стала відсутність втручання. Це означає, що агент не мав підстав активно змінювати параметри в цьому стані.
low overloaded not_throttled	increase_weight	0,001876	Для стану з низьким CPU, але підвищеним системним навантаженням агент найкраще оцінив збільшення ваги CPU. Значення близьке до нуля, тому це слід трактувати як початкову перевагу, а не як доведену оптимальну дію.
high busy not_throttled	decrease_priority	-0,013919	Для стану з високим рівнем CPU і зайнятим навантаженням агент обрав зниження пріоритетності як найкращу з наявних оцінених дій.
high overloaded not_throttled	decrease_quota	0,003180	Для перевантаженого стану з високим CPU найкращою серед зафіксованих стала дія зменшення квоти, однак її Q-значення від'ємне, тому результат потрібно інтерпретувати обережно.

Окремо слід враховувати вплив ϵ - жадібної стратегії. Оскільки агент із певною імовірністю обирає випадкову дію, поведінка під час одного короткого запуску не завжди буде повністю відповідати найбільшим Q-значенням. Це не є помилкою реалізації, а є частиною механізму дослідження середовища. Завдяки випадковим діям агент має змогу перевіряти різні варіанти керування і поступово уточнювати оцінки в Q-таблиці.

Аналіз Q-таблиці доцільно виконувати разом із аналізом розподілу дій із файлу *actions.png* та журналом *experiment_results.xlsx*. Графік розподілу дій показує, які дії агент обирав найчастіше, тоді як Q-таблиця пояснює, які дії отримали вищі оцінки в конкретних станах. Якщо певна дія часто

трапляється в журналі, але має невисоке Q-значення, це може бути наслідком випадкового вибору в межах ϵ -жадібної стратегії. Якщо ж дія часто обирається і водночас має високе Q-значення для відповідного стану, це свідчить про поступове закріплення такої поведінки в агента.

У режимі симуляції Q-таблиця відображає насамперед алгоритмічну поведінку агента, а не реальний фізичний ефект зміни параметрів операційної системи. Це важливо враховувати під час інтерпретації результатів. У цьому режимі дії змінюють внутрішні змодельовані параметри, тому Q-значення демонструють, як агент реагував на дискретизовані стани та отриману винагороду, але не доводять оптимальність керування реальними ресурсами Linux.

Водночас навіть у режимі симуляції аналіз Q-таблиці є корисним, оскільки він підтверджує роботу основного інтелектуального механізму системи. Наявність записів у *q_table.json* означає, що агент не лише випадково обирає дії, а й оновлював оцінки корисності на основі винагороди. Це відрізняє розроблений прототип від статичного набору правил, оскільки поведінка агента змінюється залежно від накопиченого досвіду.

Під час інтерпретації Q-таблиці також потрібно враховувати тривалість експерименту. За запуску тривалістю 60 секунд і з інтервалом 1 секунда агент виконує відносно невелику кількість ітерацій. Для табличного Q-навчання цього достатньо для демонстрації принципу роботи, однак недостатньо для формування повністю стабільної політики поведінки для всіх можливих станів. Тому Q-таблицю слід розглядати як результат початкового навчання, а не як остаточну оптимальну політику керування.

Для глибшого аналізу поведінки агента можна порівняти три джерела даних: файл *experiment_results.xlsx*, графік *actions.png* і файл *q_table.json*. Журнал експерименту показує послідовність станів, дій і винагород. Графік дій узагальнює частоту вибору кожної дії. Q-таблиця, у свою чергу, дозволяє побачити, які дії агент почав оцінювати як більш корисні в конкретних

станах середовища. Разом ці дані дають змогу зробити висновок про характер поведінки агента під час експерименту.

Таким чином, аналіз Q-таблиці підтверджує, що в програмному прототипі реалізовано не лише збір метрик і виконання наперед заданих дій, а й механізм навчання з підкріпленням. Агент накопичує досвід у вигляді Q-значень, використовує його для подальшого вибору дій і може змінювати свою поведінку залежно від отриманих винагород. При цьому результати Q-таблиці потрібно інтерпретувати обережно: вони демонструють працездатність підходу та початкове формування політики поведінки, але не є доказом глобально оптимального керування ресурсами Linux.

Висновки до розділу 3

У третьому розділі було розглянуто програмну реалізацію прототипу адаптивного керування процесорними ресурсами в операційній системі Linux з використанням методів навчання з підкріпленням. Розроблена система побудована як контролер простору користувача, який не змінює ядро Linux і не замінює стандартний планувальник процесів, а використовує доступні інтерфейси операційної системи для експериментального керування параметрами процесів і ресурсних груп.

Було описано загальну архітектуру програмного прототипу, яка включає модулі збору системних метрик, дискретизації стану, агента Q-навчання, адаптивного контролера, модулі застосування керуючих дій та засоби збереження результатів. Така структура забезпечує логічне розділення функцій між компонентами системи та спрощує аналіз роботи кожного з них.

Реалізований механізм збору метрик дозволяє отримувати значення використання процесора, усередненого системного навантаження, кількості

процесів і показника *throttled_usec* за наявності доступу до *cgroups v2*. На основі цих метрик формується дискретний стан середовища, який використовується агентом Q-навчання для вибору керуючої дії. Це дозволяє перетворити неперервні системні показники у скінченний простір станів, придатний для табличного навчання з підкріпленням.

Інтелектуальний компонент системи реалізовано у вигляді агента Q-навчання, який працює з Q-таблицею, використовує е-жадібну стратегію вибору дії та оновлює оцінки корисності на основі отриманої винагороди. Саме цей компонент забезпечує адаптивне прийняття рішень, оскільки поведінка агента залежить не лише від поточного стану системи, а й від накопиченого досвіду попередніх ітерацій.

У розділі також було розглянуто реалізацію керуючих дій у трьох режимах роботи. Режим симуляції «simulation» забезпечує безпечну перевірку логіки адаптивного циклу без зміни параметрів операційної системи. Режим «nice» дозволяє експериментувати зі зміною пріоритету конкретного процесу за його ідентифікатором процесу (PID), якщо користувач має відповідні права доступу. Режим реального керування «real» призначений для Linux-середовищ із підтримкою *cgroups v2* і може працювати з файлами *cru.weight* та *cru.max* за умови наявності відповідної ресурсної групи та прав на запис.

Описаний сценарій проведення експериментів показує, що програмний прототип здатний послідовно виконувати повний цикл адаптивного керування: збирати метрики, формувати дискретний стан, обирати дію, застосовувати її відповідно до режиму роботи, обчислювати винагороду, оновлювати Q-таблицю та зберігати результати. За результатами запуску формуються файли *experiment_results.csv*, *experiment_results.xlsx*, *summary.txt*, *q_table.json*, а також графіки динаміки використання CPU, винагороди та розподілу дій.

Аналіз Q-таблиці підтверджує, що розроблена система реалізує не статичний набір правил, а механізм навчання з підкріпленням. Агент накопичує оцінки корисності для пар «стан – дія» та може змінювати подальший вибір дій залежно від отриманих винагород. Водночас результати експериментів слід інтерпретувати з урахуванням обмежень прототипу: короткий запуск не забезпечує формування повністю оптимальної політики, а режим симуляції демонструє насамперед алгоритмічну працездатність системи.

Таким чином, у третьому розділі було підтверджено практичну реалізованість запропонованого підходу. Розроблений програмний прототип демонструє можливість побудови адаптивного контролера простору користувача, який аналізує поточний стан системи, використовує агент Q-навчання для прийняття рішень і зберігає результати експериментів для подальшого аналізу. Отримана реалізація відповідає меті роботи та може бути використана як основа для подальшого розвитку систем адаптивного керування ресурсами в Linux.

ВИСНОВКИ

У дипломній роботі розглянуто задачу адаптивного керування процесами в операційній системі Linux з використанням методів штучного інтелекту. Показано, що стандартні механізми Linux, зокрема nice/renice, cgroups v2, cpu.weight, cpu.max, /proc та sysfs, можуть бути використані як основа для побудови user-space adaptive controller без модифікації планувальника ядра.

Проаналізовано проблему статичного керування ресурсами, яка полягає в необхідності попереднього вибору параметрів і недостатній гнучкості в умовах змінного навантаження. Обґрунтовано доцільність застосування навчання з підкріпленням для послідовного вибору керуючих дій на основі поточних системних метрик.

Математичну модель адаптивного керування подано у вигляді марковського процесу прийняття рішень. Стан системи формується на основі дискретизованих метрик використання CPU, системного навантаження та ознак throttling, а простір дій пов'язано зі зміною параметрів керування процесами або групами процесів. Для вибору дій застосовано табличний алгоритм Q-learning.

У програмній частині реалізовано експериментальний прототип адаптивного контролера простору користувача. Система виконує повний цикл керування: збір метрик, формування дискретного стану, вибір дії агентом, застосування дії через backend, обчислення винагороди, оновлення Q-таблиці та збереження результатів експерименту.

У межах експериментального середовища простір станів складає 18 дискретних станів, а максимальна кількість пар «стан – дія» у Q-таблиці становить 126. Отже, застосування табличного Q-learning є придатним для обраної дискретизації та забезпечує інтерпретованість поведінки агента.

Функціонально-вартісний аналіз показав, що сумарна трудомісткість розроблення програмного прототипу становить 156 люд.-год, а загальні витрати на розроблення – 43 000 грн. Найбільш доцільним варіантом реалізації визначено Python-прототип, оскільки він забезпечує достатню функціональність, безпечний імітаційний режим та можливість подальшого розширення.

Практичне значення роботи полягає у створенні експериментального програмного прототипу, який може бути використаний як основа для подальших досліджень адаптивного керування системними ресурсами, автоматизованого моніторингу навантаження та інтелектуальної підтримки керування процесами в ОС Linux.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Michael Kerrisk: The Linux Programming Interface: A Linux and UNIX System Programming Handbook. 2010.URL: <https://man7.org/tlpi/>
2. Linux man-pages project: sched(7) – Linux manual page. Б. р.URL: <https://man7.org/linux/man-pages/man7/sched.7.html>
3. Ingo Molnar, Linux kernel community: CFS Scheduler. 2007.URL: <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>
4. Linux kernel community: EEVDF Scheduler. Б. р.URL: <https://docs.kernel.org/scheduler/sched-eevdf.html>
5. Linux kernel community: Scheduler Nice Design. Б. р.URL: <https://docs.kernel.org/scheduler/sched-nice-design.html>
6. Linux man-pages project: nice(2) – Linux manual page. Б. р.URL: <https://man7.org/linux/man-pages/man2/nice.2.html>
7. Linux man-pages project, util-linux: renice(1) – Linux manual page. Б. р.URL: <https://man7.org/linux/man-pages/man1/renice.1.html>
8. Linux kernel community: The /proc Filesystem. Б. р.URL: <https://docs.kernel.org/filesystems/proc.html>
9. Linux man-pages project: proc_pid_stat(5) – Linux manual page. Б. р.URL: https://man7.org/linux/man-pages/man5/proc_pid_stat.5.html
10. Linux man-pages project: proc_pid_status(5) – Linux manual page. Б. р.URL: https://man7.org/linux/man-pages/man5/proc_pid_status.5.html
11. Patrick Mochel, Mike Murphy, Linux kernel community: sysfs – The filesystem for exporting kernel objects. 2003, revised 2011.URL: <https://docs.kernel.org/filesystems/sysfs.html>
12. Tejun Heo, Linux kernel community: Control Group v2. 2015. URL: <https://docs.kernel.org/admin-guide/cgroup-v2.html>

13. Linux man-pages project: cgroups(7) – Linux manual page. B. p.URL: <https://man7.org/linux/man-pages/man7/cgroups.7.html>
14. Richard S. Sutton, Andrew G. Barto: Reinforcement Learning: An Introduction. 2018.URL: <https://mitpress.mit.edu/9780262039246/reinforcement-learning/>
15. Christopher J. C. H. Watkins, Peter Dayan: Technical Note: Q-Learning. 1992.URL: <https://link.springer.com/article/10.1007/BF00992698>
16. Hongzi Mao, Mohammad Alizadeh, Ishai Menache, Srikanth Kandula: Resource Management with Deep Reinforcement Learning. 2016.URL: <https://www.microsoft.com/en-us/research/publication/resource-management-deep-reinforcement-learning/>
17. Krzysztof Rządca, Paweł Findeisen, Jacek Świdorski, Przemysław Zych, Piotr Broniek, Jędrzej Kusmerek, Paweł Nowak, Beata Strack, Piotr Witusowski, Steven Hand, John Wilkes: Autopilot: Workload Autoscaling at Google Scale. 2020.URL: <https://research.google/pubs/autopilot-workload-autoscaling-at-google-scale/>
18. Jeffrey Dean, Luiz André Barroso: The Tail at Scale. 2013.URL: <https://research.google/pubs/the-tail-at-scale/>

ДОДАТОК А ОСНОВНІ МОДУЛІ ПРОГРАМНОГО ПРОТОТИПУ

Лістинг А.1 – Головний файл запуску main.py

```

from __future__ import annotations

import argparse
from pathlib import Path

from agent.q_learning import QLearningAgent
from controller.adaptive_controller import AdaptiveController
from controller.backends import create_backend
from experiments.exporter import export_results
from experiments.plots import create_plots
from monitoring.metrics import MetricsCollector

def main() -> None:
    parser = argparse.ArgumentParser(description="Adaptive Linux RL
Controller prototype")
    parser.add_argument("--duration", type=int, default=60, help="Experiment
duration in seconds")
    parser.add_argument("--interval", type=float, default=1.0, help="Control
loop interval in seconds")
    parser.add_argument(
        "--mode",
        choices=["simulation", "nice", "real"],
        default="simulation",
        help="simulation is safe; nice changes process priority on Unix; real
writes Linux cgroups v2 values",
    )
    parser.add_argument("--cgroup-path", type=str, default=None,
help="Existing cgroups v2 path for Linux real mode")
    parser.add_argument("--target-pid", type=int, default=None, help="Target
process PID for nice/renice priority actions")
    parser.add_argument("--results-dir", type=str, default="results",
help="Directory for CSV/XLSX, plots, Q-table and summary")
    parser.add_argument("--epsilon", type=float, default=0.25, help="Q-
learning exploration rate")
    parser.add_argument("--alpha", type=float, default=0.20, help="Q-learning
learning rate")
    parser.add_argument("--gamma", type=float, default=0.85, help="Q-learning
discount factor")
    parser.add_argument("--seed", type=int, default=42, help="Random seed for
reproducible action selection")
    parser.add_argument("--load-q-table", action="store_true", help="Continue
learning from results/q_table.json if it exists")
    args = parser.parse_args()

    results_dir = Path(args.results_dir)
    q_path = results_dir / "q_table.json"

    collector = MetricsCollector(cgroup_path=args.cgroup_path)

```

```

        backend = create_backend(args.mode, cgroup_path=args.cgroup_path,
target_pid=args.target_pid)
        agent = QLearningAgent(alpha=args.alpha, gamma=args.gamma,
epsilon=args.epsilon, seed=args.seed)
        if args.load_q_table:
            agent.load(q_path)

        controller = AdaptiveController(collector=collector, backend=backend,
agent=agent, interval=args.interval)

        print("Adaptive Linux RL Controller")
        print(f"Mode: {args.mode}")
        print(f"Duration: {args.duration}s; interval: {args.interval}s")
        print(f"Q-learning: alpha={args.alpha}, gamma={args.gamma},
epsilon={args.epsilon}, seed={args.seed}")
        if args.target_pid is not None:
            print(f"Target PID for nice/renice actions: {args.target_pid}")
        if args.mode == "simulation":
            print("Important: simulation mode does not modify the operating
system.")
        elif args.mode == "nice":
            print("Important: nice mode modifies only the selected process nice
value when priority actions are selected.")
        else:
            print("Important: real mode requires Linux cgroups v2 permissions;
nice actions need --target-pid.")
        print()

        rows = controller.run(duration=args.duration)
        exported = export_results(rows, results_dir)
        plots = create_plots(rows, results_dir)
        agent.save(q_path)

        print("\nExperiment finished.")
        print(f"CSV: {exported['csv']}")
        print(f"XLSX: {exported['xlsx']}")
        print(f"Summary: {exported['summary']}")
        print(f"Q-table: {q_path}")
        for name, path in plots.items():
            print(f"Plot {name}: {path}")

if __name__ == "__main__":
    main()

```

Лістинг A.2 – Реалізація Q-learning агента

```

from __future__ import annotations

import json
import random
from pathlib import Path
from typing import Dict, Iterable, Tuple

```

```

class QLearningAgent:
    """Tabular Q-learning agent for adaptive user-space CPU control
    decisions."""

    ACTIONS = (
        "noop",
        "increase_weight",
        "decrease_weight",
        "increase_quota",
        "decrease_quota",
        "increase_priority",
        "decrease_priority",
    )

    def __init__(
        self,
        alpha: float = 0.20,
        gamma: float = 0.85,
        epsilon: float = 0.25,
        actions: Iterable[str] | None = None,
        seed: int | None = 42,
    ) -> None:
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.actions = tuple(actions or self.ACTIONS)
        self.q_table: Dict[Tuple[str, str], float] = {}
        self.random = random.Random(seed)

    def choose_action(self, state_key: str) -> str:
        """
        Select an action using epsilon-greedy policy.

        Exploration: choose a random action with probability epsilon.
        Exploitation: choose one of the best-known actions. If several
actions
        have the same Q-value, choose among them randomly to avoid a
systematic
        bias toward the first action in the ACTIONS tuple.
        """
        if self.random.random() < self.epsilon:
            return self.random.choice(self.actions)

        values = {action: self.q_table.get((state_key, action), 0.0) for
action in self.actions}
        best_value = max(values.values())
        best_actions = [action for action, value in values.items() if value
== best_value]
        return self.random.choice(best_actions)

    def update(self, state_key: str, action: str, reward: float,
next_state_key: str) -> None:
        """Apply the tabular Q-learning update rule."""

```

```

        old_value = self.q_table.get((state_key, action), 0.0)
        next_best = max(self.q_table.get((next_state_key, next_action), 0.0)
for next_action in self.actions)
        new_value = old_value + self.alpha * (reward + self.gamma * next_best
- old_value)
        self.q_table[(state_key, action)] = round(new_value, 6)

def load(self, path: str | Path) -> None:
    """Load Q-values saved by save(). Missing files are ignored."""
    q_path = Path(path)
    if not q_path.exists():
        return

    loaded = json.loads(q_path.read_text(encoding="utf-8"))
    table: Dict[Tuple[str, str], float] = {}
    for key, value in loaded.items():
        if "::" not in key:
            continue
        state, action = key.rsplit("::", 1)
        if action in self.actions:
            table[(state, action)] = float(value)
    self.q_table = table

def save(self, path: str | Path) -> None:
    q_path = Path(path)
    q_path.parent.mkdir(parents=True, exist_ok=True)
    serializable = {f"{state}::{action}": value for (state, action),
value in sorted(self.q_table.items())}
    q_path.write_text(json.dumps(serializable, indent=2,
ensure_ascii=False), encoding="utf-8")

```

Лістинг А.3 – Реалізація адаптивного контролера

```

from __future__ import annotations

import time
from typing import Dict, List

from agent.q_learning import QLearningAgent
from controller.backends import BaseBackend
from environment.reward import calculate_reward
from environment.state import discretize_metrics
from monitoring.metrics import MetricsCollector

class AdaptiveController:
    """Coordinates metrics collection, Q-learning decisions, backend actions
and logs."""

    def __init__(self, collector: MetricsCollector, backend: BaseBackend,
agent: QLearningAgent, interval: float = 1.0) -> None:
        self.collector = collector
        self.backend = backend
        self.agent = agent
        self.interval = interval

    def run(self, duration: int) -> List[Dict[str, object]]:
        rows: List[Dict[str, object]] = []

```

```

start = time.time()
previous_metrics = self.collector.collect()
previous_throttling = previous_metrics.throttled_usec
step = 0

try:
    while time.time() - start < duration:
        step += 1
        metrics_before = self.collector.collect()
        state = discretize_metrics(metrics_before,
previous_throttling)
        action = self.agent.choose_action(state.key)
        control = self.backend.apply(action)

        time.sleep(self.interval)

        metrics_after = self.collector.collect()
        next_state = discretize_metrics(metrics_after,
metrics_before.throttled_usec)
        reward = calculate_reward(metrics_before, metrics_after)
        self.agent.update(state.key, action, reward, next_state.key)

        elapsed = round(time.time() - start, 2)
        rows.append(
            {
                "step": step,
                "time_seconds": elapsed,
                "cpu_level": state.cpu_level,
                "load_level": state.load_level,
                "throttling_level": state.throttling_level,
                "state_key": state.key,
                "action": action,
                "reward": reward,
                "cpu_percent": round(metrics_after.cpu_percent, 2),
                "load_avg_1m": round(metrics_after.load_avg_1m, 3),
                "process_count": metrics_after.process_count,
                "throttled_usec": metrics_after.throttled_usec,
                "backend_mode": control.mode,
                "action_applied": control.applied,
                "cpu_weight": control.cpu_weight,
                "cpu_quota": control.cpu_quota,
                "cpu_period": control.cpu_period,
                "nice_delta": control.nice_delta,
                "target_pid": control.target_pid,
                "current_nice": control.current_nice,
                "note": control.note,
            }
        )

        print(
            f"[{elapsed:>6.2f}s]                                state={state.key:<28}
action={action:<18} "
            f"cpu={metrics_after.cpu_percent:>5.1f}%"
reward={reward:>7.3f} "
            f"applied={str(control.applied):<5} mode={control.mode}"
        )
        previous_metrics = metrics_after
        previous_throttling = previous_metrics.throttled_usec
except KeyboardInterrupt:
    print("\nExperiment interrupted by user. Saving partial
results...")

```

```
return rows
```

Лістинг A.4 – Формування дискретного стану системи

```
from __future__ import annotations

from dataclasses import dataclass

from monitoring.metrics import SystemMetrics

@dataclass(frozen=True)
class DiscreteState:
    cpu_level: str
    load_level: str
    throttling_level: str

    @property
    def key(self) -> str:
        return f"{self.cpu_level}|{self.load_level}|{self.throttling_level}"

def discretize_metrics(metrics: SystemMetrics, previous_throttled_usec: int = 0) -> DiscreteState:
    if metrics.cpu_percent < 35:
        cpu_level = "low"
    elif metrics.cpu_percent < 75:
        cpu_level = "medium"
    else:
        cpu_level = "high"

    if metrics.load_avg_1m < 1.0:
        load_level = "normal"
    elif metrics.load_avg_1m < 3.0:
        load_level = "busy"
    else:
        load_level = "overloaded"

    throttling_level = "throttled" if metrics.throttled_usec >
previous_throttled_usec else "not_throttled"
    return DiscreteState(cpu_level=cpu_level, load_level=load_level,
throttling_level=throttling_level)
```

Лістинг A.5 – Реалізація функції винагороди

```
from __future__ import annotations
from monitoring.metrics import SystemMetrics
def calculate_reward(previous: SystemMetrics, current: SystemMetrics) -> float:
    """
    Simple reward for a bachelor-level prototype.

    The controller is rewarded for keeping CPU utilization in a stable
    working range
    and penalized for overload, high load average, and cgroup throttling
    growth.
    """
    stable_cpu_bonus = 0.30 if 35.0 <= current.cpu_percent <= 75.0 else 0.0
    overload_penalty = max(0.0, current.cpu_percent - 75.0) / 25.0
    idle_penalty = max(0.0, 20.0 - current.cpu_percent) / 100.0
    load_penalty = max(0.0, current.load_avg_1m - 2.0) / 4.0
    throttling_penalty = 1.0 if current.throttled_usec >
previous.throttled_usec else 0.0
    return round(stable_cpu_bonus - overload_penalty - idle_penalty -
load_penalty - throttling_penalty, 4)
```

Лістинг A.6 – Збір системних метрик

```

from __future__ import annotations

import os
import platform
from dataclasses import dataclass
from pathlib import Path
from typing import Optional

import psutil

@dataclass(frozen=True)
class SystemMetrics:
    """User-space snapshot of CPU-related system state."""

    timestamp: float
    cpu_percent: float
    load_avg_1m: float
    process_count: int
    throttled_usec: int = 0
    platform_name: str = platform.system()

class MetricsCollector:
    """Collects CPU metrics from psutil and optionally from cgroups v2
    cpu.stat."""

    def __init__(self, cgroup_path: Optional[str] = None) -> None:
        self.cgroup_path = Path(cgroup_path).expanduser() if cgroup_path else
None
        psutil.cpu_percent(interval=None) # prime psutil's CPU measurement

    def collect(self) -> SystemMetrics:
        load_avg = os.getloadavg()[0] if hasattr(os, "getloadavg") else 0.0
        return SystemMetrics(
            timestamp=psutil.time.time(),
            cpu_percent=float(psutil.cpu_percent(interval=None)),
            load_avg_1m=float(load_avg),
            process_count=len(psutil.pids()),
            throttled_usec=self._read_throttled_usec(),
        )

    def _read_throttled_usec(self) -> int:
        if not self.cgroup_path:
            return 0
        stat_file = self.cgroup_path / "cpu.stat"
        if not stat_file.exists():
            return 0
        try:
            for line in stat_file.read_text(encoding="utf-8").splitlines():
                key, _, value = line.partition(" ")
                if key == "throttled_usec":
                    return int(value.strip())
        except (OSError, ValueError):
            return 0
        return 0

```

ДОДАТОК Б МОДУЛІ ВИКОНАННЯ КЕРУЮЧИХ ДІЙ ТА ОБРОБЛЕННЯ РЕЗУЛЬТАТІВ

Лістинг Б.1 – Реалізація backend-механізмів керуючих дій

```

from future import annotations

import os
import platform
from dataclasses import dataclass
from pathlib import Path
from typing import Optional

@dataclass
class ControlSnapshot:
    mode: str
    cpu_weight: int
    cpu_quota: int
    cpu_period: int
    nice_delta: int
    applied: bool
    note: str = ""
    target_pid: Optional[int] = None
    current_nice: Optional[int] = None

class BaseBackend:
    def apply(self, action: str) -> ControlSnapshot: # pragma: no cover -
interface
    raise NotImplementedError

class SimulationBackend(BaseBackend):
    """Safe backend for macOS and demonstrations. It simulates control values
    only."""

    def __init__(self, target_pid: Optional[int] = None) -> None:
        self.cpu_weight = 100
        self.cpu_quota = 100_000
        self.cpu_period = 100_000
        self.nice_delta = 0
        self.target_pid = target_pid
        self.current_nice: Optional[int] = None

```

```

def apply(self, action: str) -> ControlSnapshot:
    self._update_internal_values(action)
    return ControlSnapshot(
        mode="simulation",
        cpu_weight=self.cpu_weight,
        cpu_quota=self.cpu_quota,
        cpu_period=self.cpu_period,
        nice_delta=self.nice_delta,
        applied=False,
        note="dry-run: no system values were changed",
        target_pid=self.target_pid,
        current_nice=self.current_nice,
    )

def _update_internal_values(self, action: str) -> None:
    if action == "increase_weight":
        self.cpu_weight = min(10_000, int(self.cpu_weight * 1.10))
    elif action == "decrease_weight":
        self.cpu_weight = max(1, int(self.cpu_weight * 0.90))
    elif action == "increase_quota":
        self.cpu_quota = min(400_000, int(self.cpu_quota * 1.10))
    elif action == "decrease_quota":
        self.cpu_quota = max(10_000, int(self.cpu_quota * 0.90))
    elif action == "increase_priority":
        self.nice_delta = max(-10, self.nice_delta - 1)
    elif action == "decrease_priority":
        self.nice_delta = min(10, self.nice_delta + 1)

class NiceReniceBackend(SimulationBackend):
    """
    Portable Unix backend that applies nice/renice-like priority changes.

    It uses os.getpriority/os.setpriority for a target PID. Lower nice values
    mean higher priority and usually require elevated permissions. Higher
    nice
    values reduce priority and are normally allowed for processes owned by
    the
    current user.
    """

    def __init__(self, target_pid: int) -> None:

```

```

    super().__init__(target_pid=target_pid)
    if not hasattr(os, "getpriority") or not hasattr(os, "setpriority"):
        raise RuntimeError("nice/renice backend requires a Unix-like OS
with getpriority/setpriority support.")
    self._read_current_nice()

def apply(self, action: str) -> ControlSnapshot:
    self._update_internal_values(action)
    notes = []
    applied = False

    if action in {"increase_priority", "decrease_priority"}:
        applied, note = self._apply_nice_delta(action)
        notes.append(note)
    elif action == "noop":
        notes.append("no operation")
    else:
        notes.append("action is simulated in nice mode; only priority
actions modify the OS")

    return ControlSnapshot(
        mode="nice-renice",
        cpu_weight=self.cpu_weight,
        cpu_quota=self.cpu_quota,
        cpu_period=self.cpu_period,
        nice_delta=self.nice_delta,
        applied=applied,
        note="; ".join(notes),
        target_pid=self.target_pid,
        current_nice=self.current_nice,
    )

def _read_current_nice(self) -> Optional[int]:
    if self.target_pid is None:
        return None
    try:
        self.current_nice = int(os.getpriority(os.PRIO_PROCESS,
self.target_pid))
    except OSError:
        self.current_nice = None
    return self.current_nice

```

```

def _apply_nice_delta(self, action: str) -> tuple[bool, str]:
    if self.target_pid is None:
        return False, "target PID is not provided"

    before = self._read_current_nice()
    if before is None:
        return False, f"could not read nice value for PID
{self.target_pid}"

    if action == "increase_priority":
        desired = max(-20, before - 1)
    else:
        desired = min(19, before + 1)

    if desired == before:
        return False, f"nice value already at boundary ({before})"

    try:
        os.setpriority(os.PRIO_PROCESS, self.target_pid, desired)
        after = self._read_current_nice()
        return True, f"nice value changed for PID {self.target_pid}:
{before} -> {after}"
    except PermissionError:
        self.current_nice = before
        return False, f"permission denied while changing nice value for
PID {self.target_pid}: {before} -> {desired}"
    except ProcessLookupError:
        self.current_nice = None
        return False, f"process not found: PID {self.target_pid}"
    except OSError as exc:
        self.current_nice = before
        return False, f"could not change nice value for PID
{self.target_pid}: {exc}"

class LinuxCgroupsV2Backend(SimulationBackend):
    """Linux backend that writes cpu.weight and cpu.max to an existing
cgroups v2 directory."""

    def __init__(self, cgroup_path: str, target_pid: Optional[int] = None) ->
None:
        super().__init__(target_pid=target_pid)
        self.cgroup_path = Path(cgroup_path).expanduser()

```

```

        self.nice_backend = NiceReniceBackend(target_pid) if target_pid is
not None else None
        if platform.system() != "Linux":
            raise RuntimeError("Real cgroups mode is available only on
Linux.")
        if not self.cgroup_path.exists():
            raise FileNotFoundError(f"cgroup path does not exist:
{self.cgroup_path}")

    def apply(self, action: str) -> ControlSnapshot:
        self._update_internal_values(action)
        weight_file = self.cgroup_path / "cpu.weight"
        max_file = self.cgroup_path / "cpu.max"
        notes = []
        applied = False

        if action in {"increase_weight", "decrease_weight"}:
            if weight_file.exists() and os.access(weight_file, os.W_OK):
                weight_file.write_text(str(self.cpu_weight), encoding="utf-
8")

                applied = True
                notes.append("cpu.weight updated")
            else:
                notes.append("cpu.weight is unavailable or not writable")

        elif action in {"increase_quota", "decrease_quota"}:
            if max_file.exists() and os.access(max_file, os.W_OK):
                max_file.write_text(f"{self.cpu_quota} {self.cpu_period}",
encoding="utf-8")

                applied = True
                notes.append("cpu.max updated")
            else:
                notes.append("cpu.max is unavailable or not writable")

        elif action in {"increase_priority", "decrease_priority"}:
            if self.nice_backend is None:
                notes.append("nice/renice skipped: --target-pid was not
provided")
            else:
                nice_snapshot = self.nice_backend.apply(action)
                self.current_nice = nice_snapshot.current_nice
                applied = nice_snapshot.applied

```

```

        notes.append(nice_snapshot.note)

    else:
        notes.append("no operation")

    if self.nice_backend is not None and action not in
{"increase_priority", "decrease_priority"}:
        self.current_nice = self.nice_backend._read_current_nice()

    return ControlSnapshot(
        mode="linux-cgroups-v2",
        cpu_weight=self.cpu_weight,
        cpu_quota=self.cpu_quota,
        cpu_period=self.cpu_period,
        nice_delta=self.nice_delta,
        applied=applied,
        note="; ".join(notes),
        target_pid=self.target_pid,
        current_nice=self.current_nice,
    )

def create_backend(mode: str, cgroup_path: Optional[str] = None, target_pid:
Optional[int] = None) -> BaseBackend:
    if mode == "real":
        if not cgroup_path:
            raise ValueError("--cgroup-path is required for --mode real")
        return LinuxCgroupsV2Backend(cgroup_path=cgroup_path,
target_pid=target_pid)
    if mode == "nice":
        if target_pid is None:
            raise ValueError("--target-pid is required for --mode nice")
        return NiceReniceBackend(target_pid=target_pid)
    return SimulationBackend(target_pid=target_pid)

```

Лістинг Б.2 – Експорт результатів експерименту

```

from __future__ import annotations

from collections import Counter
from pathlib import Path
from statistics import mean
from typing import Dict, List

```

```
import pandas as pd
```

```
EXPORT_COLUMNS = [
    "step",
    "time_seconds",
    "cpu_level",
    "load_level",
    "throttling_level",
    "state_key",
    "action",
    "reward",
    "cpu_percent",
    "load_avg_1m",
    "process_count",
    "throttled_usec",
    "backend_mode",
    "action_applied",
    "cpu_weight",
    "cpu_quota",
    "cpu_period",
    "nice_delta",
    "target_pid",
    "current_nice",
    "note",
]
```

```
def export_results(rows: List[Dict[str, object]], results_dir: str | Path) ->
Dict[str, Path]:
    results_path = Path(results_dir)
    results_path.mkdir(parents=True, exist_ok=True)

    df = pd.DataFrame(rows)
    if df.empty:
        df = pd.DataFrame(columns=EXPORT_COLUMNS)
    else:
        for column in EXPORT_COLUMNS:
            if column not in df.columns:
                df[column] = None
        df = df[EXPORT_COLUMNS]
```

```

csv_path = results_path / "experiment_results.csv"
xlsx_path = results_path / "experiment_results.xlsx"
summary_path = results_path / "summary.txt"

df.to_csv(csv_path, index=False, encoding="utf-8", float_format="%.6f")

with pd.ExcelWriter(xlsx_path, engine="openpyxl") as writer:
    df.to_excel(writer, sheet_name="experiment_log", index=False)
    summary_df = _build_summary_frame(df)
    summary_df.to_excel(writer, sheet_name="summary", index=False)

    workbook = writer.book
    for worksheet in workbook.worksheets:
        worksheet.freeze_panes = "A2"
        for column_cells in worksheet.columns:
            max_len = max(len(str(cell.value)) if cell.value is not None
else 0 for cell in column_cells)

        worksheet.column_dimensions[column_cells[0].column_letter].width =
min(max(max_len + 2, 12), 42)

    summary_path.write_text(_build_summary_text(df), encoding="utf-8")
    return {"csv": csv_path, "xlsx": xlsx_path, "summary": summary_path}

def _build_summary_frame(df: pd.DataFrame) -> pd.DataFrame:
    metrics = []
    if not df.empty:
        metrics.extend(
            [
                ("steps", int(len(df))),
                ("avg_cpu_percent", round(float(df["cpu_percent"].mean()),
3)),
                ("max_cpu_percent", round(float(df["cpu_percent"].max()),
3)),
                ("avg_reward", round(float(df["reward"].mean()), 4)),
                ("best_reward", round(float(df["reward"].max()), 4)),
                ("worst_reward", round(float(df["reward"].min()), 4)),
                ("most_common_action",
str(Counter(df["action"]).most_common(1)[0][0])),
                ("backend_mode", str(df["backend_mode"].iloc[-1]) if
"backend_mode" in df else "unknown"),

```

```

        ("applied_actions", int(df["action_applied"].sum()) if
"action_applied" in df else 0),
    ]
)
return pd.DataFrame(metrics, columns=["metric", "value"])

def _build_summary_text(df: pd.DataFrame) -> str:
    if df.empty:
        return "No experiment rows were collected.\n"
    actions = Counter(df["action"]).most_common()
    action_lines = "\n".join(f"- {action}: {count}" for action, count in
actions)
    applied_actions = int(df["action_applied"].sum()) if "action_applied" in
df else 0
    backend_mode = df["backend_mode"].iloc[-1]

    return (
        "Adaptive Linux RL Controller - experiment summary\n"
        "=====\n"
        f"Steps: {len(df)}\n"
        f"Average CPU usage: {mean(df['cpu_percent']):.2f}%\n"
        f"Maximum CPU usage: {max(df['cpu_percent']):.2f}%\n"
        f"Average reward: {mean(df['reward']):.4f}\n"
        f"Best reward: {max(df['reward']):.4f}\n"
        f"Worst reward: {min(df['reward']):.4f}\n"
        f"Backend mode: {backend_mode}\n"
        f"Actions applied to OS: {applied_actions}\n\n"
        "Action distribution:\n"
        f"{action_lines}\n\n"
        "Interpretation:\n"
    )

```

Лістинг Б.3 – Побудова графіків експерименту

```

from __future__ import annotations

from pathlib import Path
from typing import Dict, List

import matplotlib.pyplot as plt
import pandas as pd

```

```

def create_plots(rows: List[Dict[str, object]], results_dir: str | Path) ->
Dict[str, Path]:
    results_path = Path(results_dir)
    results_path.mkdir(parents=True, exist_ok=True)
    df = pd.DataFrame(rows)
    if df.empty:
        return {}

    cpu_path = results_path / "cpu_usage.png"
    reward_path = results_path / "reward.png"
    actions_path = results_path / "actions.png"

    plt.figure(figsize=(9, 4.8))
    plt.plot(df["time_seconds"], df["cpu_percent"], marker="o",
linewidth=1.5)
    plt.xlabel("Time, seconds")
    plt.ylabel("CPU usage, %")
    plt.title("Adaptive Controller: CPU Usage")
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig(cpu_path, dpi=160)
    plt.close()

    plt.figure(figsize=(9, 4.8))
    plt.plot(df["time_seconds"], df["reward"], marker="o", linewidth=1.5)
    plt.xlabel("Time, seconds")
    plt.ylabel("Reward")
    plt.title("Q-learning Reward Dynamics")
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig(reward_path, dpi=160)
    plt.close()

    action_counts = df["action"].value_counts().sort_values(ascending=True)
    plt.figure(figsize=(9, 4.8))
    plt.barh(action_counts.index, action_counts.values)
    plt.xlabel("Number of selected actions")
    plt.ylabel("Action")
    plt.title("Action Distribution")
    plt.tight_layout()

```

```

plt.savefig(actions_path, dpi=160)
plt.close()

return {"cpu_usage": cpu_path, "reward": reward_path, "actions":
actions_path}

```

Лістинг Б.4 – Генератор CPU-навантаження

```

from __future__ import annotations

import argparse
import multiprocessing as mp
from pathlib import Path
import time

def burn_cpu(seconds: int) -> None:
    end = time.time() + seconds
    value = 1
    while time.time() < end:
        value = (value * 13 + 7) % 1_000_003

def main() -> None:
    parser = argparse.ArgumentParser(description="Portable CPU load generator
for controller experiments.")
    parser.add_argument("--workers", type=int, default=2, help="Number of
CPU-bound worker processes")
    parser.add_argument("--duration", type=int, default=60, help="Load
duration in seconds")
    parser.add_argument("--pid-file", type=str, default=None, help="Optional
file where worker PIDs will be saved")
    args = parser.parse_args()

    processes = [mp.Process(target=burn_cpu, args=(args.duration,)) for _ in
range(args.workers)]
    for process in processes:
        process.start()

    pids = [process.pid for process in processes if process.pid is not None]
    print("Load generator started.")
    print(f"Worker PIDs: {' ', ' '.join(str(pid) for pid in pids)}")

```

```
print("Use one of these PIDs with: python3 main.py --mode nice --target-  
pid <PID>")  
  
if args.pid_file:  
    pid_path = Path(args.pid_file)  
    pid_path.parent.mkdir(parents=True, exist_ok=True)  
    pid_path.write_text("\n".join(str(pid) for pid in pids) + "\n",  
encoding="utf-8")  
    print(f"PIDs saved to: {pid_path}")  
  
for process in processes:  
    process.join()  
  
if __name__ == "__main__":  
    main()
```

Лістинг Б.5 – Файл залежностей проекту

```
psutil>=5.9.0  
pandas>=2.0.0  
openpyxl>=3.1.0  
matplotlib>=3.7.0
```