МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Міжнародний гуманітарний університет
Кафедра комп’ютерної інженерії та інноваційних технологій

Педяш В.В., Йона Л.Г., Мазур Г.Д.

ОПЕРАЦІЙНІ СИСТЕМИ

Конспект лекцій для здобувачів вищої освіти зі спеціальностей:
A4.09 Середня освіта (Інформатика),
F2 Інженерія програмного забезпечення,
F3 Комп'ютерні науки,
F7 Комп'ютерна інженерія,
F5 Кібербезпека та захист інформації,
G5 Електроніка, електронні комунікації, приладобудування та радіотехніка

Одеса 2025

Педяш В.В., Йона Л.Г., Мазур Г.Д. Операційні системи: Конспект лекцій [Електронне видання] / Йона Л.Г., Педяш В.В., Мазур Г.Д. — Кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. — Одеса, 2025. - 116 с.

Конспект лекцій з дисципліни «Операційні системи» розроблено відповідно до навчального плану підготовки здобувачів вищої освіти. Посібник охоплює сім тематичних модулів, спрямованих на освоєння теоретичних основ архітектури, функціонування та безпеки сучасних операційних систем. Матеріали містять систематизований виклад від класифікації та архітектурних підходів до побудови ОС (монолітна, мікроядерна, гібридна, модульна архітектури) до механізмів управління процесами та потоками, організації пам'яті (сегментація, сторінкування, віртуальна пам'ять), файлових систем (FAT32, NTFS, ext4, ZFS, APFS) та засобів контролю доступу (DAC, MAC, RBAC). Окремі розділи присвячені автоматизації системних завдань (скрипти, Ansible, Puppet, планування через cron/systemd), а також технологіям віртуалізації та контейнеризації (гіпервізори, Docker, Kubernetes) у контексті забезпечення безпеки та надійності IT-інфраструктури.

Посібник призначено для здобувачів першого (бакалаврського) рівня вищої освіти спеціальностей: А4.09 Середня освіта (Інформатика), F2 Інженерія програмного забезпечення, F3 Комп'ютерні науки, F7 Комп'ютерна інженерія, F5 Кібербезпека та захист інформації, G5 Електроніка, електронні комунікації, приладобудування та радіотехніка.

Матеріал може використовуватися як для аудиторної роботи під час лекцій, так і для самостійного опрацювання студентами. Теоретичні знання, отримані з посібника, можуть бути закріплені під час виконання практичних завдань у віртуальних середовищах (VirtualBox, VMware, KVM), контейнерних платформах (Docker, Podman), оркестраторах (Kubernetes) або на реальному обладнанні, доступному в лабораторіях факультету кібербезпеки, програмної інженерії та комп'ютерних наук. Такий підхід забезпечує фундаментальну теоретичну підготовку та сприяє формуванню практичних інженерних навичок, необхідних для подальшого вивчення сучасних системних технологій, адміністрування ОС сімейств Linux/Windows, проходження профільних сертифікацій (зокрема, Linux Professional Institute, Red Hat) та виконання кваліфікаційних робіт у галузі інформаційно-комунікаційних технологій.

ЗМІСТ

ВСТУП

У сучасному цифровому світі операційні системи виступають фундаментальною основою функціонування будь-якої інформаційно-комунікаційної інфраструктури. Від персональних комп'ютерів та мобільних пристроїв до потужних серверних кластерів і хмарних платформ — саме операційна система забезпечує ефективне управління апаратними ресурсами, ізоляцію процесів, безпеку даних та взаємодію між користувачем і обчислювальною технікою. Стрімкий розвиток технологій віртуалізації, контейнеризації, автоматизації конфігурації та кібербезпеки зумовлює критичну потребу у фахівцях, які глибоко розуміють архітектуру, принципи роботи та механізми захисту сучасних операційних середовищ. Якісне опанування дисципліни «Операційні системи» стає необхідною умовою для підготовки конкурентоспроможних інженерів, здатних проектувати, адмініструвати та захищати складні IT-системи в умовах зростаючих кіберзагроз.

Даний конспект лекцій розроблено кафедрою комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету відповідно до навчального плану підготовки здобувачів вищої освіти першого (бакалаврського) рівня. Посібник містить систематизований виклад матеріалу, охоплюючи сім ключових тематичних модулів, що забезпечують комплексне вивчення дисципліни. Матеріал лекцій послідовно розкриває архітектуру та класифікацію сучасних операційних систем, механізми управління процесами та потоками виконання, принципи організації віртуальної пам'яті та захисту даних. Значна увага приділяється файловим системам та організацію зберігання, моделям безпеки та контролю доступу (DAC, MAC, RBAC), а також сучасним практикам автоматизації системних завдань та управління конфігурацією. Завершує курс тема щодо віртуалізації, контейнеризації та управління сучасними IT-середовищами, що відображає актуальні тренди галузі.

Метою використання цього посібника є забезпечення фундаментальної теоретичної підготовки та формування практичних інженерних навичок у здобувачів вищої освіти. Матеріал призначений як для аудиторної роботи під час лекцій, так і для самостійного опрацювання. Теоретичні знання, отримані з посібника, рекомендовано закріплювати під час виконання практичних завдань у віртуальних середовищах (VirtualBox, KVM), контейнерних платформах (Docker, Kubernetes) та системах автоматизації (Ansible, Terraform). Такий підхід сприяє підготовці студентів до проходження профільних сертифікацій (зокрема Linux Professional Institute, Red Hat), виконання кваліфікаційних робіт та успішної професійної діяльності у галузі інформаційно-комунікаційних технологій.

Автори сподіваються, що цей конспект лекцій стане надійним помічником у вивченні операційних систем та допоможе здобувачам сформувати цілісне уявлення про принципи побудови та експлуатації сучасних програмних платформ..

Тема 1. Архітектура та класифікація сучасних операційних систем

1.1 Операційна система як системне програмне забезпечення

Операційна система (ОС) є фундаментальним компонентом будь-якого обчислювального пристрою, без якого апаратне забезпечення залишається лише набором електронних компонентів, нездатних виконувати корисну роботу. З точки зору функціонального призначення, ОС виступає посередником між апаратними ресурсами комп'ютера та прикладним програмним забезпеченням, надаючи програмам стандартизований інтерфейс для доступу до пристроїв вводу-виводу, пам'яті, процесора та мережевих засобів. Без цього рівня абстракції кожен розробник прикладного програмного забезпечення мусив би самостійно реалізовувати підтримку всіх можливих апаратних конфігурацій, що зробило б розробку програм практично неможливою в промисловому масштабі. Операційна система вирішує цю проблему, приховуючи складність апаратного рівня за уніфікованими програмними інтерфейсами — системними викликами (system calls). Завдяки цьому розробник текстового редактора або браузера може звернутись до файлової системи одним і тим самим викликом open() незалежно від того, чи це жорсткий диск, твердотільний накопичувач чи мережева файлова система.

Класичне визначення операційної системи включає два ключових аспекти її природи. По-перше, це менеджер ресурсів: ОС управляє процесором, оперативною пам'яттю, пристроями зберігання даних і периферійним обладнанням, гарантуючи ефективне та безпечне використання цих ресурсів між конкуруючими програмами. По-друге, це розширена машина (extended machine): ОС надає прикладним програмам зручну та потужну абстракцію обчислювального середовища — файлову систему замість фізичних секторів диска, мережеві сокети замість сигналів на дротах, процеси замість безпосередніх інструкцій процесора. Обидва аспекти однаково важливі і взаємодоповнюють одне одного у формуванні цілісного системного програмного середовища, де апаратна складність залишається прихованою від розробників прикладних програм.

Для розуміння ролі ОС в інформаційно-комунікаційних системах (ІКС) важливо усвідомити ієрархію програмного забезпечення. Знизу розташоване апаратне забезпечення — процесор, пам'ять, шини, контролери пристроїв. Над ним — ядро операційної системи, яке безпосередньо взаємодіє з апаратурою через драйвери пристроїв. Далі — системні утиліти та бібліотеки (libc, Win32 API тощо), що надають сервіси вищого рівня і спрощують написання прикладних програм. Нарешті, на верхньому рівні знаходяться прикладні програми, з якими безпосередньо взаємодіє користувач. Чіткий розподіл рівнів цієї ієрархії є не лише питанням зручності, але й критичним фактором безпеки: кожен рівень повинен мати чітко визначені привілеї та обмеження доступу до нижчих рівнів, а будь-яке порушення цих меж розглядається як потенційна уразливість.

Рисунок 1.1 відображає ієрархічну модель рівнів програмного забезпечення обчислювальної системи. На рисунку зображено п’ять горизонтальних шарів, розташованих один над одним у вигляді прямокутників рівної ширини.

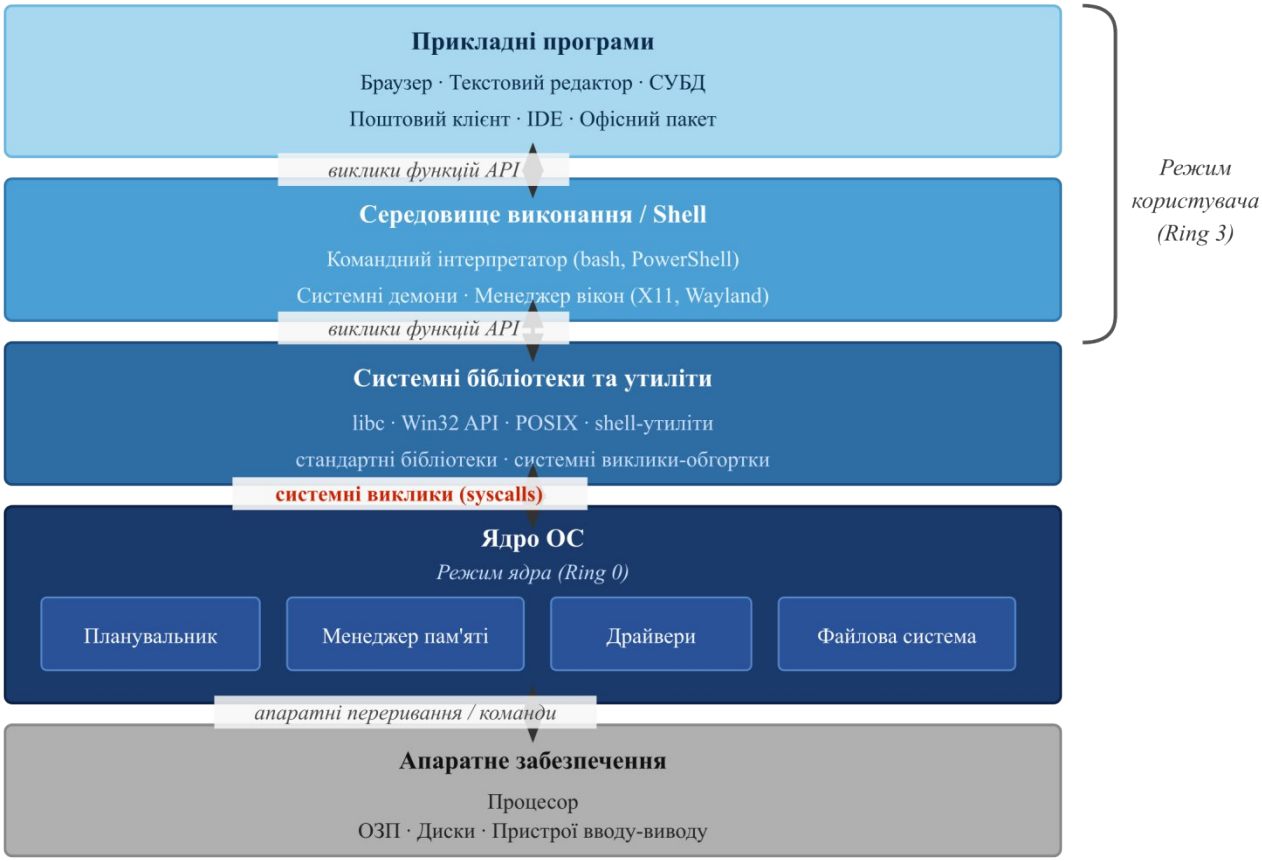

Рисунок 1.1 – Ієрархічна модель рівнів програмного забезпечення обчислювальної системи.

Розвиток операційних систем нерозривно пов’язаний з еволюцією апаратного забезпечення та вимог до програмних систем. Перші ОС 1950–60-х років були монопрограмними — вони виконували лише одну задачу за раз, а після її завершення оператор вручну завантажував наступну програму, нерідко фізично змінюючи перфокарти. Введення пакетної обробки дозволило автоматизувати чергу задач: спеціальна програма-монітор послідовно завантажувала задачі з черги, значно підвищивши ефективність використання дорогого процесорного часу. Поява мультипрограмування в 1960-х роках стала справжньою революцією: кілька програм могли одночасно знаходитись в пам’яті та почергово використовувати процесор, а процесор не простоював під час очікування повільних операцій вводу-виводу. Подальшим розвитком стали системи поділу часу (time-sharing), де кожен з численних термінальних користувачів отримував ілюзію монопольного доступу до комп’ютера — саме так у 1969 році в AT&T Bell Labs з’явилась система UNIX, архітектурні принципи якої досі визначають вигляд сучасних серверних та мобільних ОС.

Сучасні ОС є надзвичайно складними програмними системами. Ядро Linux 6.x налічує понад 30 мільйонів рядків вихідного коду від тисяч розробників по всьому світу. Така складність неминуче породжує питання архітектурного проектування: як організувати код ОС так, щоб він був надійним, безпечним, ефективним і підтримуваним протягом десятиліть? При цьому складність не є перевагою сама по собі — кожен зайвий рядок коду ядра є потенційним джерелом уразливості. Дослідження показують, що типова щільність помилок у програмному коді становить від 1 до 25 дефектів на 1000 рядків, тому навіть добре перевірений код ядра потенційно містить тисячі прихованих помилок. Саме це і стало рушієм розвитку різних архітектурних підходів до побудови ОС, кожен з яких намагається по-своєму вирішити суперечність між продуктивністю та ізоляцією компонентів.

У таблиці 1.1 представлено хронологію ключових етапів розвитку операційних систем та відповідні технологічні зрушення, що їх спричинили.

Таблиця 1.1 — Хронологія розвитку операційних систем

Період	Покоління ОС	Ключова концепція	Приклади
1950–1955	Без ОС	Ручне управління через пульт оператора	IBM 701
1955–1965	Пакетна обробка	Монітор-резидент, автозавантаження задач	FMS, IBJOB
1965–1980	Мультипрограмування / Time-sharing	Поділ часу, термінальний доступ	MULTICS, UNIX, IBM OS/360
1980–1990	Персональні ОС	Однокористувацькі, графічний інтерфейс	MS-DOS, CP/M, Apple DOS
1990–2000	Захищені багатозадачні ОС	Витісняльна багатозадачність, GUI	Windows NT, Linux 1.0, macOS
2000–2010	Мережеві та серверні ОС	SMP, кластеризація, веб-сервіси	RHEL, Windows Server 2003/2008
2010–сьогодні	Мобільні, хмарні, вбудовані	Контейнеризація, IoT, гіпервізори	Android, iOS, CoreOS, OpenWrt

1.2 Архітектурні підходи до побудови операційних систем

Вибір архітектури операційної системи є одним із найважливіших проектних рішень, що визначає не тільки продуктивність і функціональність системи, але й її надійність, безпеку та здатність до розвитку протягом десятиліть. Різні архітектурні парадигми сформувались у відповідь на конкретні технічні виклики та вимоги різних епох розвитку обчислювальної техніки, і кожна з них досі знаходить застосування у сучасних системах — жодна не є абсолютно «правильною» чи «застарілою».

Монолітна архітектура є найстарішою та найпоширенішою серед сучасних ОС. У монолітній ОС усі компоненти ядра — планувальник процесів, менеджер пам’яті, файлова система, драйвери пристроїв, мережевий стек — виконуються в єдиному адресному просторі з максимальними привілеями

(режим ядра, Ring 0 у термінах архітектури x86). Це дозволяє компонентам безпосередньо викликати функції один одного без накладних витрат на перемикання контексту чи міжпроцесну взаємодію, що забезпечує надзвичайно високу продуктивність — ключову перевагу цього підходу. Класичними прикладами монолітних ОС є UNIX і його нащадки, Linux, а також MS-DOS. Linux, незважаючи на свою монолітну природу, підтримує динамічне завантаження модулів ядра (LKM — Loadable Kernel Modules), що дещо пом'якшує жорсткість монолітної моделі та дає розробникам змогу додавати функціонал без перекомпіляції і перезавантаження системи. Головним недоліком монолітної архітектури є відсутність ізоляції між компонентами: помилка в будь-якому з них, наприклад у некоректно написаному драйвері стороннього виробника, може призвести до краху всієї системи, а успішна компрометація одного компонента ядра відкриває зловмиснику необмежений доступ до всіх ресурсів системи.

Мікроядерна архітектура виникла в 1980-х роках як теоретично обґрунтована відповідь на проблеми надійності та безпеки монолітних систем. Основна ідея полягає в тому, щоб звести ядро до абсолютного мінімуму — зазвичай це лише механізми міжпроцесної взаємодії (IPC), базове управління фізичною пам'яттю та примітивне планування потоків. Решта компонентів, що в монолітній ОС виконуються в режимі ядра, — драйвери пристроїв, файлові системи, мережеві протоколи — переносяться в режим користувача як окремі серверні процеси зі звичайними обмеженими правами доступу. Класичні приклади мікроядерних ОС — Mach (що ліг в основу macOS та iOS), QNX Neutrino (широко застосовується в промисловості та автомобілебудуванні), MINIX 3 (розроблений Ендрю Таненбаумом), а також формально верифіковане ядро seL4. Така архітектура забезпечує значно вищу надійність: збій драйвера не руйнує систему, оскільки він виконується як звичайний процес, який ОС може автоматично перезапустити. Проте ціною цих переваг є продуктивність: будь-яка взаємодія між компонентами вимагає дорогих IPC-операцій — переходу через кордон режимів і копіювання даних між адресними просторами, що може уповільнити систему в 5–10 разів порівняно з монолітним ядром для операцій з інтенсивним вводом-виводом.

Гібридна архітектура намагається знайти практичний баланс між перевагами обох підходів, розміщуючи критичні для продуктивності компоненти в режимі ядра, а менш критичні — у просторі користувача. Windows NT та всі наступні версії Windows (XP, 7, 10, 11) є найвідомішим прикладом гібридного ядра: менеджер вводу-виводу, кеш-менеджер, підсистема безпеки та навіть графічна підсистема Win32 виконуються безпосередньо в режимі ядра, хоча архітектурно система передбачала їх роботу у просторі користувача. macOS базується на гібридному ядрі XNU (X is Not Unix), що поєднує мікроядро Mach з компонентами BSD у режимі ядра, досягаючи балансу між перевагами архітектури Mach та продуктивністю прямих викликів. Гібридний підхід є прагматичним компромісом, що домінує у комерційних системах, оскільки дозволяє досягти прийнятної продуктивності при збереженні принаймні частини переваг мікроядерної ізоляції.

Модульна архітектура є не стільки самостійним типом, скільки важливим організаційним принципом, що може застосовуватись разом з будь-якою з перерахованих архітектур. Ідея полягає в тому, що функціонал ядра організовується у вигляді окремих незалежних модулів зі строго визначеними інтерфейсами, що можуть завантажуватись та вивантажуватись під час роботи системи без перезавантаження. Ядро Linux реалізує цей підхід в повній мірі: командою `modprobe btrfs` можна динамічно додати підтримку файлової системи Btrfs, `modprobe ipv6` — мережевого протоколу IPv6, `modprobe nvidia` — встановити драйвер відеокарти NVIDIA, і все це без перекомпіляції ядра та перезавантаження системи. Модульність суттєво зменшує базовий відбиток пам'яті ядра, оскільки завантажуються лише необхідні компоненти, та спрощує розробку і тестування окремих підсистем. Водночас вона вимагає ретельного проектування стабільних інтерфейсів: зміна внутрішнього ABI ядра Linux між версіями є однією з найпоширеніших причин несумісності сторонніх модулів.

Рисунок 1.2 ілюструє структурну відмінність між монолітною та мікроядерною архітектурами.

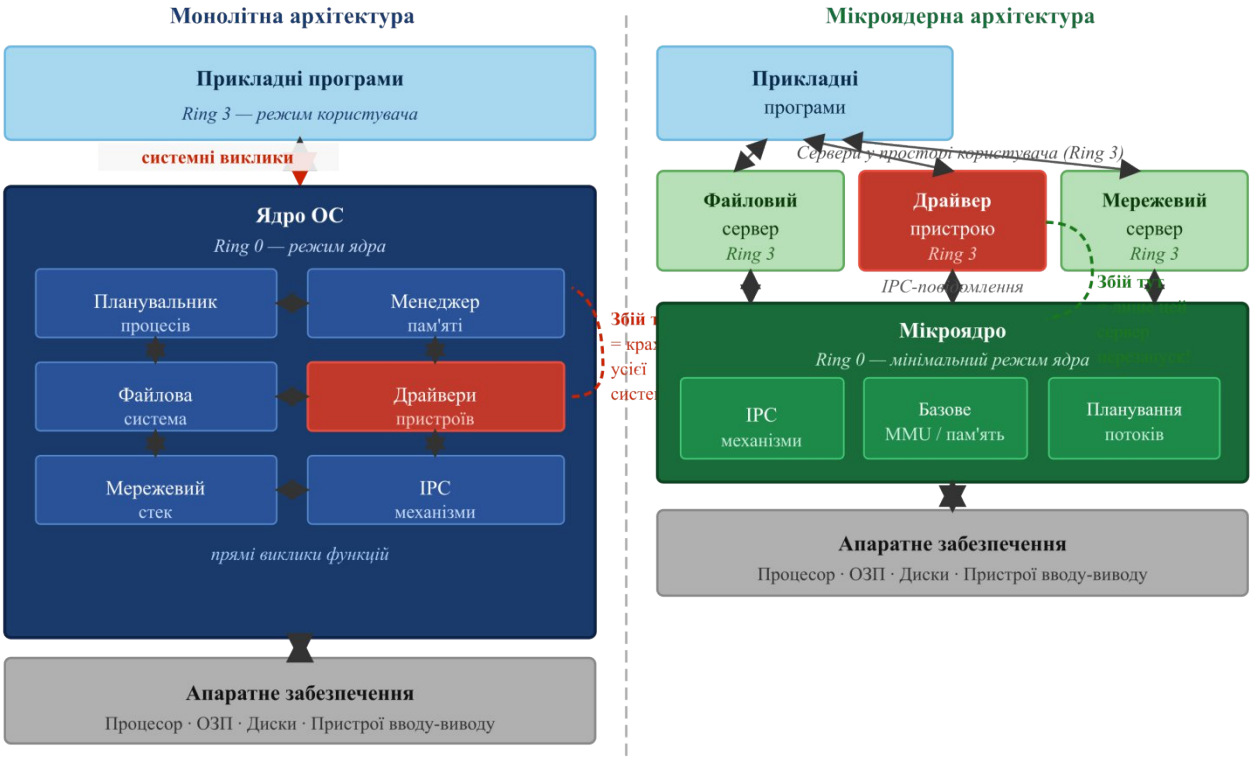

Рисунок 1.2 – Структурна відмінність між монолітною та мікроядерною архітектурами.

Порівняння архітектурних підходів наведено у таблиці 1.2. У таблиці систематизовано ключові характеристики кожної архітектури за критеріями продуктивності, надійності, безпеки та масштабованості.

Таблиця 1.2 — Порівняльна характеристика архітектур операційних систем

Характеристика	Монолітна	Мікроядерна	Гібридна	Модульна
Продуктивність	Висока	Низька	Середня	Середня / висока
Надійність при збої компонента	Низька	Висока	Середня	Низька
Ізоляція між компонентами ядра	Відсутня	Повна	Часткова	Відсутня
Безпека (зменшення поверхні атаки)	Нижча	Вища	Середня	Середня
Складність розробки	Середня	Висока	Висока	Середня
Масштабованість	Обмежена	Висока	Середня	Висока
Час переключення між компонентами	Мінімальний	Значний (IPC)	Помірний	Мінімальний
Відновлення збійного компонента	Неможливе	Автоматичне	Часткове	Неможливе
Приклади ОС	Linux, UNIX, MS-DOS	QNX, MINIX 3, seL4	Windows NT, macOS XNU	Linux з LKM

Вплив архітектурних рішень на безпеку ІКС є предметом постійних досліджень. Монолітне ядро Linux компенсує відсутність структурної ізоляції потужними механізмами безпеки: Linux Security Modules (LSM) надають загальний фреймворк для реалізації різних моделей контролю доступу; SELinux (розроблений АНБ США) та AppArmor реалізують обов'язковий контроль доступу (MAC); seccomp обмежує набір дозволених системних викликів для конкретного процесу до мінімально необхідного підмножина; KASLR ускладнює пошук адрес ядра для атак. Мікроядерна система seL4, розроблена в CSIRO Data61 (Австралія), є єдиним у світі ядром ОС, правильність якого доведена математичними методами формальної верифікації — кожна властивість безпеки доведена так само строго, як математична теорема, і тому seL4 застосовується у критичних системах авіоніки та військових комунікаційних систем, де ціна помилки може бути людською втратою.

1.3 Класифікація операційних систем за призначенням

Різноманіття сучасних обчислювальних пристроїв — від суперкомп'ютерів з мільйонами ядер до мікроконтролерів розміром з монету — породжує потребу в різних типах операційних систем, оптимізованих для конкретних умов застосування. Не існує і не може існувати єдиної «ідеальної» ОС для всіх сценаріїв: компроміси між продуктивністю, енергоспоживанням, надійністю, розміром та вартістю диктують принципово різні рішення залежно від цільової платформи.

Настільні (desktop) операційні системи орієнтовані на забезпечення комфортної роботи індивідуального користувача з широким спектром

прикладних програм. Їх характерними рисами є розвинутий графічний інтерфейс користувача (GUI), підтримка мультимедіа (аудіо, відео, 3D-графіка), широкий перелік підтримуваного периферійного обладнання та простота використання для непідготовленого користувача. Microsoft Windows домінує на ринку з часткою понад 70% за даними StatCounter; зворотна сумісність є настільки принциповою вимогою для Windows, що деякі програми, написані для Windows XP, досі працюють на Windows 11 без жодних змін — ця властивість є одночасно перевагою з погляду корпоративних замовників і серйозним обмеженням з погляду безпеки, оскільки заважає прибрати застарілі механізми. macOS від Apple забезпечує глибоку інтеграцію з власними апаратними платформами (MacBook, Mac Studio, Mac Pro на процесорах M-серії) та акцент на зручності й продуктивності творчих задач. Linux-дистрибутиви — Ubuntu, Fedora, Debian, Arch — пропонують відкритий вихідний код, виняткову гнучкість налаштування та традиційно вищий рівень безпеки, що робить їх популярними серед розробників, системних адміністраторів і дослідників безпеки.

Серверні операційні системи проектуються для роботи в умовах, принципово відмінних від настільних: безперервна робота 24/7 роками без перезавантаження, обслуговування сотень тисяч одночасних мережевих підключень, максимальна надійність та механізми автоматичного відновлення після збоїв, ефективне управління великими обсягами оперативної пам'яті (сучасні серверні платформи підтримують до 12 ТБ ОЗП) та масивами зберігання. Графічний інтерфейс у серверних ОС є або відсутнім, або суто опціональним: більшість операцій виконується через командний рядок або системи конфігураційного управління (Ansible, Puppet, Chef), що зменшує навантаження на ресурси та унеможливлює цілі класи атак, орієнтованих на графічний стек. Linux-системи (Red Hat Enterprise Linux, Ubuntu Server, Debian) займають понад 75% ринку веб-серверів за даними W3Techs станом на 2024 рік. Серверні ОС включають розширені засоби управління: LDAP та Active Directory для централізованої автентифікації, LVM та ZFS для гнучкого управління сховищами, засоби кластеризації Pacemaker та Kubernetes, а також розширені інструменти аудиту та журналізації подій.

Вбудовані (embedded) операційні системи функціонують у пристроях з жорсткими ресурсними обмеженнями: мікропроцесор ARM Cortex-M з тактовою частотою 8–500 МГц, від кількох кілобайт до кількох мегабайт ОЗП, Flash-пам'ять замість жорсткого диска, живлення від батарейки з терміном роботи роками. Сфери застосування охоплюють промислові програмовані логічні контролери, медичні прилади, автомобільну електроніку (блоки управління двигуном, ABS, ESP, системи автопілотування), мережеве обладнання та пристрої IoT. FreeRTOS є лідером за кількістю розгортань у мікроконтролерних системах; VxWorks широко застосовується в авіоніці та аерокосмічній галузі — зокрема бортові комп'ютери марсоходів Curiosity та Perseverance NASA працюють під управлінням саме VxWorks. Безпека вбудованих систем є критичною темою: ці пристрої часто мають тривалий термін служби без оновлень і слабку криптографію. Сумнозвісний ботнет Mirai

у 2016 році залучив мільйони незахищених IoT-пристроїв для рекордних DDoS-атак, ілюструючи реальні наслідки нехтування безпекою вбудованих систем.

Мобільні операційні системи — відносно молода, але надзвичайно масштабна категорія, що охоплює мільярди смартфонів і планшетів. Мобільні ОС вирішують унікальний набір взаємно суперечливих задач: максимальна продуктивність інтерфейсу при мінімальному енергоспоживанні, підтримка безлічі апаратних датчиків (GPS, акселерометр, гіроскоп, магнітометр, кілька камер), управління декількома бездротовими інтерфейсами одночасно (Wi-Fi, Bluetooth, NFC, 5G), суворий контроль над програмами з метою захисту приватності. Android від Google базується на ядрі Linux зі значними власними модифікаціями: Binder IPC ефективніший для часто викликуваних операцій ніж стандартний System V IPC, механізм Wakelocks дозволяє процесам запобігати переходу системи у режим сну, а Low Memory Killer агресивно завершує фонові програми при нестачі пам'яті. iOS від Apple побудована на ядрі XNU і реалізує значно жорсткішу модель безпеки: кожен додаток має власний ізольований контейнер у файловій системі, власний адресний простір і може взаємодіяти з іншими лише через строго визначені механізми (Share Extensions, App Groups, Universal Links) — в результаті поширення шкідливого ПЗ через офіційний App Store є вкрай рідкісним явищем.

Операційні системи реального часу (RTOS) мають специфічну ключову вимогу: гарантований детермінований час реакції на зовнішні події. У RTOS важлива не лише правильність результату, але й момент його отримання — запізнення може бути так само критичним, як і неправильна відповідь. Жорсткі (hard) RTOS гарантують абсолютне дотримання часових обмежень незалежно від навантаження: вони застосовуються у системах управління авіаційними двигунами, ABS та системах активної безпеки автомобілів, медичних апаратах штучного дихання. М'які (soft) RTOS допускають рідкісні та незначні порушення часових обмежень — мультимедійні системи, VoIP-телефонія, де поодинокі затримки є прийнятними. Представниками жорстких RTOS є QNX Neutrino (сертифікований для медичних та автомобільних систем), VxWorks та RTEMS.

Класифікацію операційних систем за призначенням наведено у таблиці 1.3.

Таблиця 1.3 — Класифікація ОС за призначенням та ключові характеристики

Тип ОС	Основне призначення	Ключові вимоги	Типовий обсяг ОЗП	Приклади
Настільні	Робота індивідуального користувача	GUI, сумісність ПЗ, зручність	4–64 ГБ	Windows 11, macOS, Ubuntu
Серверні	Обслуговування мережевих сервісів	Надійність, масштабованість, безпека	16 ГБ – 12 ТБ	RHEL, Windows Server, Debian
Вбудовані	Управління	Малий розмір,	4 КБ – 256	FreeRTOS,

	спеціалізованим обладнанням	стабільність, низьке споживання	МБ	VxWorks, OpenWrt
Мобільні	Смартфони та планшети	Енергоефективність, сенсорний UI, безпека	4–16 ГБ	Android 14, iOS 18
Реального часу	Критичні системи управління	Детерміновані затримки, передбачуваність	64 КБ – 512 МБ	QNX, VxWorks, RTEMS
Хмарні / контейнерні	Хмарна інфраструктура	Швидкий запуск, ізоляція, оркестрація	1–1024 ГБ	CoreOS, Flatcar Linux, Alpine

Рисунок 1.3 відображає частки ринку операційних систем у чотирьох ключових сегментах станом на 2024 рік. На рисунку зображено чотири кругові діаграми, розташовані у два ряди по дві штуки, під спільним заголовком «Розподіл ринку ОС за сегментами (2024)».

Важливо звернути увагу на принципові відмінності в моделях безпеки різних типів ОС. Настільні ОС традиційно надають користувачу широкі права, що спрощує роботу, але збільшує поверхню атаки — наприклад, у Windows донедавна більшість користувачів працювали з правами адміністратора. Серверні ОС, навпаки, застосовують принцип мінімальних привілеїв з самого початку: сервіси запускаються від імені окремих непривілейованих облікових записів, а root-доступ вимагає явної авторизації через sudo. Мобільні ОС реалізують найжорсткіші моделі ізоляції застосунків серед усіх категорій: в iOS і Android кожен додаток виконується в окремій «пісочниці» (sandbox) і може отримувати доступ до апаратних ресурсів лише через строго контрольовані API з явним дозволом користувача — така модель суттєво обмежує можливості зловмисного ПЗ навіть у разі його встановлення на пристрій.

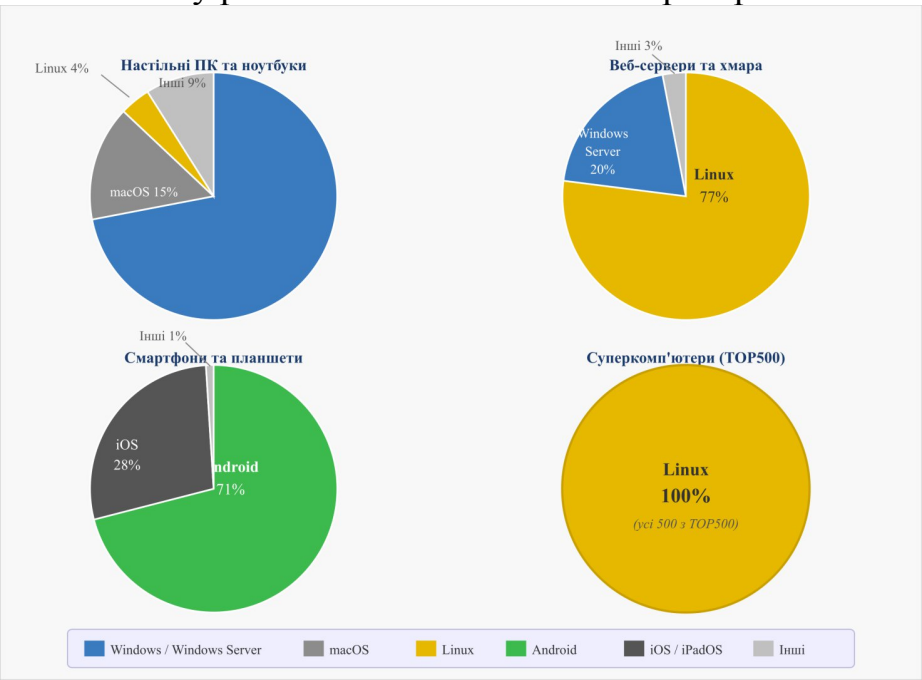

Рисунок 1.3 – Частки ринку операційних систем у чотирьох ключових сегментах станом на 2024 рік.

1.4 Моделі виконання завдань та взаємодія з апаратним забезпеченням

Сучасні операційні системи підтримують кілька фундаментальних моделей організації виконання задач, що безпосередньо впливають на ефективність використання апаратних ресурсів, рівень безпеки та масштабованість інформаційних систем. Розуміння цих моделей є необхідним для будь-якого фахівця, що займається проектуванням, адмініструванням або захистом ІКС, оскільки саме на цьому рівні виникає більшість уразливостей, пов'язаних з некоректною ізоляцією та синхронізацією процесів.

Багатозадачність (multitasking) — здатність ОС виконувати кілька задач «одночасно», переключаючи процесор між ними у швидкій послідовності, що створює ілюзію паралельного виконання навіть на однопроцесорній системі. Кооперативна (cooperative) багатозадачність, характерна для ранніх версій Windows (до 3.x) та класичної Mac OS, передбачає, що програма сама добровільно передає управління ОС через системні виклики: якщо програма зависає або входить у нескінченний цикл, вся система зупиняється, оскільки інші програми не отримують жодного процесорного часу. Витісняльна (preemptive) багатозадачність, реалізована у всіх сучасних ОС починаючи з Windows 95/NT та Linux 1.0, дозволяє ОС примусово перервати виконання будь-якого процесу після закінчення виділеного часового кванту і передати управління наступному незалежно від бажання програми. Механізм витіснення реалізується через апаратний таймерний переривач: у Linux константа HZ визначає частоту таймерних переривань (250–1000 разів на секунду), і при кожному з них планувальник ядра отримує управління та вирішує, чи продовжувати виконання поточного процесу. Витісняльна багатозадачність ускладнює розробку програм, оскільки виникає потреба в синхронізації при роботі зі спільними ресурсами, але забезпечує справедливий розподіл процесорного часу та стійкість системи до зависання окремих програм.

Багатокористувацькість (multiuser) — можливість ОС одночасно обслуговувати кількох незалежних користувачів з повною ізоляцією їхніх даних і сесій. У UNIX-подібних системах багатокористувацький режим є вбудованою фундаментальною властивістю архітектури: кожен процес виконується від імені конкретного користувача (UID) та групи (GID), файли мають власника та тришарову модель прав доступу (власник / група / інші, по три біти rwx для кожної категорії), а ядро перевіряє дозволи при кожному системному виклику без жодних винятків. Концепція суперкористувача (root у Linux, UID = 0) з необмеженими правами є компромісом між безпекою та зручністю адміністрування. Сучасний підхід намагається уникати монолітного всесилля root, розбиваючи його привілеї на окремі capability-можливості у Linux: `CAP_NET_ADMIN` для управління мережею, `CAP_SYS_PTRACE` для налагодження процесів, `CAP_DAC_OVERRIDE` для обходу перевірок прав доступу — більше 40 окремих привілеїв, кожен з яких може надаватись програмі незалежно. Це

реалізація принципу мінімальних привілеїв, докладно розглянутого в темі, присвяченій механізмам безпеки ОС.

Симетрична багатопроцесорність (SMP) та підтримка багатоядерних процесорів є ключовою моделлю для сучасних серверних та настільних систем. У SMP-системі всі процесорні ядра мають рівноправний доступ до спільної оперативної пам’яті через системну шину або кільцеву з’єднувальну структуру. Починаючи з кінця 1990-х років, коли фізичні процесори досягли практичної стелі тактової частоти (~4–5 ГГц через теплові обмеження), виробники перейшли до збільшення кількості ядер: сучасний AMD EPYC може мати до 192 ядер, а Intel Xeon Platinum — до 60. Ядро ОС для ефективної роботи в SMP-середовищі повинне враховувати топологію пам’яті NUMA (Non-Uniform Memory Access), де доступ до «своєї» пам’яті процесора в 3–10 разів швидший, ніж до пам’яті іншого процесорного вузла. Планувальники сучасних ОС — Completely Fair Scheduler (CFS) у Linux та багаторівневий планувальник Windows — підтримують спорідненість процесів (process affinity) та намагаються мінімізувати міграцію потоків між вузлами NUMA задля збереження кешової локальності і максимізації ефективної пропускної здатності.

Рисунок 1.4 ілюструє порівняння моделей виконання задач у різних типах ОС. На рисунку зображено чотири горизонтальних діаграми типу «часова шкала», розташовані вертикально одна під одною з підписами зліва. Загальна горизонтальна вісь — час, розбита на 12 рівних сегментів T1–T12.

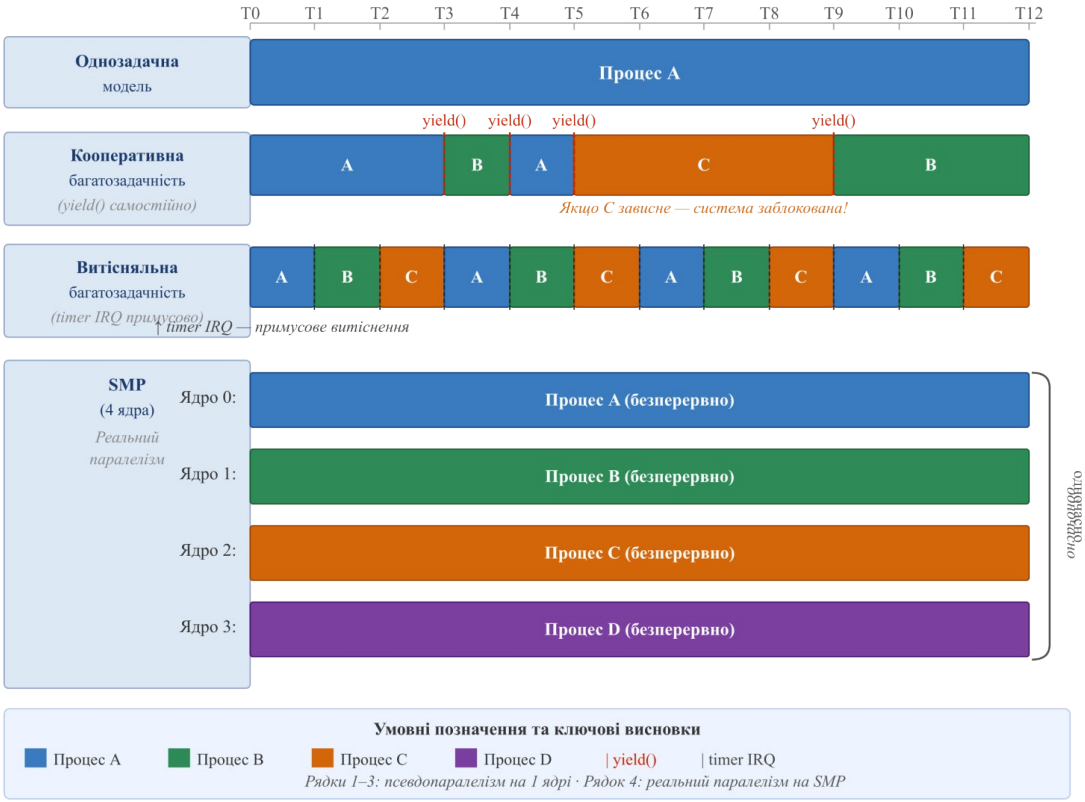

Рисунок 1.4 – Порівняння моделей виконання задач у різних типах ОС

Ключовим апаратним механізмом захисту є **розподіл привілеїв між режимом ядра і режимом користувача**. Архітектура x86/x86-64 визначає чотири рівні привілеїв (кільця захисту, rings): Ring 0 — режим ядра з повним необмеженим доступом до всього апаратного забезпечення; Ring 1 та Ring 2 — теоретично передбачені для драйверів та сервісів ОС, але на практиці не використовуються сучасними ОС; Ring 3 — режим користувача з вкрай обмеженим набором дозволених операцій. Прикладна програма у Ring 3 не може безпосередньо читати чи записувати пам'ять ядра, звертатись до портів вводу-виводу або вимикати апаратні переривання — будь-яка така спроба генерує апаратне виключення General Protection Fault, яке ядро перехоплює і завершує програму. Системний виклик є єдиним легальним способом перетину кордону Ring 3 → Ring 0: спеціальна інструкція процесора SYSCALL виконує перехід у Ring 0 за фіксованою точкою входу ядра, яку неможливо змінити з простору користувача.

У таблиці 1.4 наведено основні категорії системних викликів сучасних ОС та їх призначення.

Таблиця 1.4 — Основні категорії системних викликів та приклади

Категорія	Призначення	Приклади Linux	Приклади Windows
Управління процесами	Створення, завершення, очікування	fork(), exec(), exit(), wait()	CreateProcess(), TerminateProcess()
Управління файлами	Відкриття, читання, запис, закриття	open(), read(), write(), close()	CreateFile(), ReadFile(), WriteFile()
Управління пристроями	Конфігурація та вводу-виводу	ioctl(), mmap(), poll()	DeviceIoControl()
Управління пам'яттю	Виділення та звільнення пам'яті	mmap(), brk(), munmap()	VirtualAlloc(), VirtualFree()
IPC та синхронізація	Труби, сокети, семафори	pipe(), socket(), sem_wait()	CreatePipe(), CreateSemaphore()
Управління інформацією	Час, атрибути, системна інформація	getpid(), stat(), uname()	GetSystemInfo(), GetFileAttributes()
Мережеві операції	Мережеві з'єднання та передача	socket(), bind(), connect(), send()	WSASocket(), WSAConnect()

Розуміння механізму системних викликів є критично важливим для аналізу безпеки: більшість атак на ОС зводиться або до маніпуляції системними викликами (syscall hijacking, ret2libc, SROP), або до пошуку шляхів обходу перевірок, що виконуються при їх обробці. Засоби системного моніторингу — strace у Linux, Process Monitor у Windows — дозволяють відстежувати всі системні виклики конкретного процесу з аргументами та результатами, що є незамінним інструментом як для налагодження програм, так і для аналізу підозрілої активності шкідливого програмного забезпечення під час криміналістичного розслідування інцидентів інформаційної безпеки.

Вплив моделей виконання на безпеку ІКС є багатогранним і не завжди очевидним навіть для досвідчених фахівців. Багатозадачність дозволяє шкідливому ПЗ виконуватись у фоновому режимі непомітно для користувача. Незахищений міжпроцесний обмін через спільну пам’ять може стати каналом витоку конфіденційних даних — класичний приклад цього класу атак є Spectre та Meltdown (2018), що використовували особливості спекулятивного виконання інструкцій у процесорах для читання пам’яті ядра через побічний канал процесорного кешу. Синхронізаційні уразливості — стани гонитви (race conditions) — можуть бути навмисно спровоковані зловмисником для атак типу TOCTOU (Time-of-Check Time-of-Use): якщо між перевіркою прав доступу до файлу та самою файловою операцією існує часовий проміжок, зловмисник може підмінити файл через символічне посилання (symlink), обійшовши перевірку безпеки — цей вектор використовувався в численних реальних уразливостях проти привілейованих процесів Linux та macOS.

Підсумовуючи матеріал першої теми, слід наголосити: архітектура операційної системи — це фундаментальна основа, що визначає можливості, обмеження та рівень безпеки будь-якої ІКС протягом усього її життєвого циклу. Вибір ОС для конкретного застосування повинен базуватись на чіткому розумінні архітектурних компромісів, вимог до продуктивності та надійності, а також моделі загроз для даного середовища і категорії даних, що обробляються системою. Фахівець з інформаційної безпеки зобов’язаний розуміти внутрішній устрій ОС на достатньому рівні, щоб передбачати потенційні вектори атаки, коректно трактувати логи аудиту та обирати адекватні засоби захисту. Наступні теми курсу розвивають цей фундамент, заглиблюючись у конкретні механізми управління процесами і потоками, пам’яттю, файловими системами та засобами безпеки сучасних операційних систем.

Тема 2. Механізми управління процесами та потоками виконання

2.1. Модель процесу в операційній системі

Будь-яка сучасна операційна система є складним програмним комплексом, що управляє великою кількістю обчислювальних задач одночасно. Центральним поняттям у цьому управлінні є поняття процесу — фундаментальної абстракції, на якій ґрунтується вся багатозадачність. Розуміння моделі процесу є необхідною умовою для грамотного системного програмування, адміністрування та забезпечення безпеки інформаційно-комунікаційних систем, оскільки саме на рівні процесів реалізується більшість механізмів управління ресурсами, розподілу привілеїв та ізоляції задач. Процес є значно складнішою сутністю, ніж просто програмний код, що виконується на процесорі, і для його коректного опису необхідно розглянути ряд ключових концепцій.

Процес можна визначити як програму в стані виконання, тобто не просто набір інструкцій, збережених на диску у вигляді виконуваного файлу, а динамічну сутність, що включає поточний стан виконання, виділені ресурси та контекст. Один і той самий виконуваний файл може породжувати кілька одночасно активних процесів, кожен з яких є цілком незалежною сутністю з власним адресним простором та власними ресурсами. Наприклад, запустивши текстовий редактор двічі, ми отримуємо два різних процеси, хоча вони обидва завантажені з одного й того самого файлу на диску. Ця відмінність між статичною програмою та динамічним процесом є принциповою для розуміння того, як операційна система організовує паралельне виконання задач.

З точки зору апаратного забезпечення процесор у будь-який момент часу виконує лише одну послідовність команд. Ілюзія одночасного виконання кількох програм досягається за рахунок дуже швидкого перемикання між процесами — настільки швидкого, що людина не помічає цього чергування і вважає, що всі програми працюють одночасно. Цей підхід, відомий як псевдопаралелізм (pseudoparallelism) або логічний паралелізм, є основою концепції багатозадачності. У системах із кількома фізичними процесорними ядрами (multicore або multiprocessor) забезпечується вже справжній фізичний паралелізм: різні процеси або потоки можуть виконуватись на різних ядрах дійсно одночасно, без будь-якого чергування. Операційна система повинна коректно організовувати обидва види паралелізму і забезпечувати узгоджений доступ до спільних ресурсів у будь-якому випадку.

Важливо розуміти, що операційна система сама є програмою, що виконується на тому самому апаратному забезпеченні, що і прикладні процеси. Ядро ОС не є «зовнішньою» сутністю — воно також займає частину пам’яті та виконується на процесорі. Відмінність полягає у рівні привілеїв: ядро виконується у привілейованому режимі (режимі ядра, ring 0 у x86-архітектурі), де доступні всі інструкції процесора та весь адресний простір, тоді як прикладні процеси виконуються у режимі користувача (user mode, ring 3), де доступ до апаратних ресурсів суворо обмежений. Перехід між цими режимами

відбувається через системні виклики (system calls) — спеціальний механізм, що дозволяє прикладним процесам запитувати у ядра виконання привілейованих операцій. Саме ця межа між режимами є однією з головних ліній захисту операційної системи від некоректної або зловмисної поведінки процесів.

Кожен процес має власний адресний простір — область пам'яті, відокремлену від інших процесів, у якій зберігаються: програмний код (текстовий сегмент або сегмент коду), ініціалізовані та неініціалізовані статичні дані (сегменти .data та .bss), динамічно виділена пам'ять (купа або heap) та стек виклику функцій (stack). Це розмежування адресних просторів є ключовим механізмом захисту: збій одного процесу, як правило, не призводить до пошкодження даних іншого, оскільки вони просто не мають доступу до пам'яті один одного без явного дозволу операційної системи. Таким чином, операційна система виступає арбітром, що надає кожному процесу ілюзію повноправного та ізольованого використання обчислювальних ресурсів.

Ядро операційної системи зберігає всю необхідну інформацію про кожен процес у спеціальній структурі даних. У UNIX-подібних системах вона традиційно називається Process Control Block (PCB), у Windows — EPROCESS, а у деяких навчальних джерелах використовується термін «дескриптор процесу». Незалежно від назви, ця структура є «посвідченням особи» процесу в очах операційної системи і містить увесь необхідний набір метаданих для управління ним. Саме PCB дозволяє операційній системі призупиняти один процес, перемикатися на інший і через деякий час повертатися до першого — з тієї самої точки, де він був зупинений.

Типова структура PCB включає такі поля: ідентифікатор процесу (PID — Process Identifier), ідентифікатор батьківського процесу (PPID — Parent PID), поточний стан процесу (Running, Ready, Waiting тощо), значення лічильника команд та усіх регістрів процесора на момент останнього призупинення, інформацію про розподіл пам'яті (базові адреси та розміри сегментів, таблиці сторінок), список відкритих файлових дескрипторів та мережевих сокетів, планувальну інформацію (пріоритет, статистика використання CPU), а також відомості про власника процесу (UID, GID) та права доступу. Операція збереження та відновлення вмісту PCB при перемиканні між процесами називається перемиканням контексту (context switch) і є невід'ємною частиною роботи планувальника.

PID є унікальним числовим ідентифікатором, що присвоюється кожному процесу під час його створення і залишається незмінним протягом усього його існування. У Linux PID 1 традиційно належить процесу init або systemd — першому процесу, що запускається ядром після завантаження і є прабатьком усіх інших процесів у системі. Переглянути список активних процесів та їх PID можна за допомогою команд ps, top або htop. Крім PID, у UNIX-системах існує поняття PGID (Process Group ID) та SID (Session ID), що дозволяють об'єднувати процеси у логічні групи та сесії для управління сигналами і ресурсами.

Перемикання контексту є одним з найбільш частих і ресурсомістких системних операцій. Під час перемикання ядро зберігає весь вміст регістрів

поточного процесу у його PCB, оновлює допоміжні структури планувальника, перемикає покажчик на таблицю сторінок нового процесу (що автоматично змінює адресний простір), завантажує регістри нового процесу з його PCB і передає управління лічильнику команд нового процесу. На сучасних процесорах x86-64 перемикання контексту займає від кількох сотень до кількох тисяч тактів, але непряма вартість (через «холодний» TLB та кеш-пам'ять) може бути суттєво вищою. Саме тому надмірно часте перемикання між великою кількістю процесів знижує загальну продуктивність системи — ефект, відомий як «thrashing планувальника». Оптимальний квант часу є компромісом між швидким відгуком та мінімізацією накладних витрат на перемикання.

Таблиця 2.1 – Основні поля блоку управління процесом (PCB)

Поле	Тип даних	Призначення
PID	Ціле число	Унікальний ідентифікатор процесу
PPID	Ціле число	Ідентифікатор батьківського процесу
Стан	Перелік	Поточний стан процесу (Ready, Running, Waiting…)
Лічильник команд	Адреса	Наступна інструкція для виконання
Регістри CPU	Масив	Повний знімок регістрів процесора
Таблиця сторінок	Покажчик	Адресація віртуальної пам'яті процесу
Список файлових дескрипторів	Масив	Відкриті файли, сокети, пристрої
Інформація планувальника	Структура	Пріоритет, час CPU, черги
UID / GID	Ціле число	Власник та група процесу
Сигнальна маска	Бітова маска	Заблоковані та оброблювані сигнали

Центральне місце у моделі процесу посідає поняття стану процесу. Операційна система у будь-який момент часу веде облік того, чим займається кожен процес: виконується він на процесорі, очікує черги на виконання, чи заблокований у зв'язку з очікуванням введення-виведення. Класична п'ятистанова модель, що застосовується у більшості підручників та реальних операційних систем, описує такі стани: новий (New), готовий (Ready), виконується (Running), очікування (Waiting або Blocked) та завершений (Terminated). Кожен з цих станів має чітко визначену семантику, а переходи між ними відбуваються під дією конкретних подій — диспетчеризації, переривань, системних викликів або завершення операцій введення-виведення.

Таблиця 2.2 – Стани процесу та умови переходів

Стан	Позначення	Опис	Умова входу
Новий	New	Процес щойно створено, ОС ще не розмістила його в пам'яті повністю	fork() / CreateProcess()
Готовий	Ready	Процес готовий до виконання,	Завершення завантаження;

Стан	Позначення	Опис	Умова входу
		але очікує доступу до CPU	I/O complete; витіснення
Виконується	Running	Процес активно виконується на процесорі	Dispatch планувальника
Очікування	Waiting	Процес чекає завершення I/O або настання події	Системний виклик I/O; очікування сигналу
Завершений	Terminated	Виконання процесу завершено або він примусово зупинений	exit(); сигнал SIGKILL

На рисунку 2.1 зображено класичну п’ятистанову діаграму переходів між станами процесу у вигляді орієнтованого графа. Вузли відповідають п’яти станам, а підписані дуги — подіям або діям, що ініціюють переходи.

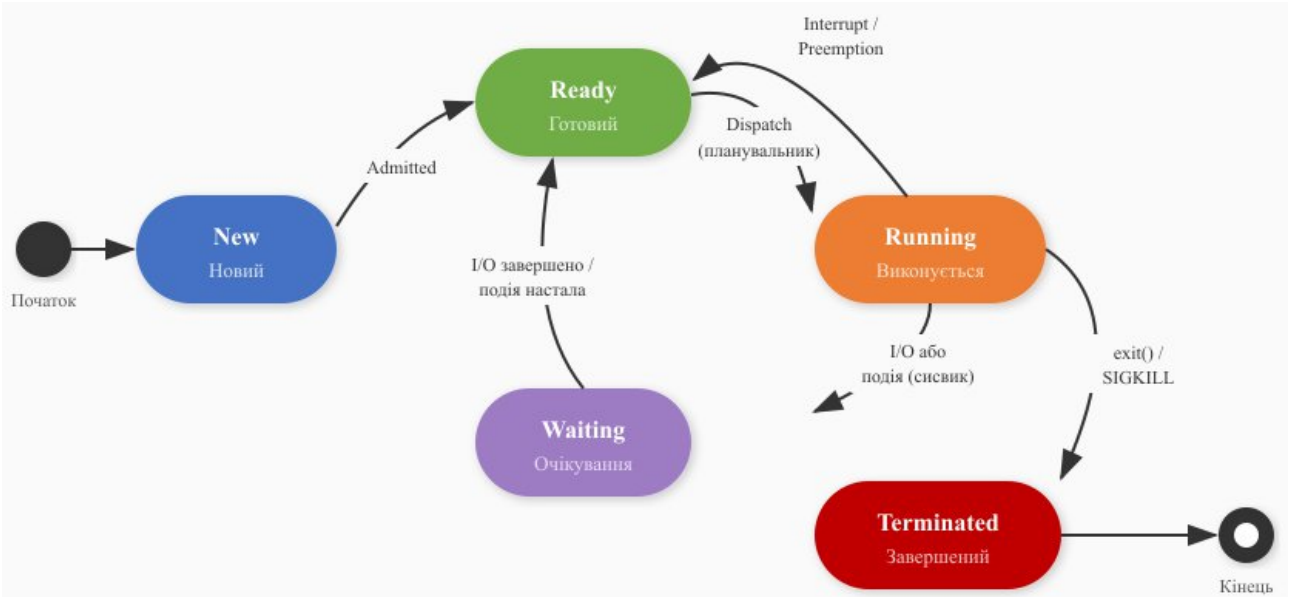

Рисунок 2.1 – П’ятистанова діаграма переходів між станами процесу

На діаграмі видно, що процес може повертатися між станами Ready та Running багаторазово протягом свого існування — кожного разу, коли планувальник призначає або забирає процесор. Стан Waiting є тупиковим в тому сенсі, що процес не може самостійно вийти з нього — він залежить від зовнішньої події (завершення операції введення-виведення, отримання сигналу, спрацювання таймера). Стан Terminated є кінцевим, але процес може затриматися у ньому до того моменту, поки батьківський процес не зчитає код завершення через wait() — у цьому випадку він перебуває у стані «зомбі».

У реальних операційних системах ця базова п’ятистанова модель може розширюватись. Наприклад, у Linux та Windows існує стан «призупинений» або «заморожений» (Suspended), коли процес вивантажується у область підкачки (swap) через нестачу оперативної пам’яті. Цей стан може застосовуватись як до готових (Suspended Ready), так і до заблокованих (Suspended Blocked) процесів. Крім того, у системах реального часу може бути стан «очікування на таймер» (Timer Wait), що відрізняється від загального Waiting більш жорсткими гарантіями часу пробудження. У Solaris та деяких UNIX-системах існує також

стан Zombie — завершений процес, що чекає на зчитування коду виходу батьківським процесом. Таким чином, повна діаграма станів реальної ОС є значно складнішою за навчальну п’ятистанову модель, проте остання охоплює усі принципово важливі переходи і є достатньою для розуміння поведінки планувальника.

Стан процесу зберігається у відповідному полі PCB і є головним критерієм для планувальника при прийнятті рішення про те, кому надати процесор. Планувальник ОС підтримує щонайменше дві ключові структури даних: чергу готових процесів (Ready Queue) — список усіх процесів у стані Ready, що очікують на отримання CPU, та чергу очікування (Wait Queue або Device Queue) — список процесів, що очікують на певний ресурс або подію. У системах із дисковими пристроями може існувати окрема черга очікування для кожного пристрою, що дозволяє оптимізувати порядок виконання операцій введення-виведення незалежно від планування CPU. Взаємодія цих черг і переходи між ними формують основу динамічного управління ресурсами у операційній системі.

Важливим є механізм ієрархії процесів. У UNIX-подібних операційних системах процеси організовані у дерево: кожен процес (крім першого — init або systemd) має батьківський процес, що породив його за допомогою системного виклику fork(). Виклик fork() створює майже ідентичну копію батьківського процесу: дочірній процес успадковує адресний простір, відкриті файлові дескриптори, маски сигналів та більшість атрибутів батька. Єдина відмінність у тому, що fork() повертає дочірньому процесу 0, а батьківському — PID дочірнього. Після fork() зазвичай викликається execve(), що замінює образ дочірнього процесу новою програмою, зберігаючи при цьому PID та більшість атрибутів.

Сучасні ОС реалізують техніку Copy-On-Write (COW) для оптимізації fork(): дочірній процес спочатку розділяє сторінки пам’яті з батьком, і лише при спробі запису у них відбувається фактичне копіювання. Це різко прискорює створення процесів, адже у більшості випадків після fork() відразу ж відбувається execve(), і копіювати весь адресний простір не потрібно. Коли дочірній процес завершується, але батько ще не зчитав його код завершення через wait(), процес перебуває у стані «зомбі» (zombie): його ресурси звільнено, але запис у таблиці процесів зберігається. Надмірне накопичення зомбі є поширеною проблемою у довгострокових серверних додатках і може вичерпати ліміт PID у системі. У Windows замість fork()/exec() використовується CreateProcess() — єдиний виклик, що одночасно і створює процес, і завантажує виконуваний файл.

Міжпроцесова комунікація (Inter-Process Communication, IPC) є невід’ємною частиною роботи з процесами, адже ізольовані процеси часто мають потребу у взаємодії. UNIX-системи надають кілька механізмів IPC: неіменовані та іменовані канали (pipes, FIFO), що забезпечують однонаправлену потокову передачу даних між спорідненими процесами; черги повідомлень (message queues), що дозволяють обмінюватись структурованими блоками даних; спільна пам’ять (shared memory), що є найшвидшим

механізмом — кілька процесів відображають одну й ту саму фізичну сторінку пам’яті у свої адресні простори і можуть безпосередньо читати та писати в неї; сигнали (signals) — легковагові асинхронні сповіщення; та сокети (sockets) — універсальний механізм, що працює як для локальної (UNIX-domain sockets), так і для мережевої комунікації. Кожен з цих механізмів має свої переваги, обмеження та сценарії застосування, і вибір між ними є важливим архітектурним рішенням при проектуванні розподілених та багатопроцесних систем.

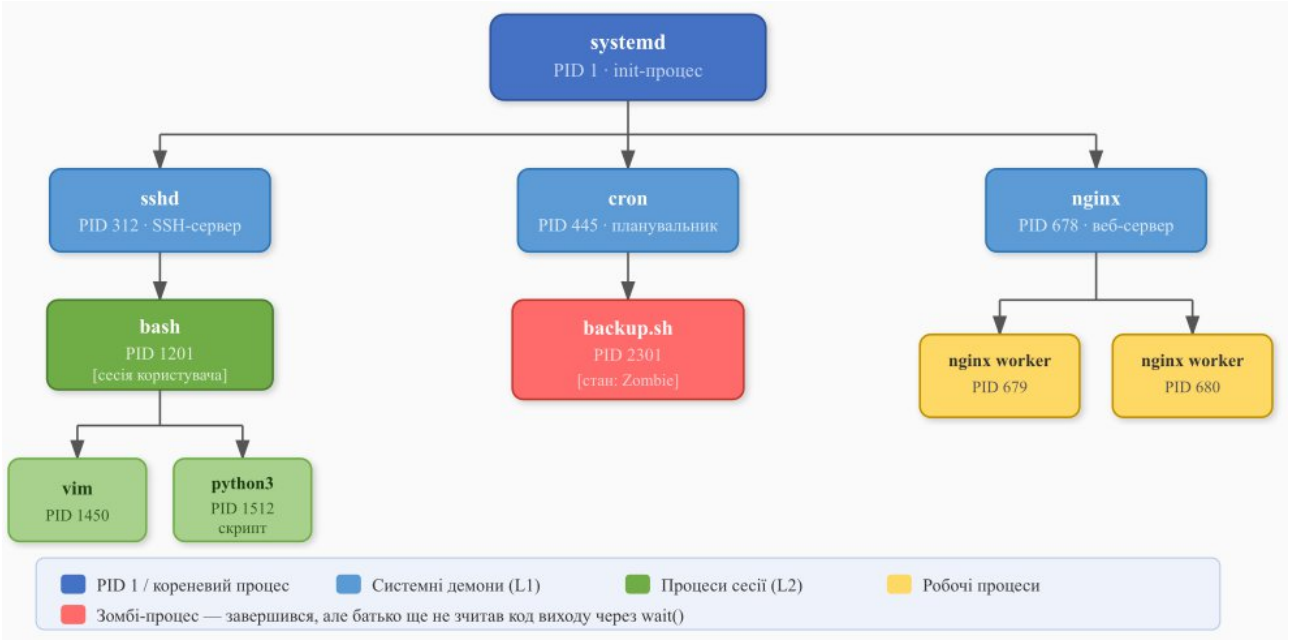

Рисунок 2.2 – Ієрархія процесів у UNIX-системі: дерево від systemd

На рисунку 2.2 зображено типове дерево процесів UNIX-системи. Кореневим є systemd з PID 1, від якого розгалужуються системні демони: sshd (SSH-сервер), cron (планувальник задач), nginx (веб-сервер). Від sshd відгалужується bash-сесія конкретного користувача, що у свою чергу породила vim та python3. Nginx має кілька робочих (worker) процесів. Виділеним червоним кольором позначено процес backup.sh, що завершив роботу та перебуває у стані зомбі в очікуванні зчитування коду завершення процесом cron.

2.2. Потоки виконання

Поряд з процесами сучасні операційні системи підтримують ще одну, більш легковагову одиницю паралелізму — потік виконання (thread, або нитка). Потік є послідовністю команд, що виконується у рамках одного процесу і розділяє з іншими потоками того ж процесу адресний простір, відкриті файлові дескриптори, глобальні змінні та інші ресурси. Власними для кожного потоку є лише лічильник команд, набір регістрів процесора та стек виконання, що і робить потоки набагато «дешевшими» за процеси: їх створення та знищення

відбувається у рази швидше, а перемикання між потоками одного процесу не потребує зміни адресного простору, що різко скорочує накладні витрати.

Порівняння процесів та потоків за ключовими характеристиками наведено у таблиці 2.3.

Таблиця 2.3 – Порівняння процесів та потоків виконання

Характеристика	Процес	Потік
Адресний простір	Власний, ізольований	Спільний для всіх потоків процесу
Вартість створення	Висока (копіювання ресурсів)	Низька (виділення лише стеку і PCB потоку)
Вартість перемикання	Висока (зміна адресного простору)	Низька (не потрібна зміна простору)
Комунікація	IPC: pipes, sockets, shared memory	Безпосередньо через спільну пам’ять
Ізоляція при збої	Висока (збій не торкається інших)	Низька (збій потоку може знищити весь процес)
Власні ресурси	Файлові дескриптори, пам’ять, сигнали	Лічильник команд, регістри, стек
Управління ядром	Завжди в ядрі	Ядро або простір користувача

Багатопотокова модель програмування широко застосовується у сучасних прикладних програмах для досягнення паралелізму та підвищення відгуку. Веб-сервери (Apache, nginx), бази даних (PostgreSQL, MySQL), відеоредактори та ігри активно використовують потоки: один потік може відповідати за прийом мережевих з’єднань, інший — за обробку запитів, третій — за запис у журнал подій. Такий поділ дозволяє уникнути ситуації, коли повільна операція (наприклад, запис великого файлу на диск) блокує весь додаток і робить інтерфейс невідповідним. Крім того, на багатоядерних процесорах потоки одного процесу можуть справді виконуватися паралельно на різних ядрах, що дає реальне прискорення обчислень.

Стандартизація потоків у UNIX-системах здійснена через специфікацію POSIX (POSIX.1c, 1995), що визначає набір функцій для роботи з потоками, відомих як pthreads (POSIX threads). Ця бібліотека є частиною GNU C Library (glibc) і реалізована у більшості UNIX-подібних систем. Основні функції pthreads включають pthread_create() для створення потоку, pthread_join() для очікування завершення, pthread_mutex_lock() та pthread_mutex_unlock() для роботи з м’ютексами, а також pthread_cond_wait() та pthread_cond_signal() для роботи з умовними змінними. У Windows потоки управляються через WinAPI-функції CreateThread(), WaitForSingleObject() та CRITICAL_SECTION. Сучасні мови програмування, як правило, надають власні більш зручні абстракції поверх системних потоків: Java.lang.Thread, C++ std::thread, Python threading.Thread, Rust std::thread — всі вони врешті-решт спираються на системні примітиви відповідної ОС.

На системному рівні підтримка потоків може реалізовуватись трьома принципово різними способами. Потоки ядра (kernel-level threads, KLT) безпосередньо відомі операційній системі: ядро зберігає власний блок управління для кожного потоку, планувальник ОС розглядає потоки як самостійні одиниці планування і може виконувати їх паралельно на різних процесорних ядрах. Блокування одного потоку на операції введення-виведення не зупиняє інші потоки того самого процесу. Однак ціна за це — більші накладні витрати на системні виклики при операціях з потоками. Потоки простору користувача (user-level threads, ULT) керуються спеціальною бібліотекою і є невидимими для ядра: з точки зору ОС весь процес є одним потоком, а розподіл між кількома ULT відбувається всередині бібліотеки. Переключення між ULT не вимагає системного виклику і є дуже швидким, але блокування одного ULT на системному виклику блокує весь процес. Гібридна модель M:N (green threads у Go, goroutines) відображає M потоків простору користувача на N потоків ядра, намагаючись поєднати переваги обох підходів.

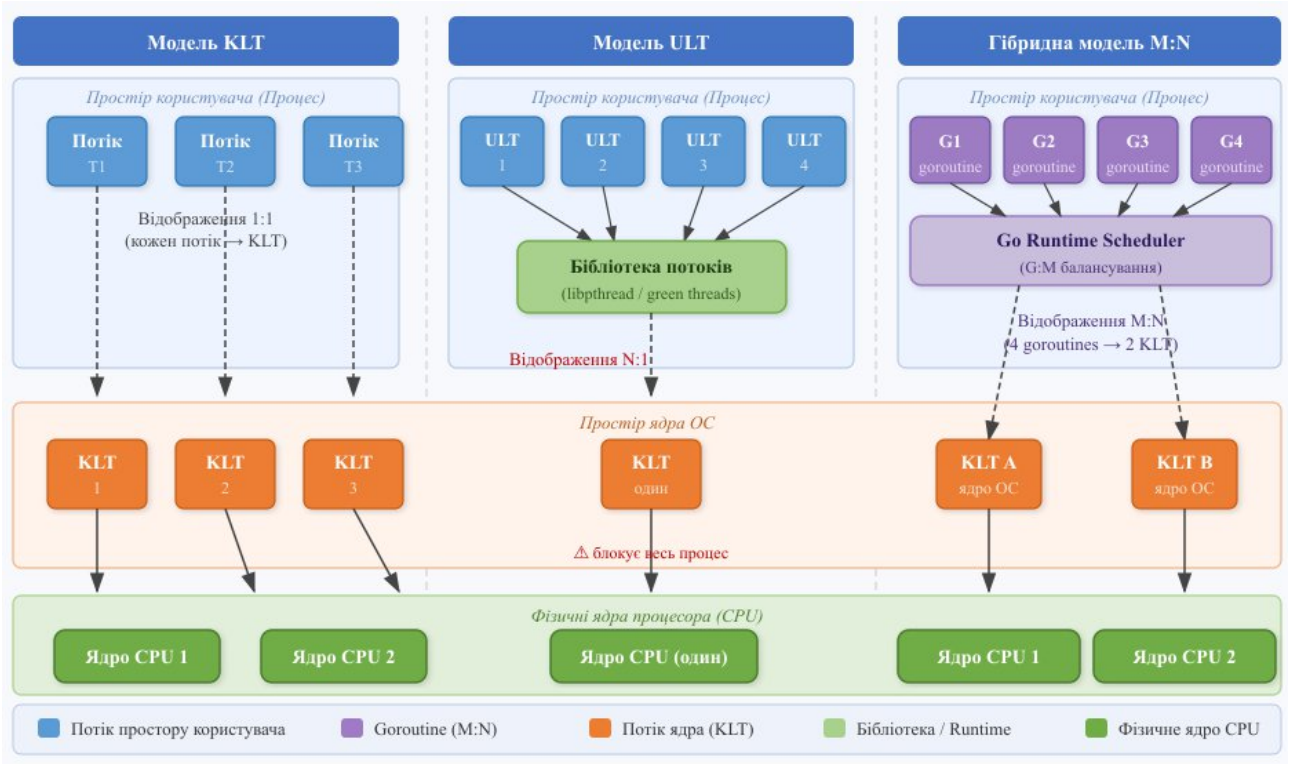

Рисунок 2.3 – Моделі реалізації потоків: KLT, ULT та гібридна M:N

На рисунку 2.3 схематично зображено три моделі реалізації потоків. У моделі KLT кожен потік процесу має відповідний потік ядра та може виконуватись на окремому ядрі CPU. У моделі ULT усі потоки відображаються на єдиний потік ядра через бібліотеку — паралельне виконання на різних ядрах неможливе. У гібридній моделі M:N (приклад — goroutines у Go) чотири goroutine відображаються на два потоки ядра, що розподіляються по двох ядрах CPU: ядро Go-рантайму самостійно планує goroutine між KLT.

Переключення між потоками одного процесу є значно дешевшим за переключення між процесами, проте все одно несе певні накладні витрати, що

необхідно враховувати при проектуванні систем. При переключенні між потоками ядру необхідно зберегти регістри поточного потоку (включаючи лічильник команд та покажчик стеку), завантажити регістри нового потоку і переключити покажчик стеку на стек нового потоку. Адресний простір при цьому не змінюється, таблиця сторінок залишається тією самою, і дорогої операції TLB flush (очищення буфера асоціативної трансляції адрес) не відбувається. Це є принциповою відмінністю від переключення між процесами, де TLB flush є обов'язковим і суттєво знижує продуктивність.

Слід окремо зупинитися на проблемі безпеки у багатопотокових програмах. Спільна пам'ять, що є головною перевагою потоків перед процесами, водночас є і головним джерелом помилок. Функція або модуль вважається потокобезпечним (thread-safe), якщо він коректно виконується при одночасному виклику кількома потоками. Причиною відсутності потокобезпечності найчастіше є наявність статичних або глобальних змінних, що змінюються у функції без синхронізації. Класичний приклад — функція стандартної бібліотеки C strtok(), що зберігає внутрішній стан у статичній змінній і тому не є потокобезпечною; її потокобезпечний варіант strtok_r() вимагає передачі зовнішнього буфера стану. При розробці бібліотек та фреймворків, що будуть використовуватись у багатопотоковому середовищі, потокобезпечність є обов'язковою вимогою до документування кожної публічної функції чи метода.

2.3. Алгоритми планування задач

Після з'ясування сутності процесів і потоків природно виникає питання: яким чином операційна система вирішує, якому з них надати процесор у конкретний момент? Цим займається планувальник (scheduler) — підсистема ядра ОС, реалізована у вигляді певного алгоритму або набору алгоритмів. Метою планувальника є досягнення балансу між кількома, часто протилежними, цілями: максимізувати завантаження процесора (CPU utilization), мінімізувати час відгуку для інтерактивних задач, забезпечити справедливий розподіл ресурсів між усіма процесами, мінімізувати час обертання (turnaround time — від початку до завершення задачі) та задовольнити вимоги реального часу для відповідних задач. Жоден алгоритм планування не є ідеальним у всіх ситуаціях, тому у реальних ОС використовуються складні гібридні схеми.

Для оцінки якості алгоритмів планування використовуються такі основні метрики: час обертання (turnaround time) — загальний час від подання задачі до її завершення; час очікування (waiting time) — сумарний час, проведений у черзі Ready; час відгуку (response time) — час від подання задачі до першої реакції системи; пропускна здатність (throughput) — кількість задач, завершених за одиницю часу. Оптимізувати всі ці метрики одночасно неможливо: алгоритм, що мінімізує середній час обертання, може бути несправедливим щодо окремих процесів; алгоритм, що гарантує короткий

відгук, може мати низьку пропускну здатність через накладні витрати на часті перемикання.

Розрізняють також планувальники з витісненням (preemptive scheduling) та без нього (non-preemptive або cooperative scheduling). У невитісняючих системах процес утримує процесор доти, поки добровільно не звільнить його (через системний виклик або завершення). Перші версії Windows (3.x) та ранні версії Mac OS використовували саме кооперативну багатозадачність, що мала суттєвий недолік: зависаючий процес міг повністю заблокувати систему. Сучасні ОС (починаючи з Windows NT та Linux) використовують переважно витісняючі планувальники, де ядро може примусово перервати виконання будь-якого процесу і передати процесор іншому — наприклад, після вичерпання кванту або при появі процесу з вищим пріоритетом. Витіснення є значно надійнішим підходом, оскільки забезпечує відгук системи навіть при наявності процесів у нескінченних циклах або зі збоями, хоча й вимагає більш обережного програмування — адже процес може бути перерваний у будь-який момент.

Найпростішим алгоритмом є FCFS (First-Come, First-Served) — обслуговування у порядку надходження. Черга готових процесів обслуговується як звичайна черга (FIFO): процес, що першим потрапив у стан Ready, першим і отримає процесор. Алгоритм простий у реалізації та не спричиняє голодування, але страждає від «ефекту конвою»: якщо перший у черзі є CPU-bound процес із тривалим виконанням, всі наступні процеси (навіть дуже короткі) змушені чекати. Це різко збільшує середній час очікування.

SJF (Shortest Job First) — найкоротша задача спочатку. Планувальник обирає для виконання той процес, що має найменший очікуваний час виконання. Алгоритм є оптимальним за критерієм мінімального середнього часу очікування, однак має суттєву практичну проблему: реальний час виконання задачі заздалегідь невідомий. Для вирішення цього застосовуються прогнозні методи, зокрема експоненційне усереднення попередніх вимірів. Витісняюча версія SJF називається SRTF (Shortest Remaining Time First) і перераховує рішення щоразу, коли з’являється новий процес.

Round Robin (RR) є основою більшості практичних планувальників загального призначення. Кожному процесу надається фіксований квант часу (time quantum, зазвичай 10–100 мс), після вичерпання якого він примусово витісняється і переміщується у кінець черги. Алгоритм забезпечує справедливий розподіл CPU та хороший час відгуку для інтерактивних задач. Вибір розміру кванту є компромісом: малий квант дає кращий відгук, але збільшує накладні витрати на перемикання контексту; великий квант перетворює RR на FCFS.

Планування за пріоритетами (Priority Scheduling) призначає кожному процесу числовий пріоритет і обирає для виконання процес із найвищим пріоритетом серед готових. Пріоритети можуть бути статичними (призначаються під час створення і не змінюються) або динамічними (ОС коригує їх на основі поведінки процесу). Головна проблема — голодування (starvation): процеси низького пріоритету можуть чекати нескінченно. Рішення

— «старіння» (aging): пріоритет процесу поступово підвищується зі збільшенням часу очікування.

Багаторівнева черга зі зворотним зв’язком (Multilevel Feedback Queue, MLFQ) є найскладнішим і найпоширенішим алгоритмом у реальних системах. Вона підтримує кілька черг з різними пріоритетами та різними квантами часу. Нові процеси потрапляють у чергу найвищого пріоритету. Якщо процес вичерпує квант, не завершившись, він переміщується до черги нижчого пріоритету з більшим квантом. Процеси, що часто звільняють CPU добровільно (I/O-bound), залишаються у чергах вищого пріоритету. Таким чином, MLFQ адаптивно визначає природу кожного процесу: CPU-bound задачі опускаються вниз, а інтерактивні — залишаються нагорі.

Таблиця 2.4 – Порівняльна характеристика алгоритмів планування

Алгоритм	Витіснення	Середній час очікування	Відгук	Справедливість	Реалізація
FCFS	Ні	Погано (ефект конвою)	Погано	Порядкова	Дуже проста
SJF	Ні / SRTF — так	Оптимально	Добре	Дискримінація довгих	Складна (прогноз)
Round Robin	Так (квант)	Середньо	Відмінно	Висока	Проста
Priority	Так/Ні	Залежить	Добре для пріоритетних	Низька (голодування)	Середня
MLFQ	Так	Добре	Дуже добре	Висока	Дуже складна
CFS (Linux)	Так	Добре	Добре	Висока (вагова)	Дуже складна

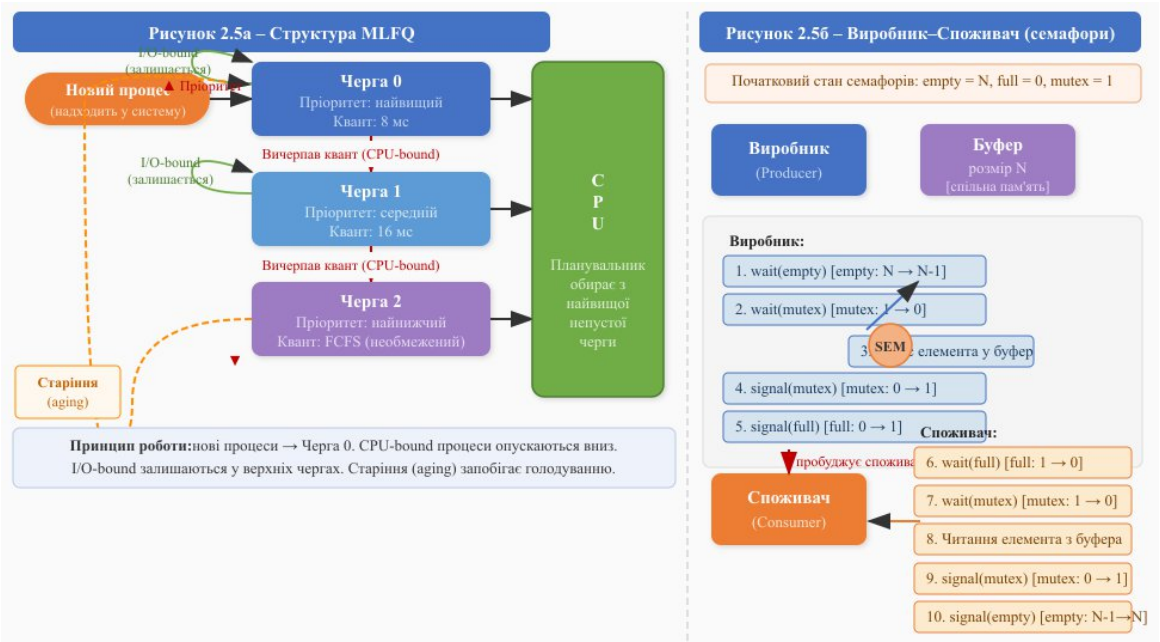

Рисунок 2.5 – Структура багаторівневої черги зі зворотним зв’язком (MLFQ)

На рисунку 2.5 зображено структуру MLFQ з трьома рівнями черг. Новий процес потрапляє у чергу найвищого пріоритету (Q0) з найменшим квантом часу (8 мс). Якщо він вичерпує квант без добровільного звільнення CPU, планувальник класифікує його як CPU-bound і переміщує у Q1 (квант 16 мс), а потім — у Q2 (FCFS, необмежений квант). I/O-bound процеси, що часто добровільно звільняють CPU до вичерпання кванту, залишаються у чергах вищого пріоритету. Механізм «старіння» (aging) переміщує процеси з Q2 назад до Q0 після тривалого очікування, запобігаючи голодуванню.

Починаючи з версії 2.6.23 (2007 рік) у Linux використовується планувальник CFS (Completely Fair Scheduler). Його ідея полягає не у фіксованих квантах часу, а у забезпеченні кожному процесу рівної частки процесорного часу, пропорційної його «вазі» (що визначається параметром nice). CFS відстежує для кожного процесу «віртуальний час виконання» (vruntime) і завжди обирає для виконання процес із мінімальним vruntime — тобто той, що найбільше «недоотримав» процесорного часу. Для ефективного вибору мінімуму використовується червоно-чорне дерево (сортоване за vruntime), що дає час O(log n) на операцію. CFS є єдиним планувальником, що одночасно справедливий, ефективний та масштабується на сотні процесів без деградації.

Параметр nice у Linux задає відносну «ввічливість» процесу щодо інших: від -20 (найвищий пріоритет, процес «не поступається» іншим) до +19 (найнижчий пріоритет, максимально «ввічливий»). Значення за замовчуванням — 0. Підвищити пріоритет (знизити значення nice) може лише суперкористувач, тоді як будь-який користувач може знизити пріоритет власного процесу. Вага процесу у CFS розраховується за формулою, що відображає геометричну прогресію: кожен крок у значенні nice дає приблизно 10% зміну у частці CPU. Таким чином, процес з nice=0 отримує приблизно в 10 разів більше CPU, ніж процес з nice=10, за рівних умов.

Окремо слід розглянути планування у системах реального часу, де вимоги до часових характеристик є жорсткими та детермінованими. Системи реального часу поділяють на жорсткі (hard real-time), де пропуск дедлайну є неприпустимим (авіоніка, медичні прилади, промислові контролери), та м’які (soft real-time), де пропуск дедлайну небажаний, але не катастрофічний (відеопотокове мовлення, VoIP). Для жорстких систем реального часу використовуються алгоритми EDF (Earliest Deadline First) та Rate Monotonic Scheduling (RMS). Linux підтримує клас SCHED_FIFO та SCHED_RR для процесів реального часу, де вони завжди матимуть пріоритет над звичайними процесами. Для справжніх жорстких систем реального часу часто використовуються спеціалізовані RTOS (Real-Time Operating System): FreeRTOS, VxWorks, QNX, Zephyr — системи, розраховані на детерміновані та обмежені часи відгуку, часто у вбудованих мікроконтролерних середовищах із жорсткими ресурсними обмеженнями.

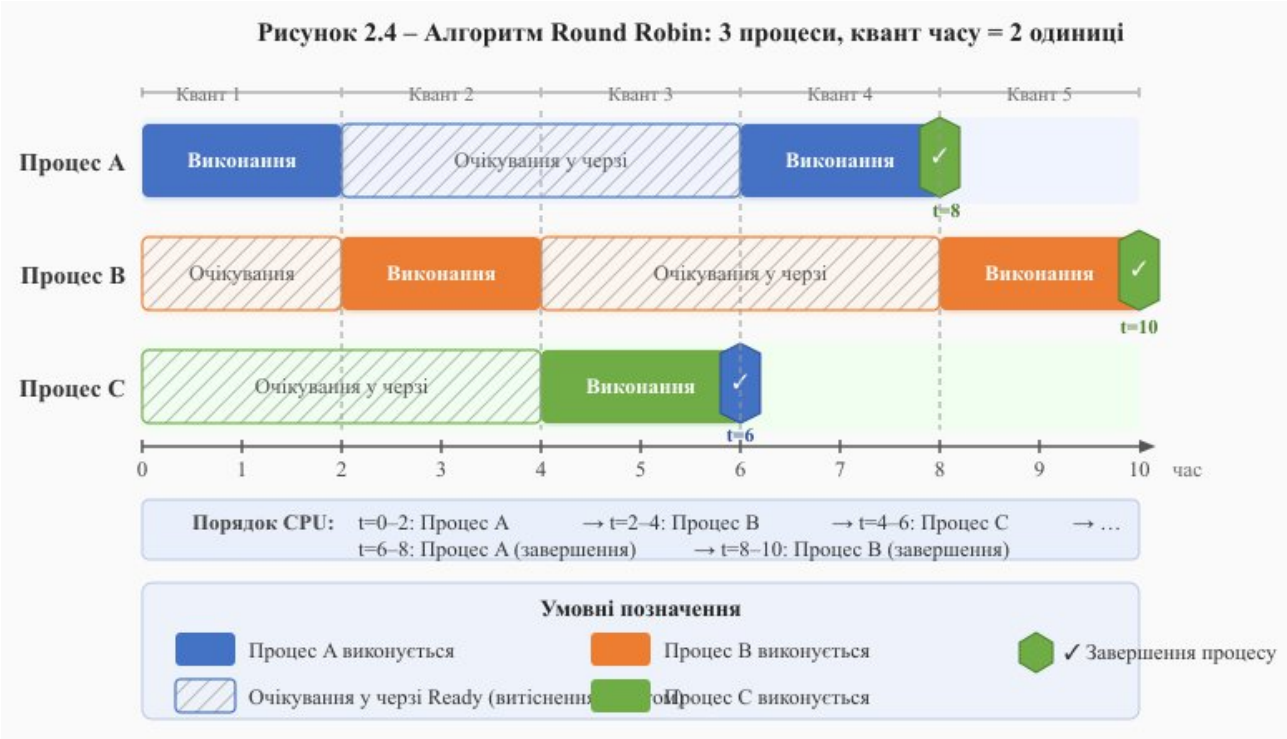

Рисунок 2.4 – Принцип роботи алгоритму Round Robin із квантом часу

На рисунку 2.4 зображено розподіл процесорного часу між трьома процесами (A, B, C) за алгоритмом Round Robin із квантом 2 одиниці часу. Процес A отримує CPU першим (0–2), потім поступається процесу B (2–4), далі C (4–6), після чого знову A (6–8). Червоним позначено інтервали очікування в черзі Ready. Процес C завершується першим у момент t=6, процес A — у t=8, а процес B продовжує виконання. Діаграма наочно демонструє рівномірне чергування і відсутність монополізації CPU.

Важливим аспектом планування у сучасних багатоядерних системах є проблема балансування навантаження (load balancing). Планувальник повинен не лише вирішити, коли виконувати задачу, але й на якому фізичному ядрі процесора. Переміщення задачі між ядрами (міграція) може погіршити продуктивність через «холодний кеш»: нове ядро не має у своєму L1/L2-кеші даних, якими оперує процес, і відповідні кешові рядки доведеться завантажувати з повільнішої пам’яті. Тому сучасні планувальники намагаються зберігати спорідненість процесора (CPU affinity), виконуючи задачу переважно на тому самому ядрі, водночас рівномірно розподіляючи загальне навантаження між ядрами. У Linux системний адміністратор може вручну задати CPU affinity для конкретного процесу командою taskset.

Додаткову складність вносить архітектура NUMA (Non-Uniform Memory Access), що характерна для багатопроцесорних серверних систем. У NUMA-системах кожен процесор (або група процесорів) має власну «локальну» оперативну пам’ять, доступ до якої є суттєво швидшим, ніж до «віддаленої» пам’яті інших вузлів. Планувальник повинен враховувати NUMA-топологію і намагатися виконувати задачу на тому ядрі, що знаходиться у тому ж NUMA-вузлі, де розміщені дані процесу. У Linux управління пам’яттю з урахуванням NUMA здійснюється через підсистему numactl та автоматичні оптимізації у

ядрі. Ігнорування NUMA-топології у великих серверних додатках може призводити до суттєвого падіння продуктивності — до 2–4 разів порівняно з NUMA-оптимізованим виконанням.

2.4. Синхронізація процесів і вирішення проблем паралельного виконання

Паралельне виконання кількох процесів або потоків, що спільно використовують деякі ресурси, неминуче породжує проблеми синхронізації. Головна з них — умова гонитви (race condition) — виникає тоді, коли результат виконання програми залежить від порядку, у якому процеси або потоки отримують доступ до спільних даних. Класичним прикладом є ситуація, коли два потоки одночасно читають значення лічильника, збільшують його на одиницю і записують назад: якщо обидва прочитали однакове початкове значення до того, як перший з них встиг записати результат, підсумкове значення виявиться збільшеним лише на одиницю замість двох. Подібні помилки надзвичайно важко відтворити та відлагодити, оскільки вони проявляються нерегулярно і залежать від точного хронометражу виконання команд на рівні мікроархітектури процесора.

Для запобігання умовам гонитви необхідно забезпечити взаємне виключення (mutual exclusion) при доступі до спільних даних. Ділянка коду, де відбувається доступ до спільного ресурсу і яка не може виконуватись одночасно кількома процесами, називається критичною секцією. Задача синхронізації зводиться до того, щоб у будь-який момент часу лише один процес перебував у критичній секції. Коректне рішення цієї задачі має задовольняти три необхідні властивості. По-перше — взаємне виключення: якщо процес P виконується у критичній секції, жоден інший процес не може перебувати у своїй критичній секції (для того самого ресурсу). По-друге — прогрес: якщо жоден процес не знаходиться у критичній секції і деякий процес хоче туди увійти, то це рішення про те, хто увійде, приймається за кінцевий час, без нескінченного відкладання. По-третє — обмежене очікування: між моментом, коли процес запитав вхід у критичну секцію, і реальним входом інші процеси можуть увійти і вийти звідти лише обмежену (фіксовану) кількість разів.

Найпростішою (і неправильною) спробою вирішення проблеми критичної секції є використання прапорця у спільній пам’яті. Наприклад, процес читає прапорець, переконується що він вільний і встановлює його — проте між читанням і встановленням він може бути перерваний, і інший процес теж прочитає «вільно» і теж встановить прапорець, після чого обидва опиняться у критичній секції. Алгоритм Петерсона (1981 рік) є правильним програмним рішенням задачі взаємного виключення для двох процесів: він використовує два масиви прапорців та одну змінну черги і гарантує усі три необхідні властивості. Однак на сучасних процесорах із переупорядкуванням команд (out-of-order execution) навіть алгоритм Петерсона може давати збої без явних бар’єрів пам’яті (memory barriers). Це підкреслює той факт, що правильна

синхронізація на сучасному обладнанні є складним завданням, що вимагає глибокого розуміння як моделі виконання процесора, так і механізмів ОС.

Для реалізації взаємного виключення та складніших сценаріїв синхронізації розроблено ряд примітивів. Найпростішим є м'ютекс (mutex, mutual exclusion lock) — двійковий замок, що може перебувати у двох станах: вільний або захоплений. Потік, що захоплює м'ютекс, може увійти у критичну секцію; решта потоків, що намагаються захопити захоплений м'ютекс, блокуються до його звільнення. Важлива властивість м'ютексу — «власність»: лише той потік, що захопив м'ютекс, може його звільнити. Це відрізняє м'ютекс від семафора та запобігає ряду помилок, проте робить його непридатним для деяких сценаріїв сигналізації між потоками.

Семафор, введений Е. Дейкстрою у 1965 році, є більш загальним примітивом синхронізації. Він являє собою цілочисельний лічильник, що підтримує дві атомарні операції: wait (також P або down) та signal (також V або up). Операція wait зменшує лічильник на одиницю: якщо він стає від'ємним, потік блокується; якщо залишається невід'ємним — виконання продовжується. Операція signal збільшує лічильник на одиницю: якщо є заблоковані потоки, один з них пробуджується і переводиться у стан Ready. Бінарний семафор (з початковим значенням 1) поводиться аналогічно м'ютексу. Лічильний семафор використовується для управління доступом до пулу однотипних ресурсів: наприклад, семафор зі значенням 10 дозволяє одночасний доступ до ресурсу не більш ніж 10 потокам.

Монітор є вищорівневим об'єктно-орієнтованим примітивом, що об'єднує спільні дані, набір операцій над ними та вбудований механізм взаємного виключення. Лише один потік може виконувати метод монітора водночас — це автоматично забезпечує захист спільних даних. Моніторам також притаманні умовні змінні (condition variables), що дозволяють потоку всередині монітора тимчасово звільнити блокування і чекати настання певної умови. Інший потік може увійти в монітор, змінити стан і сигналізувати про настання умови за допомогою операцій signal або broadcast. Монітори є основою синхронізації у мовах Java (synchronized методи), C# (lock), Python (threading.Condition).

Спін-блокування (spinlock) є особливим видом блокування, що реалізує активне очікування замість переводу потоку у стан блокування. Потік у циклі перевіряє стан блокування («крутиться»), доки воно не звільниться. Перевага спін-блокування — відсутність накладних витрат на переключення контексту при блокуванні і пробудженні, що робить його ефективним для дуже коротких критичних секцій. Недолік — марне споживання процесорного часу при тривалому очікуванні. Спін-блокування широко використовуються у ядрі Linux для захисту даних, до яких відбувається доступ з обробників переривань, де переключення контексту неприпустиме. В обробнику переривань не можна «заснути» (викликати sleep або blocking-примітив), тому лише spinlock є прийнятним варіантом синхронізації у цьому контексті. Атомарні операції (atomic operations) є найлегковагішим механізмом синхронізації: операції типу compare-and-swap (CAS), fetch-and-add та інші виконуються як єдина

неподільна машинна інструкція без будь-яких блокувань. Вони є основою lock-free структур даних (черги, стеки, хеш-таблиці), що забезпечують синхронізацію без традиційних примітивів і можуть давати суттєво кращу продуктивність у висококонкурентних сценаріях.

Таблиця 2.5 – Примітиви синхронізації та сфера їх застосування

Примітив	Тип	Власність	Блокування	Типове застосування
М’ютекс	Бінарний замок	Так (потік-власник)	Ядро ОС (сплячий замок)	Захист критичної секції
Семафор	Лічильниковий	Ні	Ядро ОС	Пул ресурсів; виробник-споживач
Спін-блокування	Бінарний замок	Ні	Активне (busy-wait)	Ядро ОС; короткі секції на SMP
Монітор	Об’єктний	Автоматично	Ядро ОС	Java/C# синхронізація
Умовна змінна	Сигнальний	Ні	Ядро ОС	Очікування умови в моніторі
Атомарні операції	Апаратний	Ні	Немає	Lock-free структури даних

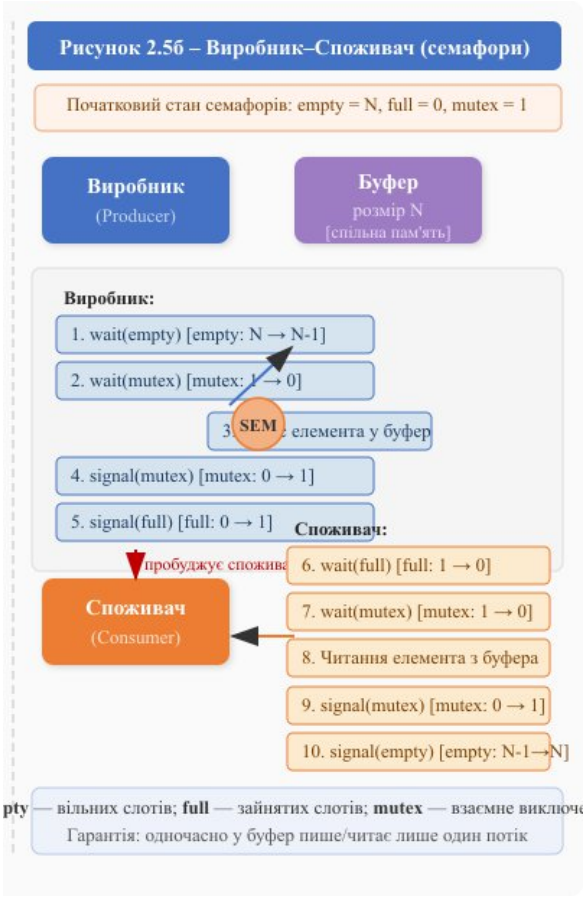

Рисунок 2.5 – Взаємодія потоків «виробник–споживач» через семафор

На рисунку 2.5 показано класичну задачу «виробник–споживач» і її розв’язання через три семафори. Семафор empty відстежує кількість вільних

слотів у буфері (початкове значення N), семафор full — кількість заповнених слотів (початкове значення 0), а mutex забезпечує взаємне виключення при доступі до самого буфера (початкове значення 1). Виробник спочатку перевіряє наявність місця (wait(empty)), потім захоплює м'ютекс, записує елемент і звільняє м'ютекс і сигналізує споживачу (signal(full)). Споживач симетрично чекає на наповнений слот, зчитує елемент і звільняє місце.

Однією з найбільш небезпечних проблем у синхронізації є взаємне блокування, або дедлок (deadlock). Дедлок виникає, коли два або більше процеси чекають на ресурси, що утримуються один одним, і жоден з них не може продовжити виконання. Необхідні умови дедлоку, відомі як умови Коффмана (1971 рік), включають чотири одночасних аспекти: по-перше, взаємне виключення (ресурс може утримуватись лише одним процесом водночас); по-друге, утримання та очікування (процес утримує хоча б один ресурс і одночасно очікує на інші); по-третє, відсутність витіснення (ресурс не може бути примусово забраний у процесу — лише добровільно звільнений); по-четверте, кругове очікування (існує кільцевий ланцюг процесів, де кожен чекає на ресурс, що утримується наступним у ланцюгу).

Якщо хоча б одна з умов Коффмана порушується, дедлок неможливий. Стратегії роботи з дедлоками поділяються на чотири основні класи. Запобігання (Prevention) — проектування системи таким чином, що одна з умов Коффмана структурно неможлива: наприклад, вимагати, щоб процес запитував усі ресурси одночасно (порушення умови «утримання та очікування»). Уникнення (Avoidance) — динамічний аналіз кожного запиту на ресурс і надання його лише тоді, коли це не приведе систему у небезпечний стан; класичний алгоритм — «алгоритм банкіра» Дейкстри. Виявлення та відновлення (Detection and Recovery) — ОС регулярно перевіряє граф розподілу ресурсів на наявність циклів і при виявленні дедлоку завершує або «відкочує» один або кілька процесів («жертв»). Нарешті, «підхід страуса» — просте ігнорування проблеми, застосовне тоді, коли дедлоки трапляються вкрай рідко і ціна їх запобігання вища за ціну перезапуску.

Алгоритм банкіра, запропонований Дейкстрою у 1965 році в контексті операційної системи THE, є класичним алгоритмом уникнення дедлоків. Аналогія з банком полягає в тому, що банкір (ОС) не видає кредит (ресурс), якщо це може призвести до стану, коли банкір не зможе задовольнити жодного клієнта. Система вважається «безпечною», якщо існує порядок виконання всіх процесів, при якому кожен з них може отримати необхідні ресурси і завершити роботу. При надходженні запиту на ресурс алгоритм симулює його надання і перевіряє, чи залишиться система у безпечному стані. Якщо ні — запит відкладається. Хоча алгоритм банкіра є теоретично правильним, практичне його застосування ускладнене тим, що максимально можливі потреби кожного процесу у ресурсах мають бути відомі заздалегідь, що у реальних системах рідко можливо. Тому сучасні ОС загального призначення переважно застосовують підхід виявлення та відновлення або просто ігнорування.

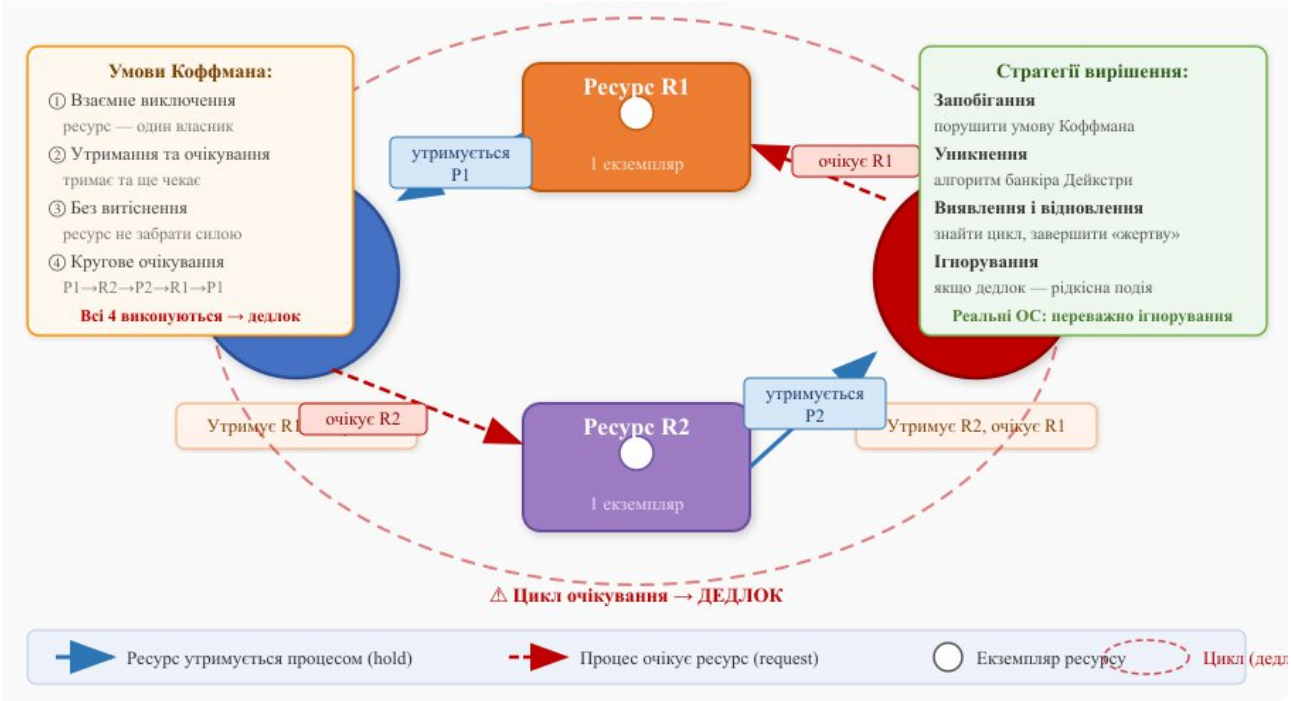

Рисунок 2.6 – Граф розподілу ресурсів: приклад дедлоку між двома процесами

На рисунку 2.6 зображено граф розподілу ресурсів для класичного дедлоку між двома процесами. Процес P1 утримує ресурс R1 (суцільна стрілка від R1 до P1) і очікує на ресурс R2 (пунктирна стрілка від P1 до R2). Процес P2 утримує ресурс R2 і очікує на ресурс R1. Утворюється цикл: P1→R2→P2→R1→P1, що є графічним відображенням взаємного блокування. Жоден з процесів не може продовжити виконання без втручання ОС.

2.5. Вплив нестабільної роботи процесів на безпеку інформаційно-комунікаційних систем

Безпека інформаційно-комунікаційних систем нерозривно пов’язана зі стабільністю та передбачуваністю поведінки процесів. Нестабільна або некоректна робота процесів може стати не лише причиною зниження доступності системи, але й відкрити вектори для активних атак. Умови гонитви, що є передусім джерелом функціональних помилок, можуть також бути цілеспрямовано використані зловмисниками для отримання несанкціонованого доступу до ресурсів.

Так звані TOCTOU-атаки (Time-Of-Check to Time-Of-Use) використовують розрив між моментом перевірки прав доступу до ресурсу і моментом фактичного доступу до нього. Якщо зловмисник може модифікувати об’єкт (наприклад, підмінити файл символічним посиланням) у цьому часовому вікні, він отримує доступ до ресурсу, на який у нього немає прав. Класичний приклад: привілейована програма перевіряє, що файл /tmp/data є безпечним (наприклад, належить поточному користувачу і не є символічним посиланням), але зловмисник встигає замінити його на symlink до /etc/shadow до того, як програма відкриває файл на запис. У результаті привілейована програма записує дані безпосередньо у файл тіньових паролів. Для захисту від TOCTOU

рекомендується: відкривати файловий дескриптор і потім працювати лише через нього (а не через ім'я файлу), використовувати атомарні операції там, де це можливо, та перевіряти права у тому самому системному виклику, що і відкриває ресурс (наприклад, open() з O_NOFOLLOW).

Привілейовані процеси являють собою особливу категорію ризику. Процеси, що виконуються з правами суперкористувача (root у Linux або SYSTEM у Windows), мають необмежений доступ до ресурсів системи: можуть читати та записувати будь-які файли, завантажувати та вивантажувати модулі ядра, змінювати мережеві налаштування тощо. Якщо такий процес містить вразливість — переповнення буфера, помилку форматного рядка (format string), використання пам'яті після звільнення (use-after-free) або некоректну перевірку вхідних даних — зловмисник може скористатися нею для виконання довільного коду з підвищеними привілеями. Це призводить до повної компрометації системи. Саме тому принцип найменших привілеїв (Principle of Least Privilege, PoLP) є фундаментальним у системному проектуванні: кожен процес повинен мати лише той мінімум прав, що необхідний для виконання його функції, і не більше.

Практичною реалізацією PoLP у UNIX є механізм SUID/SGID бітів, а також можливість відмови від привілеїв (drop privileges): привілейований процес виконує частину роботи, що потребує прав root, а потім знижує власні привілеї до рівня звичайного користувача через виклики setuid()/setgid(). Більш сучасним підходом є можливості Linux (capabilities): замість монолітного root-доступу процесу призначаються лише окремі дозволи, наприклад CAP_NET_BIND_SERVICE (прив'язка до портів < 1024) або CAP_SYS_TIME (зміна системного часу). Таким чином, навіть якщо процес буде скомпрометовано, зловмисник отримає лише обмежений набір привілеїв, а не повний контроль над системою.

Ще однією небезпечною ситуацією є некоректне завершення процесів. Якщо процес або потік аварійно завершується, не звільнивши захоплені м'ютекси або семафори, інші процеси, що очікують на ці примітиви синхронізації, зависнуть назавжди — аж до перезапуску системи або примусового завершення усіх зациклених процесів. Таке явище називається «отруєним» блокуванням (poisoned lock або lock poisoning). Подібний сценарій може бути цілеспрямовано використаний для атак типу «відмова у обслуговуванні» (DoS): зловмисник навмисно аварійно завершує ключовий процес у момент, коли він тримає блокування, що призводить до паралічу залежних від нього сервісів. Сучасні POSIX-системи надають механізм robust mutexes, що дозволяє іншим потокам визначити, що власник м'ютексу завершився, і відновити роботу з обробкою помилки, а не зависати. У Java з версії 5.0 введено поняття «отруєного стану» (poisoned state) для ReentrantLock, де потік, що чекає, отримує виняток при загибелі власника.

Сигнали у UNIX-подібних системах є механізмом асинхронного повідомлення процесів і можуть також стати вектором атаки. Операційна система визначає близько 30 стандартних сигналів: SIGTERM (запит на завершення), SIGKILL (примусове завершення, не можна перехопити),

SIGSEGV (помилка сегментації), SIGINT (переривання з клавіатури, Ctrl+C), SIGHUP (сигнал завершення сесії). Якщо зловмисник може надіслати сигнал привілейованому процесу, він може змусити його завершитись, скинути конфігурацію або виконати певну дію, залежно від написаних обробників. Асинхронна природа сигналів у поєднанні з неправильно написаними обробниками може породжувати умови гонитви: якщо обробник сигналу модифікує глобальну структуру даних, до якої водночас отримує доступ основний потік, виникає race condition. Функції, безпечні для виклику зі обробників сигналів, утворюють обмежений перелік (async-signal-safe), і порушення цього правила часто призводить до непередбачуваної поведінки або вразливостей. У Linux переважно рекомендованим підходом є використання signalfd() — перетворення сигналів на події файлового дескриптора, що дозволяє обробляти їх у звичайному потоці коду без асинхронних ризиків.

Сучасні операційні системи впроваджують ряд механізмів ізоляції процесів, що суттєво ускладнюють атаки. У Linux це насамперед простори імен (namespaces), що ізолюють різні аспекти середовища процесу: мережеві інтерфейси (net namespace), файлову систему (mnt namespace), PID-простір (pid namespace), міжпроцесову комунікацію (ipc namespace) тощо. На основі namespaces побудовані контейнери Docker та podman. Групи контролю (cgroups) обмежують споживання ресурсів: CPU, пам’ять, пропускна здатність мережі, кількість процесів — захищаючи від атак типу «виснаження ресурсів».

Профілі seccomp (Secure Computing Mode) дозволяють обмежити набір системних викликів, доступних процесу, до мінімально необхідного. Наприклад, браузер Chromium використовує seccomp-bpf для обмеження системних викликів у процесах рендерингу, що значно зменшує потенційний вплив вразливості у рендерері. AppArmor та SELinux реалізують обов’язкові політики контролю доступу на рівні ядра, що обмежують взаємодію процесів з файловою системою, мережею та іншими ресурсами незалежно від стандартних прав UNIX. У Windows аналогічні функції забезпечуються механізмами Job Objects (обмеження ресурсів для груп процесів), Integrity Levels (рівні цілісності), AppContainer (ізоляція застосунків Windows Store) та Windows Defender Credential Guard.

Журналювання та моніторинг подій процесів є критично важливим елементом виявлення інцидентів та реагування на них. Операційні системи надають засоби відстеження системних викликів: strace та perf у Linux, Process Monitor та ETW (Event Tracing for Windows) у Windows. Системи виявлення вторгнень на рівні хоста (HIDS, наприклад OSSEC, Wazuh) активно аналізують ці журнали для виявлення аномальної поведінки — незвичного набору системних викликів, спроб доступу до заборонених ресурсів, запуску нових процесів з підозрілими параметрами. Зберігання та аналіз журналів підій процесів після факту компрометації дозволяє відтворити хронологію атаки, визначити початковий вектор проникнення та оцінити реальний масштаб збитків.

Комплексний захист, що поєднує ізоляцію процесів, мінімізацію привілеїв, фільтрацію системних викликів та постійний аудит, реалізує

принцип глибокого захисту (Defense in Depth). Цей принцип передбачає, що жоден окремий механізм захисту не є абсолютним, і компрометація одного рівня не повинна автоматично означати повну компрометацію системи. Наприклад, якщо зловмисник успішно використав вразливість у вебсервері, механізм seccomp обмежить перелік доступних системних викликів, SELinux заблокує доступ до файлів поза дозволеним контекстом, а cgroup обмежить споживання пам'яті та CPU. Поєднання цих механізмів вимагає від зловмисника значно більшої кількості послідовних успішних кроків для досягнення своєї мети, що суттєво підвищує вартість атаки і дає захисникам більше часу для виявлення та реагування.

Таблиця 2.6 – Механізми ізоляції та безпеки процесів у Linux та Windows

Механізм	ОС	Принцип роботи	Захист від
Namespaces	Linux	Ізоляція системних ресурсів (PID, net, mnt, ipc…)	Несанкціонованого доступу між контейнерами
cgroups	Linux	Обмеження споживання CPU, пам'яті, I/O	Атак виснаження ресурсів (DoS)
seccomp-bpf	Linux	Фільтрація системних викликів за BPF-правилами	Використання вразливостей через сисколи
AppArmor / SELinux	Linux	MAC-політики доступу до файлів та ресурсів	Ескалації привілеїв; горизонтального переміщення
Capabilities	Linux	Фрагментація root-привілеїв на окремі дозволи	Надмірних прав у сервісних процесах
Job Objects	Windows	Групове обмеження ресурсів для набору процесів	Виснаження ресурсів
Integrity Levels	Windows	Мандатні рівні довіри (Low/Medium/High/System)	Несанкціонованого доступу між рівнями
AppContainer	Windows	Ізоляція застосунків Windows Store	Компрометації системи через застосунок

Управління процесами та потоками є одним з ключових механізмів операційної системи, що визначає як функціональність, так і безпеку всієї обчислювальної платформи. Розуміння моделі процесу, принципів планування, механізмів синхронізації та їх зв'язку з проблематикою безпеки є фундаментальним для підготовки фахівців у сфері кіберзахисту та системного адміністрування.

Модель процесу описує динамічну сутність, що включає стан, контекст та ідентифікатори. Блок управління процесом (PCB) містить повну інформацію, необхідну ядру для управління кожним процесом і перемикання між ними. Переходи між станами (New → Ready → Running → Waiting → Terminated) відображають реакцію ОС на системні події — диспетчеризацію, переривання та завершення операцій введення-виведення. Ієрархія процесів, механізм fork()/exec() та стан «зомбі» є невід'ємними аспектами практичного управління процесами у UNIX-подібних системах.

Потоки виконання як легковагові процеси забезпечують ефективний внутрішньопроцесний паралелізм за значно менших накладних витрат на перемикання. Три моделі реалізації (KLT, ULT, M:N) мають різні компроміси між продуктивністю, паралелізмом та простотою управління. Алгоритми планування — від простого FCFS до складного MLFQ та CFS — визначають, як саме розподіляється процесорний час між конкурентними задачами. Вибір алгоритму та налаштування його параметрів (квант часу, пріоритети) суттєво впливають на час відгуку, пропускну здатність та справедливість системи.

Синхронізація є необхідним, але потенційно небезпечним аспектом паралельного програмування. Умови гонитви, дедлоки, «отруєні» блокування — усе це реальні проблеми, що мають як функціональні, так і безпекові наслідки. TOCTOU-атаки, ескалація привілеїв через вразливості у привілейованих процесах, DoS через «отруєні» блокування — конкретні вектори загроз, що беруть початок у механізмах управління процесами. Сучасні ОС відповідають на ці виклики комплексом засобів ізоляції (namespaces, cgroups, seccomp, Integrity Levels), аудиту та обов’язкового контролю доступу (SELinux, AppArmor), що реалізують принцип глибокого захисту і значно підвищують стійкість систем до атак.

ТЕМА 3. УПРАВЛІННЯ ПАМ'ЯТТЮ ТА МЕХАНІЗМИ ЗАХИСТУ

3.1. Організація пам'яті в обчислювальних системах

Для розуміння принципів управління пам'яттю необхідно чітко розрізнити кілька рівнів її організації: фізичну пам'ять як реальний апаратний ресурс; віртуальну пам'ять як абстракцію, яку операційна система надає процесам; та простір підкачки (swap), що дозволяє розширити доступний адресний простір за рахунок дискового сховища. Кожен рівень відіграє власну роль і підпорядкований різним принципам роботи, однак усі вони взаємопов'язані та утворюють єдину ієрархічну систему зберігання виконуваних даних. Вибір архітектурних рішень щодо організації пам'яті безпосередньо впливає на безпеку, стабільність і продуктивність всієї інформаційно-комунікаційної системи.

Фізична пам'ять являє собою оперативну пам'ять (RAM), безпосередньо доступну процесору. З апаратної точки зору вона є послідовним масивом байтів, кожен з яких має унікальну фізичну адресу. Сучасні серверні системи можуть містити від кількох гігабайт до кількох терабайт оперативної пам'яті, і управління всім цим ресурсом здійснюється виключно ядром операційної системи. Ядро зберігає внутрішні структури даних, що описують, які ділянки фізичної пам'яті зайняті, яким процесом і з якою метою. Пряма робота з фізичними адресами у сучасних системах доступна лише ядру — прикладні програми позбавлені безпосереднього доступу до фізичних адрес, що є одним із базових принципів ізоляції.

Віртуальна пам'ять є центральною абстракцією, яку операційна система надає кожному процесу. Процесу надається ілюзія власного великого та безперервного адресного простору, що починається з нульової адреси і простягається до максимально можливої, яка визначається розрядністю архітектури. На 64-розрядній системі теоретичний розмір віртуального адресного простору складає 2^{+8} байт; архітектура x86-64 практично обмежує його 48 або 57 значущими бітами залежно від режиму сторінкування. Ключова перевага цієї абстракції — ізоляція процесів: звернення процесу A до адреси 0x1000 і звернення процесу B до тієї самої адреси 0x1000 відповідають різним ділянкам фізичної пам'яті, оскільки кожен процес має власну таблицю сторінок.

Простір підкачки (swap) є третім рівнем ієрархії і являє собою виділений розділ або файл на диску, що використовується ядром для тимчасового зберігання сторінок пам'яті з низькою активністю використання. Коли фізичної пам'яті не вистачає, ядро виконує операцію витіснення (page-out): переміщує відповідні сторінки у swap, звільняючи фізичні кадри для активних процесів. При подальшому зверненні до витісненої сторінки відбувається зворотна операція (page-in). Зловживання swap-простором суттєво знижує продуктивність системи через різницю у швидкості доступу між RAM та диском, що становить два-три порядки величини.

Ієрархія пам'яті у сучасних обчислювальних системах охоплює також кеш-пам'ять процесора трьох рівнів (L1, L2, L3), яка є значно швидшою за

RAM, але значно меншою за обсягом. Управління кешами процесора здійснюється апаратно і перебуває поза прямим контролем операційної системи, хоча ОС враховує особливості кешування при прийнятті рішень щодо розміщення даних у пам'яті. Рисунок 3.1 унаочнює цю ієрархію у вигляді піраміди, де вершина відповідає найшвидшому та найменшому рівню, а основа — найбільшому та найповільнішому.

Рисунок 3.1 — Ієрархія пам'яті в обчислювальній системі

На рисунку 3.1 подано піраміду ієрархії пам'яті. Кожен рівень підписано з зазначенням типового діапазону місткості та часу доступу, що унаочнює обернену залежність між цими характеристиками. Ліва вертикальна стрілка позначає збільшення швидкості доступу до вершини піраміди, права — збільшення місткості до основи. Бічними дужками виділено дві зони управління: рівні RAM і swap — під керуванням операційної системи; рівні кешів і регістрів — під керуванням апаратного забезпечення CPU.

Таблиця 3.1 — Порівняння рівнів ієрархії пам'яті

Рівень	Час доступу, нс	Типова місткість	Управління
Регістри CPU	< 1	Десятки байт	Процесор / компілятор
Кеш L1	2–5	32–64 КБ	Апаратне (CPU)
Кеш L2	~10	256 КБ – 4 МБ	Апаратне (CPU)
Кеш L3	30–40	8–64 МБ	Апаратне (CPU)
RAM	50–100	4–256 ГБ	Ядро ОС
Swap	10 000–100 000	1–64 ГБ	Ядро ОС
SSD / HDD	> 100 000	Сотні ГБ – ТБ	ОС / файлова система

Таблиця 3.1 підтверджує, що різниця у часі доступу між рівнями ієрархії становить від одного до п'яти порядків величини. Завдання операційної системи полягає у забезпеченні ефективного використання наявних рівнів так, щоб дані з високою частотою звернень залишалися у швидкій пам'яті, а рідко використовувані — переміщалися на нижчі рівні.

3.2. Механізми адресації: сегментація та сторінкування

Перетворення віртуальної адреси у фізичну — центральна операція підсистеми пам'яті, що виконується при кожному зверненні процесора до пам'яті. В архітектурах процесорів було розроблено два основні механізми адресації — сегментацію та сторінкування — які вирішують одне ключове завдання: відображення логічного адресного простору програми на фізичну пам'ять. Обидва механізми принципово відрізняються за підходом до розбивки адресного простору на блоки, що тягне за собою різні наслідки для продуктивності, гнучкості захисту та складності управління.

Сегментація як механізм організації пам'яті полягає в поділі адресного простору на логічно значущі частини — сегменти — кожен з яких характеризується базовою фізичною адресою та максимально дозволеним розміром (лімітом). Типові сегменти процесу — сегмент коду, сегмент ініціалізованих даних, сегмент стека та купи. Процесор при кожному зверненні до пам'яті перевіряє дотримання лімітів відповідного сегмента і у разі порушення генерує апаратний виняток, що є засобом апаратного захисту від несанкціонованого доступу. В архітектурі IA-32 (x86) сегментація присутня на апаратному рівні та є обов'язковим елементом механізму адресації. Проте сучасні операційні системи — Linux, Windows, macOS — фактично нейтралізують сегментацію як інструмент захисту: всі сегменти налаштовуються з нульовою базою та максимальним лімітом, а захист реалізується виключно через сторінкування.

Сторінкування є домінуючим механізмом управління пам'яттю у всіх сучасних операційних системах. Принцип полягає в тому, що вся пам'ять — як фізична, так і віртуальна — поділяється на блоки фіксованого розміру. Блоки у віртуальному просторі називаються сторінками (pages), а відповідні їм блоки у фізичній пам'яті — кадрами (page frames). Типовий розмір сторінки у сучасних системах складає 4 КБ; апаратне забезпечення також підтримує «великі сторінки» (huge pages) розміром 2 МБ або 1 ГБ, що застосовуються в задачах з інтенсивною роботою з пам'яттю (бази даних, гіпервізори) для зниження кількості промахів TLB. Відповідність між сторінками і кадрами зберігається в структурі даних — таблиці сторінок (page table), що підтримується ядром ОС.

Кожен запис таблиці сторінок (PTE — Page Table Entry) містить номер фізичного кадру та набір прапорців: P (Present) — сторінка присутня у RAM; R/W — дозволений запис; U/S — доступ з рівня користувача; NX (No-Execute, або XD у термінології AMD) — виконання коду з цієї сторінки заборонено. Процесор автоматично перевіряє ці прапорці при кожному зверненні до пам'яті: спроба виконання коду зі сторінки з встановленим NX-бітом або запис у

сторінку «лише читання» негайно призведуть до апаратного виключення (page fault або general protection fault), яке ядро ОС оброблятиме відповідно до контексту. Саме ці прапорці є апаратною основою багатьох механізмів безпеки, розглянутих у підрозділі 3.5.

Для прискорення трансляції адрес процесори використовують спеціальний апаратний кеш — TLB (Translation Lookaside Buffer), що зберігає декілька сотень або тисяч нещодавно використаних відповідностей «віртуальна сторінка → фізичний кадр». При влученні в TLB (TLB hit) трансляція виконується за один такт без звернення до таблиці сторінок у пам'яті; при промаху (TLB miss) відбувається так зване «обходження таблиць» (page table walk). При переключенні контексту між процесами ядро або інвалідує весь TLB, або позначає записи ідентифікатором процесу (PCID — Process-Context Identifier у архітектурі Intel), щоб уникнути некоректного перетину трансляцій різних процесів.

Рисунок 3.2 ілюструє механізм дворівневої трансляції адрес для 32-розрядної архітектури IA-32, яка є найбільш прозорою для демонстрації принципу. Верхня частина рисунка показує розбиття 32-розрядної віртуальної адреси на три поля: 10 біт індексу в каталозі сторінок (Page Directory), 10 біт індексу в таблиці сторінок (Page Table) та 12 біт зміщення (Offset), що визначає позицію всередині 4-КБ сторінки. Далі показано послідовний пошук по ієрархії від регістра CR3 (вказівника на каталог сторінок) через каталог — до запису PTE, з якого зчитується номер фізичного кадру і прапорці захисту. Фізична адреса формується конкатенацією 20-бітного номера кадру і 12-бітного зміщення. Рисунок також містить примітку щодо структури таблиць у 64-розрядному режимі (4- та 5-рівнева ієрархія).

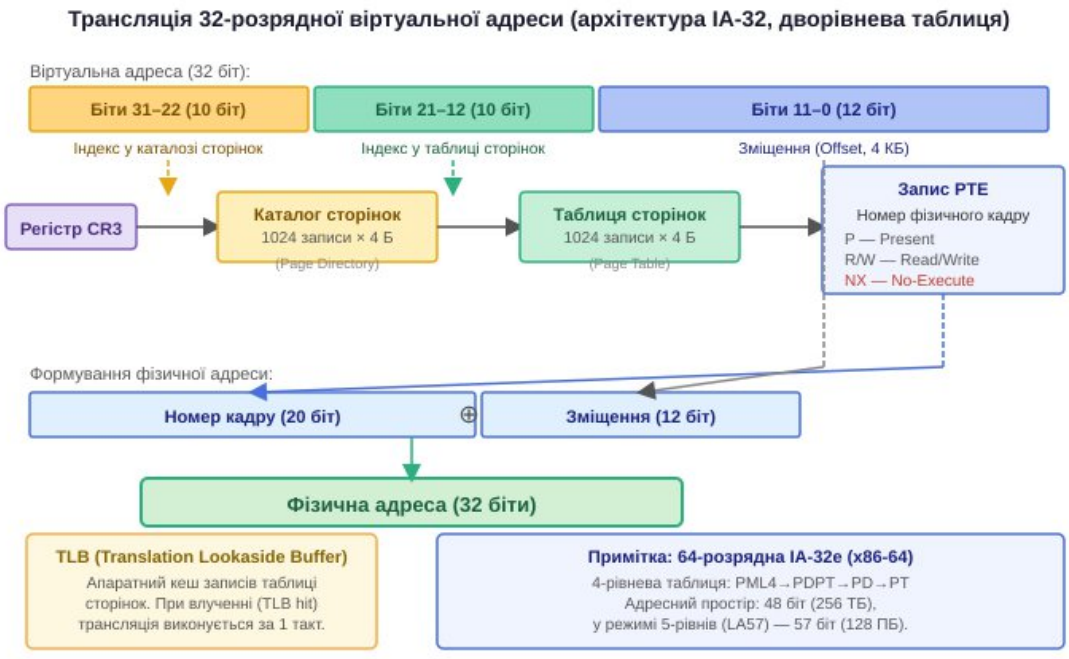

Рисунок 3.2 — Трансляція 32-розрядної віртуальної адреси (IA-32, дворівнева таблиця сторінок)

Таблиця 3.2 — Порівняння механізмів сегментації та сторінкування

Характеристика	Сегментація	Сторінкування
Розмір блоку	Змінний (задається програмістом)	Фіксований (4 КБ, 2 МБ, 1 ГБ)
Логічна структура	Відображає логіку програми	Прозора для програми
Тип фрагментації	Зовнішня (між сегментами)	Внутрішня (в межах сторінки)
Гранулярність захисту	Рівень сегмента	Рівень сторінки (4 КБ)
Підтримка swap	Складна (блоки різного розміру)	Однорідна (ідентичні блоки)
Застосування в Linux	Мінімальне (формально присутня)	Основний механізм захисту

Як видно з таблиці 3.2, перевага сторінкування над сегментацією у сучасних системах визначається насамперед однорідністю блоків пам'яті: оскільки всі сторінки мають однаковий розмір, операції підкачки та управління фізичними кадрами суттєво спрощуються. Крім того, захист на рівні окремих 4-КБ сторінок є значно гнучкішим, ніж захист цілих сегментів: наприклад, сторінки коду можна позначити як «лише читання та виконання» (R-X), а сторінки даних — як «читання та запис без виконання» (RW-), що є основою механізму DEP, детально розглянутого у підрозділі 3.5.

У 64-розрядних системах (архітектура x86-64) таблиці сторінок мають чотири рівні ієрархії: PML4 (Page Map Level 4), PDPT (Page Directory Pointer Table), PD (Page Directory) та PT (Page Table). Кожен рівень займає один кадр (4 КБ) і містить 512 записів по 8 байт, що відповідає 9-бітному індексу. Сумарно $4 \times 9 = 36$ бітів адресують сторінки, і ще 12 бітів — зміщення всередині сторінки, що дає 48-бітний адресний простір (256 ТБ). У режимі 5-рівневих таблиць (LA57), підтримуваному сучасними процесорами Intel і AMD, адресний простір розширюється до 57 біт (128 ПБ). Ієрархічна структура таблиць дозволяє виділяти лише ті частини таблиці, які реально використовуються процесом, що є критично важливим при великому розмірі адресного простору.

3.3. Організація пам'яті в операційній системі Linux

Ядро Linux реалізує розглянуті в попередніх підрозділах принципи управління пам'яттю за допомогою добре структурованої та широко задокументованої підсистеми. Знання архітектури цієї підсистеми є необхідним для аналізу поведінки програм, діагностики проблем та розуміння механізмів безпеки ОС. У даному підрозділі розглянуто основні структури даних ядра, організацію адресного простору процесу, алокатори фізичної пам'яті та ключові механізми оптимізації.

Кожен процес у Linux описується в ядрі структурою task_struct, невід'ємною частиною якої є mm_struct — дескриптор адресного простору процесу. Структура mm_struct містить, зокрема: вказівник pgd на таблицю сторінок верхнього рівня (завантажується у регістр CR3 при переключенні контексту); дерево та зв'язаний список структур vm_area_struct (VMA), кожна з яких описує одне неперервне відображення у віртуальному адресному просторі; а також лічильники статистики використання пам'яті (VmRSS, VmSize та інші). Кожна VMA характеризується початковою і кінцевою адресою, набором прав доступу (читання, запис, виконання), типом відображення (анонімне або файлове) та прапорцями поведінки. Ядро перебирає список VMA під час обробки page fault для визначення законності звернення.

Організацію адресного простору процесу в GNU/Linux (64-розрядна система) ілюструє рисунок 3.3. У нижній частині адресного простору (адреси, близькі до нуля) розміщено захищену NULL-зону — ділянку без жодних відображень, звернення до якої спричиняє page fault. Далі вгору розташовуються: сегмент коду .text (права R-X: читання і виконання, запис заборонено), сегменти .data та .bss (ініціалізовані та неініціалізовані глобальні змінні, права RW-), купа (heap), що росте у бік збільшення адрес від значення, яке підтримується системним викликом brk, та область mmap із відображеннями спільних бібліотек і анонімними відображеннями. Стек (stack) розміщується у верхній частині простору користувача і росте у бік зменшення адрес. Верхня половина адресного простору зарезервована для ядра і недоступна з режиму користувача — відповідні PTE мають прапорець U/S=0.

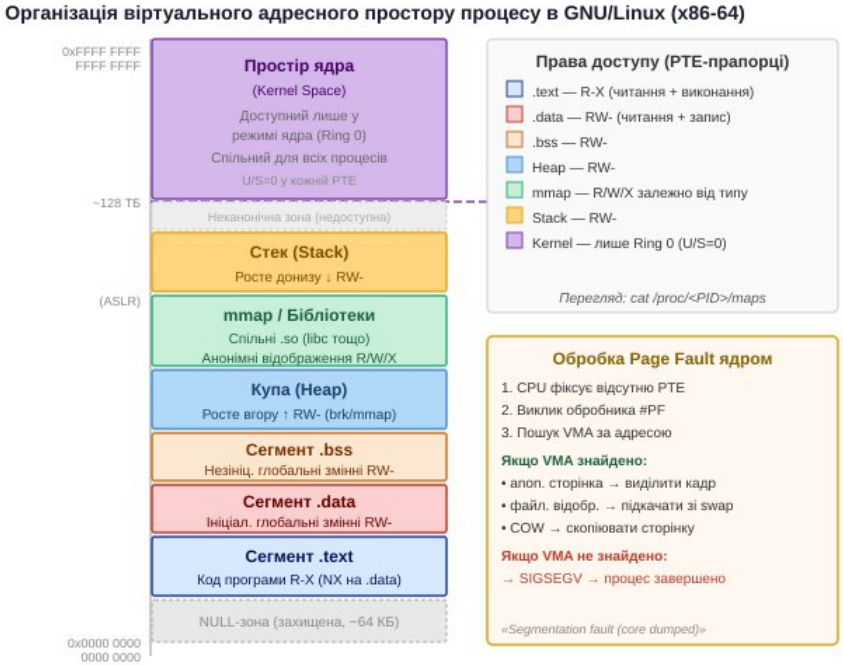

Рисунок 3.3 — Організація віртуального адресного простору процесу в GNU/Linux (x86-64)

Рисунок 3.3 містить вертикальну схему адресного простору з позначенням усіх основних сегментів, їх прав доступу та напрямків росту динамічних ділянок. Пунктирна горизонтальна лінія чітко розмежовує простір ядра (верхня половина) та простір користувача (нижня половина). Права частина рисунка містить дві окремих панелі: легенду прав доступу (R/W/X) для кожного сегмента та схему обробки page fault ядром з переліком можливих сценаріїв (підкачка анонімної сторінки, підкачка з файлу, copy-on-write, SIGSEGV при виході за межі VMA).

Механізм ледачого виділення пам'яті (lazy allocation, demand paging) є одним із ключових принципів роботи підсистеми пам'яті Linux і дозволяє суттєво скоротити реальне споживання фізичної пам'яті. Під час виклику mmap або розширення купи через brk ядро лише реєструє нову VMA, не виділяючи фізичних кадрів. Фізична пам'ять виділяється пізніше — при першому зверненні до кожної сторінки, коли процесор генерує page fault. Ядро у цей момент виділяє вільний фізичний кадр, ініціалізує його нулями (щоб унеможливити витік даних попереднього процесу) та вносить новий запис у таблицю сторінок процесу. Алгоритм обробки page fault показано на рисунку 3.4.

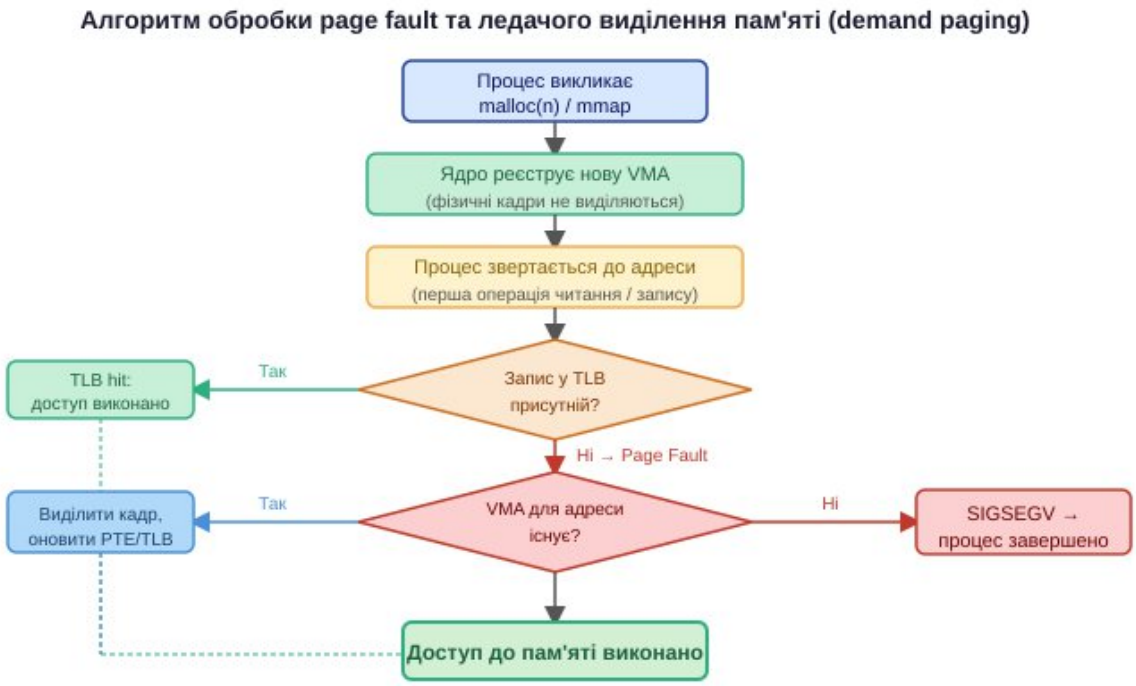

Рисунок 3.4 — Алгоритм обробки page fault та ледачого виділення пам'яті

Блок-схема на рисунку 3.4 відображає послідовність подій від виклику malloc до фактичного виконання доступу. На верхньому рівні показано виклик malloc/mmap та реєстрацію VMA без виділення кадрів. При першому зверненні процесора ядро перевіряє наявність запису в TLB: при влученні (TLB hit) доступ виконується негайно (ліва гілка). При промаху (TLB miss) виникає page fault; ядро перевіряє наявність VMA для даної адреси. Якщо VMA знайдено —

виконується виділення кадру та оновлення PTE/TLB (ліва нижня гілка). Якщо VMA відсутня — ядро надсилає процесу сигнал SIGSEGV і завершує його (права гілка). Усі законні гілки завершуються блоком «Доступ до пам'яті виконано».

Механізм copy-on-write (COW) є оптимізацією, що безпосередньо базується на demand paging і особливо важливий для роботи системного виклику fork. Під час fork батьківський та дочірній процеси отримують спільні фізичні кадри для всіх сторінок пам'яті, позначених у таблицях обох процесів як «лише читання». У момент, коли один із процесів виконує спробу запису до такої сторінки, page fault сигналізує ядру про необхідність створення фізичної копії відповідного кадру; лише після цього запис виконується у нову копію. Завдяки COW виклик fork є надзвичайно швидкою операцією навіть для процесів, що займають гігабайти пам'яті, оскільки реальне копіювання кадрів відкладається до моменту фактичного запису.

Підсистема управління пам'яттю Linux включає два основних алокатори фізичних кадрів. Buddy-система виділяє неперервні блоки кадрів у степенях двійки: від одного кадру (4 КБ) до 1024 кадрів (4 МБ). Назва алокатора пов'язана з алгоритмом злиття: при звільненні блоку система перевіряє вільність сусіднього блоку тієї самої розрядності («buddy»), і якщо він вільний — зливає їх у блок подвійного розміру. Це ефективно знижує зовнішню фрагментацію фізичної пам'яті. Slab-алокатор (у сучасних ядрах — SLUB), побудований поверх Buddy-системи, призначений для ефективного виділення численних дрібних об'єктів ядра фіксованого розміру: структур task_struct, inode, dentry, vm_area_struct тощо. Він заздалегідь виділяє «плити» (slabs) пам'яті та організовує кеші об'єктів, скорочуючи накладні витрати на виділення та звільнення. Архітектуру рівнів підсистеми управління пам'яттю ілюструє рисунок 3.5.

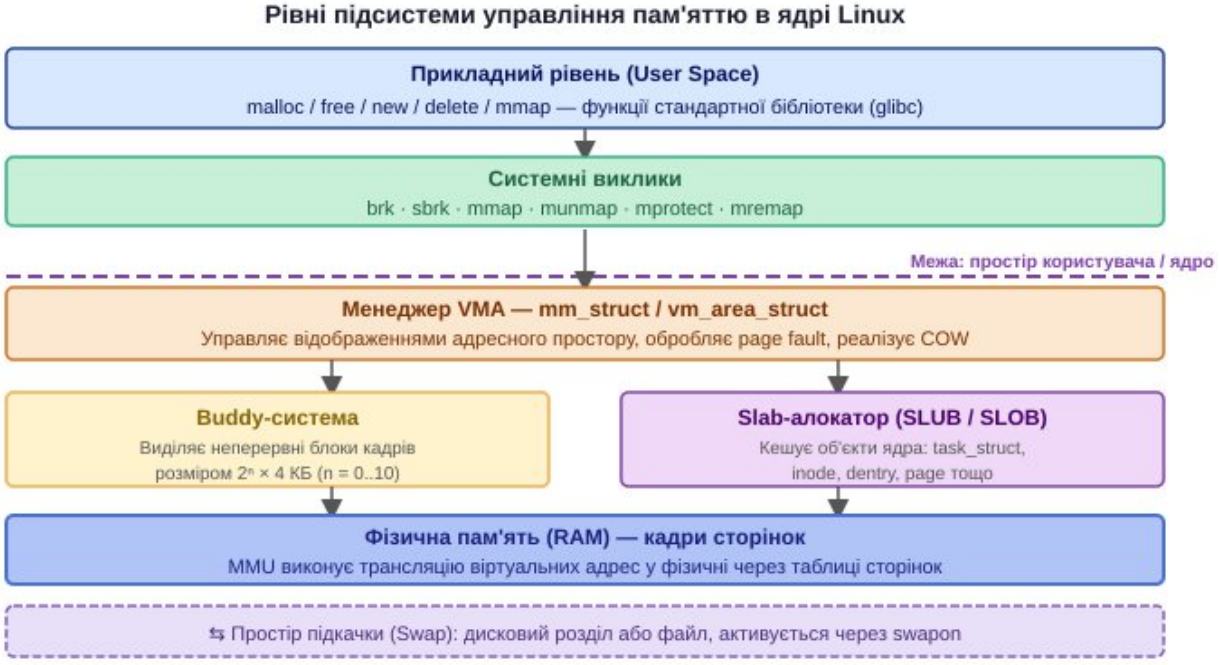

Рисунок 3.5 — Рівні підсистеми управління пам'яттю в ядрі Linux

Рисунок 3.5 відображає стекову архітектуру підсистеми пам'яті. Горизонтальна пунктирна лінія чітко відокремлює простір користувача (верхні два рівні) від простору ядра (нижні рівні). На прикладному рівні розміщено функції стандартної бібліотеки malloc/free/mmap; нижче — системні виклики як інтерфейс між режимами; далі — менеджер VMA, що управляє відображеннями та обробляє page fault; на одному рівні розміщено Buddy-систему та Slab-алокатор; і в основі — рівень фізичної пам'яті (RAM), де виконується апаратна трансляція адрес через MMU. Пунктирна стрілка внизу позначає зв'язок із простором підкачки.

Механізм відображення файлів у пам'ять (memory-mapped files) через системний виклик mmap є важливим елементом архітектури підсистеми пам'яті Linux. Після відображення вміст файлу безпосередньо доступний через адреси у віртуальному просторі процесу, а ядро ефективно управляє підкачкою сторінок через сторінковий кеш (page cache). Якщо кілька процесів відображають один і той самий файл з однаковими правами, ядро надає їм спільні фізичні кадри, що усуває дублювання даних. Саме через mmap завантажуються розділені бібліотеки (.so файли): всі процеси, що використовують libc.so, поділяють одні й ті самі кадри з кодом бібліотеки в режимі R-X, що суттєво економить фізичну пам'ять у багатопроцесному середовищі.

Для моніторингу стану підсистеми пам'яті в середовищі GNU/Linux доступний широкий набір діагностичних інструментів. Базова статистика використання RAM та swap виводиться командою free; деталізовану статистику зі статистикою сторінкових операцій надає vmstat; перелік VMA конкретного процесу з адресами та правами доступу відображається через псевдофайл /proc/PID/maps або утиліту pmap; зведена статистика пам'яті процесу (VmRSS, VmSize, VmSwap) доступна через /proc/PID/status. Перелічені інструменти є незамінними при налагодженні програм та аналізі поведінки системи під навантаженням.

Практичний приклад

Перевірка стану пам'яті та swap у системі:

```
$ free -h
```

Деталізована статистика (у тому числі сторінкові операції si/so):

```
$ vmstat 1 5
```

Відображення VMA конкретного процесу (підставити реальний PID):

```
$ cat /proc/$(pgrep firefox | head -1)/maps
$ pmap -x $(pgrep firefox | head -1)
```

Зведена статистика пам'яті процесу:

```
$ grep -i vm /proc/$(pgrep firefox | head -1)/status
```

Стан slab-кешів ядра (потребує привілеїв root):

```
$ sudo slabtop
```

3.4. Принципи виділення, звільнення та ізоляції пам'яті

Розглянуті в попередньому підрозділі структури ядра Linux реалізують три ключові функції підсистеми управління пам'яттю: виділення пам'яті процесам, її звільнення після завершення роботи та ізоляцію адресних просторів. Кожна з цих функцій потребує балансу між продуктивністю, ощадливим використанням фізичних ресурсів та гарантованою безпекою.

Виділення пам'яті на рівні прикладних програм здійснюється через функції стандартної бібліотеки C — malloc, calloc, realloc та їх аналоги у мовах вищого рівня. Ці функції є обгорткою над системними викликами brk/sbrk та mmap, доповненими власною логікою управління купою: підтриманням списків вільних блоків, стратегіями розподілу (first fit, best fit), злиттям сусідніх вільних блоків. Важливою властивістю є вже розглянуте ледаче виділення: виклик malloc резервує віртуальний простір (реєструє VMA), але фізичні кадри виділяються ядром лише при першому зверненні до кожної сторінки. Це дозволяє програмам ефективно резервувати великі буфери, сплачуючи «вартість» фізичної пам'яті лише за реально використані частини.

Звільнення пам'яті у мові C виконується через функцію free, яка повертає блок у внутрішні структури менеджера купи. Слід відзначити, що free у більшості випадків не повертає пам'ять безпосередньо ядру через munmap — натомість менеджер купи зберігає звільнені блоки у власних списках для повторного використання, звертаючись до munmap лише при звільненні великих блоків або при наявності достатньо великої вільної ділянки на «кінці» купи. Після завершення процесу ядро автоматично і беззворотно звільняє всю пам'ять: перебирає список VMA у mm_struct, знімає всі відображення, повертає фізичні кадри алокатору та видаляє таблицю сторінок. Для спільних файлових відображень фізичні кадри звільняються лише тоді, коли посилань на відповідний файл у page cache не залишається.

Ізоляція адресних просторів є фундаментальним механізмом безпеки операційної системи. Кожен процес у Linux має власну таблицю сторінок (на яку вказує поле pgd у mm_struct), і при переключенні контексту ядро завантажує її фізичну адресу у регістр CR3, змінюючи тим самим «об'єктив», через який процесор перетворює адреси. Внаслідок цього процес не може прочитати або модифікувати пам'ять іншого процесу: звернення до будь-якої адреси, не представленої у власній таблиці сторінок процесу, призведе до page fault, і ядро завершить процес сигналом SIGSEGV. Це є апаратно гарантованою ізоляцією: навіть некоректно написана програма не здатна порушити пам'ять іншого процесу без явного використання механізмів міжпроцесного спілкування.

Механізм OOM Killer (Out-Of-Memory Killer) є засобом останньої лінії захисту стабільності системи у критичних ситуаціях вичерпання пам'яті. Коли вся фізична пам'ять і swap-простір вичерпані, а черговий запит на виділення не може бути задоволений, ядро активує OOM Killer, який обирає один із запущених процесів і примусово завершує його, звільняючи пам'ять. Критерій вибору «жертви» визначається балом oom_score, який нараховується на основі

обсягу спожитої пам'яті, часу роботи процесу, його привілеїв та ряду інших факторів. Системні процеси та процеси з oom_score_adj = −1000 захищені від вибору OOM Killer. Адміністратор може налаштувати oom_score_adj для критичних служб, щоб OOM Killer у першу чергу завершував менш важливі процеси.

Оскільки ізоляція не дозволяє процесам безпосередньо обмінюватися пам'яттю, ядро Linux надає явні механізми міжпроцесного спілкування (IPC). Найшвидшим з них є спільна пам'ять (shared memory), яка реалізується через mmap з прапорцем MAP_SHARED або через POSIX-об'єкти shm_open: обидва процеси отримують відображення одних і тих самих фізичних кадрів у свої адресні простори. Після встановлення відображення обмін даними відбувається без участі ядра (оскільки обидва процеси читають і пишуть до тих самих фізичних кадрів), однак потребує явної синхронізації (семафори, мютекси) для запобігання умовам гонитви.

3.5. Проблеми безпеки пам'яті та сучасні механізми захисту

Вразливості, пов'язані з некоректним управлінням пам'яттю, формують одну з найбільших і найстабільніших категорій загроз безпеці програмного забезпечення. За даними Microsoft Security Response Center, близько 70% виправлених CVE щороку пов'язано саме з помилками управління пам'яттю. Розуміння природи цих вразливостей та механізмів захисту від них є обов'язковим для фахівців у галузі безпеки ІКС, оскільки більшість гучних порушень систем безпеки у серверній та вбудованій інфраструктурі використовує саме вразливості пам'яті.

Переповнення буфера (buffer overflow) є, мабуть, найбільш класичною вразливістю, відомою з публікації Morica 1988 року. Вона виникає, коли програма записує до масиву (буфера) більше даних, ніж він може вмістити, а зайві байти перезаписують сусідні ділянки пам'яті. Особливо небезпечне переповнення стекових буферів: на стеку поруч із локальними змінними зберігається адреса повернення з функції (return address). Зловмисник, контролюючи вхідні дані, що спричиняють переповнення, може перезаписати цю адресу значенням за своїм вибором — наприклад, адресою ін'єктованого шкідливого коду (shellcode). Коректно складена вхідна послідовність, що містить і корисне навантаження, і сфальсифіковану адресу повернення, отримала назву «exploit». Наведений нижче фрагмент коду демонструє типовий сценарій вразливості.

Приклад коду C: вразливість переповнення стекового буфера (gets)

```c
#include <stdio.h>

void vulnerable_function() {
    char buffer[64];
    gets(buffer);  /* gets() не перевіряє довжину — вразлива! */
    printf("Hello, %s\n", buffer);
}
```

```
/* Безпечний варіант: */
/* fgets(buffer, sizeof(buffer), stdin); */

/* При вводі понад 64 байти gets() перезаписує адресу */
/* повернення на стеку -> виконання довільного коду.    */
/* GCC попереджає: warning: 'gets' is deprecated        */
```

Уразливість «використання після звільнення» (use-after-free, UAF) виникає, коли програма після виклику free зберігає вказівник на звільнений блок і в подальшому звертається до нього. Після звільнення блок може бути перевиділено для іншого об'єкта; якщо зловмисник може вплинути на вміст цього нового об'єкта (наприклад, через мережевий протокол або файловий формат), він отримує можливість «підмінити» будь-які поля вихідного об'єкта. Особливо небезпечне UAF у мові C++: якщо звільнений об'єкт містить vtable-вказівник (таблицю віртуальних методів), зловмисник може підмінити адресу методу у vtable і отримати виконання довільного коду при наступному виклику віртуальної функції. Наступний фрагмент ілюструє принцип UAF.

Приклад коду C: вразливість use-after-free

```
#include <stdlib.h>
#include <string.h>

typedef struct { char name[32]; void (*action)(); } Object;

void safe_action()   { /* ... */ }
void exploit_code()  { /* шкідливий код зловмисника */ }

void uaf_example() {
    Object *obj = malloc(sizeof(Object));
    obj->action = safe_action;
    free(obj);          /* obj звільнено               */
                        /* але вказівник obj ще живе   */

    /* Зловмисник перевиділяє той самий блок:          */
    char *fake = malloc(sizeof(Object));
    memset(fake, 0, sizeof(Object));
    /* Записує адресу exploit_code у поле action:      */
    ((Object*)fake)->action = exploit_code;

    obj->action();  /* USE-AFTER-FREE: виклик exploit! */
}
```

Витікання пам'яті (memory leak) є проблемою стабільності, а не прямою вразливістю безпеки, однак може мати серйозні наслідки для доступності сервісу. Витікання виникає, коли програма виділяє пам'ять, але не звільняє її після того, як вона перестала бути потрібною. Накопичення невивільнених блоків поступово вичерпує оперативну пам'ять, активує swap і може призвести

до завершення процесу ядром через OOM Killer. У сучасній практиці для виявлення витікань застосовуються інструменти динамічного аналізу: Valgrind із модулем memcheck та AddressSanitizer (ASan), що вбудовується у програму компілятором GCC або Clang. Обидва інструменти дозволяють точно локалізувати місце виділення «витікаючої» пам'яті.

Практичний приклад

Перевірка програми на витікання пам'яті за допомогою Valgrind:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./my_program
```

Типовий вивід при виявленні витікання:

```
==12345== LEAK SUMMARY:
==12345== definitely lost: 64 bytes in 1 blocks
==12345== at 0x4C2FB0F: malloc (vg_replace_malloc.c:299)
==12345== by 0x10916B: uaf_example (example.c:5)
```

Компіляція з AddressSanitizer для виявлення UAF та overflow:

```
$ gcc -fsanitize=address -fno-omit-frame-pointer -g prog.c -o prog
$ ./prog # ASan виводить повідомлення при порушенні
```

Стан захисних механізмів конкретного виконуваного файлу:

```
$ checksec --file=./prog
```

Для нейтралізації розглянутих класів вразливостей сучасні операційні системи реалізують комплекс апаратних і програмних засобів захисту. DEP (Data Execution Prevention), відомий також як NX (No-Execute) або XD (eXecute Disable), запобігає виконанню коду зі сторінок, позначених лише для даних. Механізм реалізується через NX-біт у записах PTE: при спробі виконання інструкції за адресою з встановленим NX-бітом процесор генерує апаратний виняток. DEP безпосередньо нейтралізує атаки з ін'єкцією shellcode: навіть якщо зловмисник зміг записати свій код на стек або у купу, виконати його неможливо, оскільки відповідні сторінки позначено як невиконувані. Перевірити наявність підтримки NX на рівні процесора у середовищі GNU/Linux можна командою grep nx /proc/cpuinfo.

ASLR (Address Space Layout Randomization) рандомізує базові адреси ключових ділянок адресного простору при кожному запуску програми: стека, купи, розділених бібліотек та, якщо програма скомпільована як Position Independent Executable (PIE), — сегмента коду. Рандомізація суттєво ускладнює атаки, що потребують точного знання адрес: зловмисник не може заздалегідь знати, де у пам'яті розміщена функція system, рядок «/bin/sh» або таблиця GOT. Ступінь ентропії рандомізації залежить від конкретної реалізації ОС і архітектури; у 64-розрядних системах GNU/Linux він достатньо великий, щоб унеможливити перебір. ASLR не захищає від атак, що базуються на витоку адреси (information leak), — якщо зловмисник може прочитати будь-яку адресу у пам'яті жертви, він здатен обчислити базові адреси всіх ділянок.

SMEP (Supervisor Mode Execution Prevention) та SMAP (Supervisor Mode Access Prevention) є апаратними механізмами захисту, що унеможливлюють для ядра виконання коду та доступ до даних, розміщених у просторі користувача, без явного дозволу. SMEP запобігає виконанню ядром коду з сторінок, позначених як користувацькі (U/S=1), що нейтралізує атаки класу ret2usr, де зловмисник розміщував шкідливий код у власному адресному просторі та змушував ядро «стрибнути» на нього. SMAP розширює цю гарантію: будь-яке читання або запис ядром у простір користувача (крім явно дозволених операцій між позначками STAC/CLAC) спричиняє апаратний виняток. Обидва механізми активуються автоматично ядром Linux за наявності відповідної апаратної підтримки, що відображається у прапорцях CR4.

Stack Canaries (стекові «канарки») є захисним механізмом, що реалізується на рівні компілятора. GCC і Clang з прапорцем -fstack-protector (або -fstack-protector-strong) додають до функцій, що містять стекові буфери, код ініціалізації та верифікації: перед збереженою адресою повернення на стек записується «канарка» — псевдовипадкове значення, яке генерується при запуску програми. Перед виходом з функції перевіряється незмінність канарки; якщо переповнення буфера перезаписало її — програма аварійно завершується викликом __stack_chk_fail. Канарки суттєво ускладнюють використання стекових переповнень, але не захищають від атак з довільним записом (arbitrary write) або від витоків значення канарки. Рисунок 3.6 узагальнює взаємодію всіх механізмів захисту.

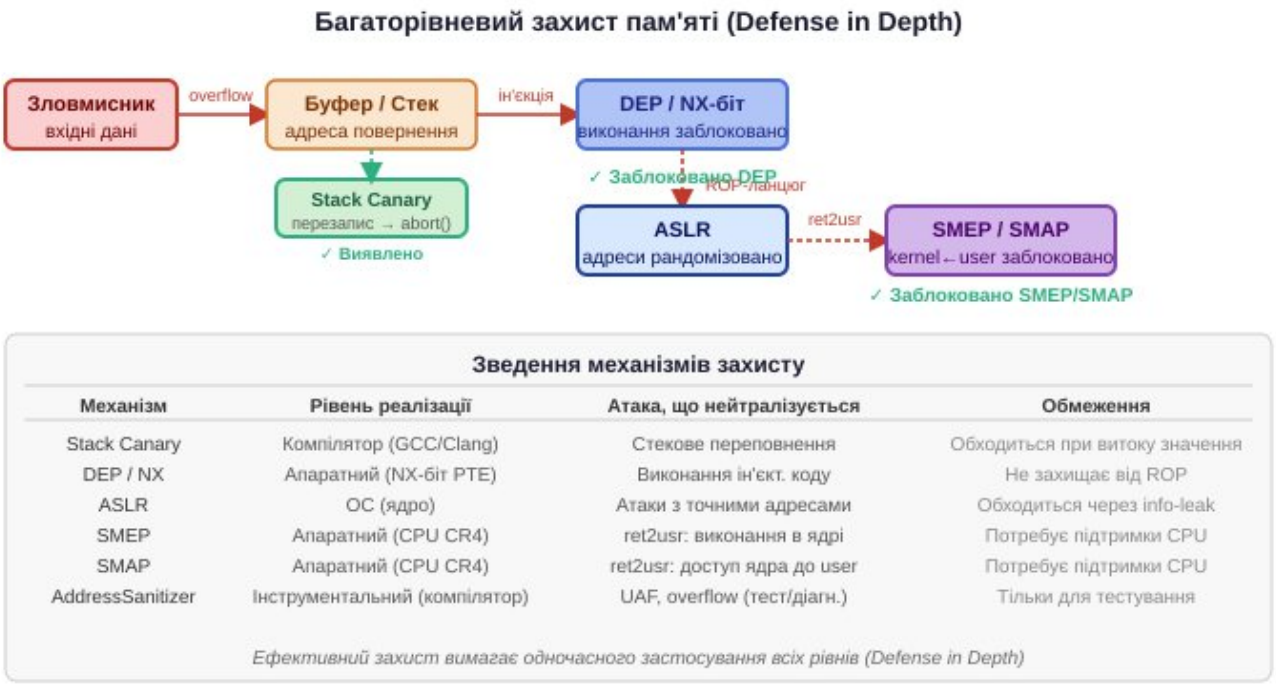

Рисунок 3.6 — Багаторівневий захист пам'яті (Defense in Depth)

Рисунок 3.6 відображає ланцюжок подолання механізмів захисту з точки зору зловмисника та показує, на якому кроці кожен механізм спрацьовує. Ліворуч — зловмисник із вхідними даними. Стрілка «overflow» веде до блоку

«Буфер / Стек»; під ним Stack Canary фіксує перезапис і переривае виконання. Якщо канарка обійдена, стрілка «ін'єкція» веде до блоку DEP/NX, що блокує виконання shellcode. Якщо зловмисник переходить до ROP-атаки (використання наявного коду), ASLR унеможливлює точне знання адрес гаджетів. Нарешті, спроба ret2usr зупиняється блоком SMEP/SMAP. У нижній частині рисунка розміщено зведену таблицю з рівнем реалізації, класом нейтральованих атак та обмеженнями кожного механізму.

Наведений аналіз підтверджує: жоден із механізмів захисту не є самодостатнім. DEP не захищає від ROP; ASLR обходиться через витік адреси; Stack Canary не перешкоджає атакам з довільним записом. Ефективний захист вимагає одночасного застосування всіх доступних рівнів — стратегії «захист у глибину» (Defense in Depth), де кожен механізм компенсує обмеження інших. Для практичної перевірки стану захисних механізмів у середовищі GNU/Linux рекомендується використання наступних команд.

Практичний приклад

Перевірка підтримки NX, SMEP, SMAP процесором:

```
$ grep -E ' nx | smep | smap ' /proc/cpuinfo
```

Поточний стан ASLR (0=вимк., 1=частковий, 2=повний):

```
$ cat /proc/sys/kernel/randomize_va_space
```

Перевірка захисних механізмів виконуваного файлу:

```
$ checksec --file=/bin/bash
Очікуваний вивід: RELRO: Full, Stack: Canary found,
NX: NX enabled, PIE: PIE enabled, ASLR: enabled
```

Демонстрація ASLR: базова адреса libc різна при кожному запуску:

```
$ ldd /bin/bash | grep libc       # адреса змінюється
$ ldd /bin/bash | grep libc       # повторити і порівняти
```

Таблиця 3.3 — Механізми захисту пам'яті: рівень реалізації, покриття та обмеження

Механізм	Рівень реалізації	Клас нейтральованих атак	Обмеження
DEP / NX	Апаратний (NX-біт PTE)	Виконання ін'єктованого коду	Не захищає від ROP
ASLR	ОС (ядро)	Атаки з точними адресами	Обходиться через info-leak
Stack Canary	Компілятор (GCC/Clang)	Стекове переповнення буфера	Обходиться при витоку значення
SMEP	Апаратний (CR4 CPU)	ret2usr (виконання у Ring 0)	Потребує підтримки CPU
SMAP	Апаратний (CR4 CPU)	ret2usr (доступ до даних user)	Потребує підтримки CPU
AddressSanitizer	Компілятор	UAF, overflow (тест. та	Лише для

Механізм	Рівень реалізації	Клас нейтральованих атак	Обмеження
	(інструментація)	діагн.)	тестування

Наведені в таблиці 3.3 відомості дають цілісне уявлення про архітектуру захисту сучасних операційних систем від вразливостей пам'яті. Поєднання апаратних механізмів (NX, SMEP, SMAP), засобів ядра (ASLR) та компіляторних інструментів (Stack Canary, PIE, AddressSanitizer) утворює той рівень захисту, який забезпечує стійкість сучасних систем до переважної більшості відомих атак. Разом з тим безперервний розвиток методів обходу цих механізмів підкреслює важливість комплексного підходу до безпеки та регулярного оновлення програмного забезпечення.

Тема 4. Файлові системи та організація зберігання даних

4.1 Базові концепції та архітектура файлових систем

Файлова система є одним із фундаментальних компонентів будь-якої операційної системи, оскільки саме вона визначає спосіб організації, зберігання та доступу до даних на носіях інформації. З точки зору користувача, файлова система надає абстракцію ієрархічного простору імен, де дані організовані у вигляді файлів та каталогів. Проте з точки зору системного рівня, за цією простою абстракцією приховується складна структура метаданих, алгоритмів розміщення блоків та механізмів синхронізації. Розуміння внутрішньої будови файлових систем є необхідною умовою для грамотного адміністрування систем, забезпечення їх надійності та відновлення після збоїв.

Будь-який носій інформації — жорсткий диск, твердотільний накопичувач (SSD), USB-накопичувач або оптичний диск — є, по суті, масивом блоків фіксованого розміру. Завдання файлової системи полягає в тому, щоб перетворити цей лінійний простір блоків на зручну для людини ієрархічну структуру. Для цього файлова система зберігає на носії дві принципово різні категорії даних: власне вміст файлів (корисні дані) та метадані — інформацію про самі файли. Метадані включають такі атрибути, як ім’я файлу, розмір, час створення та модифікації, права доступу, власник, а також фізичне розташування блоків даних файлу на носії. Баланс між обсягом метаданих та ефективністю їх зберігання є одним із ключових параметрів, за якими порівнюють різні файлові системи.

Щоб звернутися до конкретного файлу, операційна система повинна мати змогу відшукати його метадані, а за ними — і самі дані. Ця задача вирішується по-різному залежно від архітектури файлової системи. Існують три принципові підходи до організації метаданих: метадані, що зберігаються разом із даними у тій самій ділянці диска (характерно для дуже простих файлових систем); метадані, що централізовано зберігаються в одному місці (як MFT у NTFS); та розподілені метадані, де inode-таблиці рівномірно розподілені по групах блоків (як у ext4). Централізований підхід зручний для пошуку та дефрагментації, але робить зону метаданих потенційною точкою відмови. Розподілений підхід є кращим для великих томів із великою кількістю файлів, оскільки знижує ймовірність одночасного пошкодження всіх метаданих і зменшує час пошуку inode завдяки просторовій близькості inode до відповідних даних.

Центральне місце у більшості Unix-подібних файлових систем займає структура, відома як inode (index node). Кожен файл або каталог у системі описується одним inode, який містить усі метадані файлу, крім його імені. Ім’я файлу зберігається окремо — у записі каталогу, який пов’язує текстове ім’я із числовим ідентифікатором inode. Така архітектура дозволяє одному файлу мати декілька імен (так звані “жорсткі посилання”), оскільки декілька записів каталогу можуть вказувати на один і той самий inode. Поле лічильника посилань у inode відстежує, скільки імен посилаються на даний файл: коли лічильник досягає нуля, простір файлу може бути повторно використаний.

Кожен inode також містить покажчики на блоки даних файлу: для малих файлів це прямі покажчики (зазвичай перші 12), а для великих файлів використовується каскадна структура з одинарними, подвійними та потрійними непрямими покажчиками, що дозволяє адресувати файли розміром до кількох терабайт.

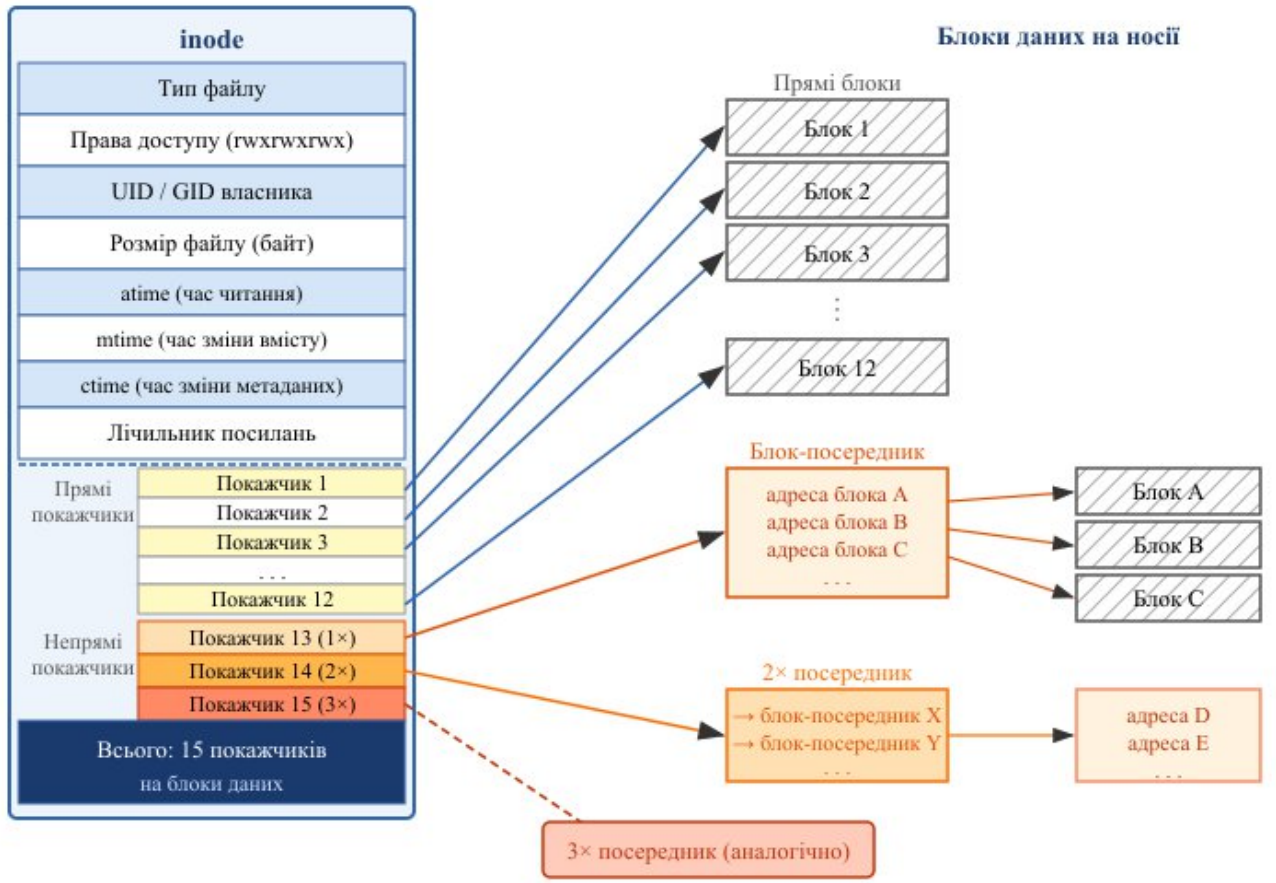

Рисунок 4.1 — Структура inode у Unix-подібній файловій системі

У системах Windows аналогічну роль виконує MFT (Master File Table) файлової системи NTFS. MFT є таблицею записів фіксованого розміру (як правило, 1 КБ кожен), де кожен запис описує один файл або каталог. Для малих файлів (до кількох сотень байт) самі дані можуть зберігатися безпосередньо у записі MFT — такий режим називається резидентними атрибутами. Цей підхід суттєво пришвидшує доступ до малих файлів, оскільки виключає необхідність зчитування додаткових блоків. Для більших файлів у запис MFT вносяться так звані потоки даних із зазначенням місця розташування екстентів — неперервних ділянок носія, що містять дані файлу. NTFS резервує перші 16 записів MFT для власних системних файлів: $MFT (сама таблиця), $MFTMirr (дзеркальна копія перших 4 записів), $LogFile (журнал транзакцій), $Volume (інформація про том), $AttrDef (таблиця атрибутів) та інші. Пошкодження цих системних записів фактично рівнозначно пошкодженню файлової системи, тому NTFS зберігає часткові резервні копії найкритичніших структур.

Важливим поняттям при роботі з файловими системами є розмір кластера (або блоку). Носій розбивається на кластери — мінімальні одиниці виділення

простору. Якщо розмір файлу не кратний розміру кластера, останній кластер виділяється цілком, але не заповнюється повністю — невикористаний простір у кінці кластера називається внутрішньою фрагментацією. Чим більший розмір кластера, тим ефективнішою є адресація великих файлів (менше покажчиків, менше фрагментів), але тим більше місця витрачається при зберіганні безлічі дрібних файлів. Стандартний розмір кластера у 4 КБ є розумним компромісом для більшості сценаріїв. У таблиці 4.1 наведено типові розміри кластерів NTFS для різних обсягів томів.

Таблиця 4.1 — Розміри кластерів NTFS залежно від обсягу тому

Обсяг тому	Розмір кластера за замовчуванням
До 512 МБ	512 байт
512 МБ — 1 ГБ	1 КБ
1 ГБ — 2 ГБ	2 КБ
2 ГБ — 2 ТБ	4 КБ
2 ТБ — 16 ТБ	8 КБ
16 ТБ — 32 ТБ	16 КБ
32 ТБ — 64 ТБ	32 КБ
64 ТБ — 128 ТБ	64 КБ
128 ТБ — 256 ТБ	128 КБ

Важливою характеристикою сучасних файлових систем є підтримка **журналювання** (journaling). Уявіть ситуацію, коли система раптово втрачає живлення в момент складної операції — наприклад, під час перейменування файлу, яке включає запис нового запису до каталогу-призначення, видалення старого запису та оновлення метаданих inode. Якщо буде виконана лише частина цих кроків, файлова система опиниться в неузгодженому стані: файл може виявитися “загубленим” — виключеним із каталогу, але таким, що займає місце на диску. Журналювання вирішує цю проблему шляхом попереднього запису планованих змін до спеціальної ділянки на носії, яку називають журналом транзакцій. Якщо живлення втрачається під час операції, після перезавантаження система перегляне журнал і або завершить незакінчену операцію (redo), або відкотить її до початкового стану (undo), гарантуючи узгодженість файлової системи за лічені секунди — тоді як без журналювання аналогічна перевірка fsck могла тривати годинами для великих томів.

Залежно від того, що саме записується до журналу, розрізняють три режими журналювання. У режимі writeback до журналу записуються лише зміни метаданих, а дані записуються безпосередньо на носій без проходження через журнал — цей режим є найшвидшим, але не гарантує цілісності вмісту файлів після збою (метадані можуть вказувати на нові блоки, що ще не були повністю записані). У режимі ordered (упорядкованому) метадані журналюються, але запис даних відбувається гарантовано до того, як відповідні зміни метаданих вносяться до журналу — це забезпечує, що метадані завжди вказуватимуть на коректно записані дані; ext4 використовує цей режим за

замовчуванням як компроміс між надійністю та продуктивністю. У режимі data до журналу записуються і метадані, і самі дані, що забезпечує максимальну цілісність ціною двократного запису кожного блоку та суттєво нижчої продуктивності.

4.2 Порівняльний аналіз файлових систем

Протягом десятиліть розвитку обчислювальних систем було розроблено велику кількість файлових систем, кожна з яких відображає технологічні можливості та вимоги свого часу. Вибір файлової системи — це не просто технічна деталь: він визначає продуктивність системи, максимально підтримуваний обсяг даних, наявність механізмів захисту та відновлення, а також сумісність із різними операційними системами та інструментами. Сучасний системний адміністратор або інженер повинен розуміти особливості найпоширеніших файлових систем, щоб зробити обґрунтований вибір для конкретного завдання.

FAT32 (File Allocation Table, 32-бітна версія) є однією з найстаріших файлових систем, що продовжує широко використовуватися завдяки максимальній сумісності між різними операційними системами. Архітектура FAT32 проста: після завантажувального сектора розташовані дві копії таблиці FAT (друга є резервною), кореневий каталог і область даних. Таблиця FAT описує стан кожного кластера: чи зайнятий він (із номером наступного кластера ланцюжка), чи вільний, чи є останнім у ланцюжку. Головне обмеження FAT32 — максимальний розмір одного файлу в 4 ГБ — є прямим наслідком 32-бітного поля розміру у записах каталогу. Відсутність журналювання, прав доступу та шифрування робить FAT32 непридатною для системних розділів, але вона залишається стандартом для знімних носіїв. Нащадком FAT32 є **exFAT** (Extended FAT), що знімає обмеження на розмір файлу і тому, зберігаючи просту архітектуру — саме exFAT є рекомендованим форматом для флеш-накопичувачів та SD-карт великого обсягу стандартом SD Association.

NTFS (New Technology File System) є основною файловою системою Windows, починаючи з NT 3.1, і поєднує у собі розвинену систему атрибутів, журналювання та повноцінні механізми безпеки. Вона підтримує ACL на рівні файлів і каталогів, шифрування через EFS, прозоре стиснення, тверді та символічні посилання, а також розширені атрибути. Журналювання у NTFS реалізоване через системний файл $LogFile і забезпечує відновлення метаданих після збоїв за секунди. Структура на основі MFT є надзвичайно ефективною для пошуку за ідентифікатором файлу — операція, що потребує O(1) замість O(log N) при пошуку по дереву каталогів. Разом із тим NTFS є власницькою розробкою Microsoft, і хоча у Linux є зрілий драйвер ntfs3 (з ядра 5.15), для критичних сценаріїв повної взаємодії NTFS-томи найкраще обслуговувати на рідній платформі Windows.

ext4 є де-факто стандартом для більшості дистрибутивів Linux, включаючи Ubuntu, Debian та Red Hat. Ключовою особливістю ext4 порівняно з ext3 є підтримка **extents** — замість зберігання списку окремих блоків файл

описується набором неперервних діапазонів (extent задається початковим блоком і довжиною), що суттєво зменшує кількість метаданих для великих файлів. Відкладене виділення блоків (delayed allocation) дозволяє ядру накопичувати запити і виділяти блоки більш оптимально — у неперервних ділянках, зменшуючи фрагментацію. ext4 є зворотно сумісною з ext2/ext3 і підтримує томи до 1 ЕБ та файли до 16 ТБ. Зрілість і надійність ext4 після багатьох років виробничого використання у найвибагливіших серверних середовищах роблять її одним із найнадійніших варіантів для загального призначення.

APFS (Apple File System) з'явилася у 2017 році як заміна HFS+ і будувалася з нуля з урахуванням особливостей SSD-накопичувачів. Архітектура APFS базується на copy-on-write B+-дереві: будь-яка зміна завжди записується у новий блок, що унеможливлює "перезапис у те саме місце" і гарантує атомарність будь-якої операції. Контейнерна модель APFS дозволяє розміщувати кілька томів у межах одного "контейнера", що динамічно ділить простір між ними без заздалегідь заданих фіксованих розмірів — це значне покращення порівняно з традиційною жорсткою схемою розділів. Атомарне клонування файлів і каталогів (без фактичного копіювання даних до першої зміни) є особливо зручним для інструментів розробки, що роблять копії великих образів.

ZFS (Zettabyte File System) від OpenZFS є найбільш функціонально насиченою файловою системою серед розглянутих і поєднує у собі файлову систему, менеджер логічних томів та підсистему перевірки цілісності. Архітектура ZFS базується на пулах зберігання (zpool) — об'єднаннях фізичних пристроїв, поверх яких створюються набори даних (dataset) з індивідуальними налаштуваннями. Кожен блок даних захищений 256-бітною контрольною сумою, що дозволяє виявляти та автоматично виправляти пошкодження — включно з "тихим" пошкодженням, яке не проявляється у вигляді апаратної помилки. Підтримка RAID-Z, дедуплікації, прозорого стиснення (lz4, zstd, gzip), реплікації між вузлами та гнучкого керування квотами і резервуваціями робить ZFS незамінним вибором для корпоративного зберігання даних.

Окремо варто згадати **XFS** — файлову систему, розроблену Silicon Graphics у 1994 році і нині стандартну для RHEL/CentOS. XFS відрізняється чудовою масштабованістю для паралельних операцій вводу-виводу та ефективністю при роботі з великими файлами, що робить її популярним вибором для медіасерверів, систем відеозапису та кластерів зберігання. XFS підтримує відкладене виділення, extents і журналювання метаданих, а також онлайн-дефрагментацію та розширення тому без відмонтування. Важливе обмеження: скорочення тому XFS не підтримується — при плануванні розмірів розділів це необхідно враховувати.

У таблиці 4.2 наведено порівняння ключових характеристик розглянутих файлових систем.

Таблиця 4.2 — Порівняльні характеристики файлових систем

Характеристика	FAT32	exFAT	NTFS	ext4	XFS	APFS	ZFS
Журналювання / COW	—	—	Журн.	Журн.	Журн.	COW	COW
Макс. розмір файлу	4 ГБ	16 ЕБ	16 ЕБ	16 ТБ	8 ЕБ	8 ЕБ	16 ЕБ
Макс. розмір тому	~2 ТБ	128 ПБ	256 ТБ	1 ЕБ	8 ЕБ	Необм.	256 ЗБ
Права доступу	—	—	ACL	POSIX+ACL	POSIX+ACL	POSIX+ACL	POSIX+ACL
Вбудоване шифрування	—	—	EFS	fscrypt	—	Так	Так
Моментальні знімки	—	—	VSS*	—	—	Так	Так
Перевірка цілісності	—	—	Частк.	fsck	fsck	Частк.	SHA-256
Оптимізація для SSD	—	Частк.	Частк.	Частк.	Частк.	Так	Частк.
Типова платформа	Крос	Крос	Windows	Linux	Linux/RHEL	Apple	FreeBSD/Linux

*VSS (Volume Shadow Copy Service) реалізується на рівні ОС, а не безпосередньо в NTFS.

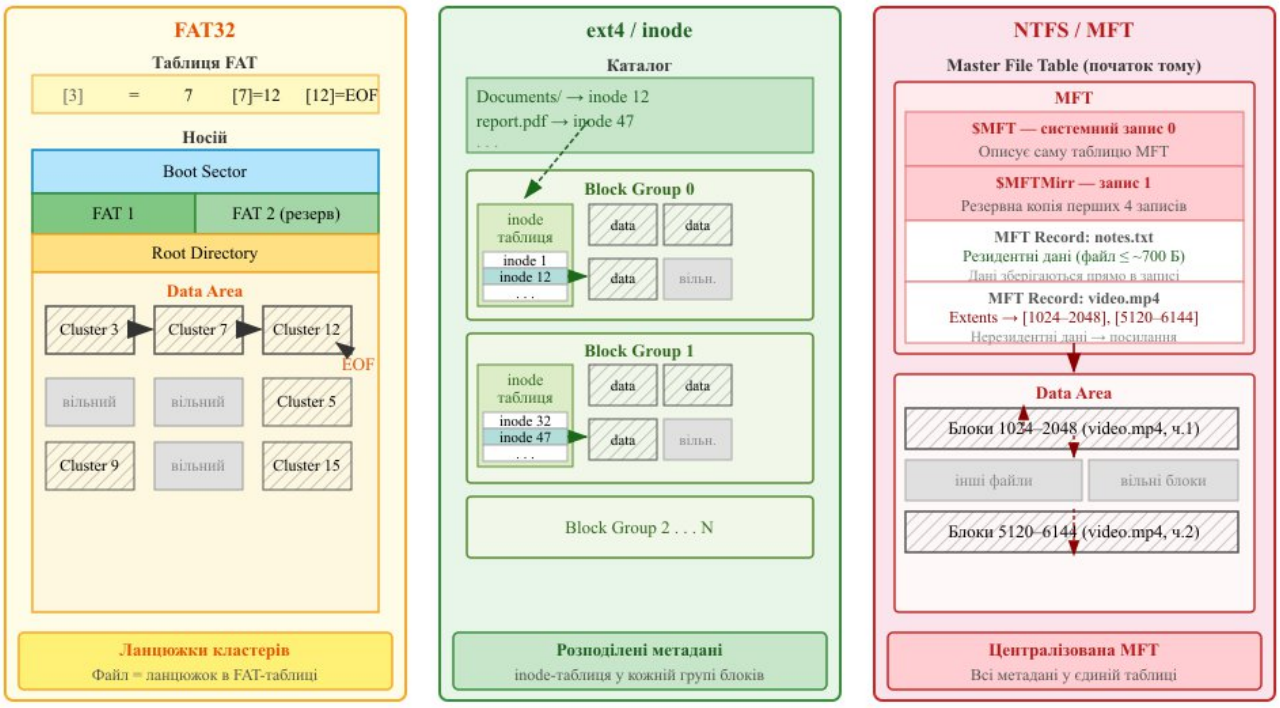

Рисунок 4.2 — Архітектурне порівняння підходів до організації метаданих

Вибір файлової системи суттєво залежить від конкретного сценарію. Для домашніх Linux-серверів загального призначення ext4 залишається надійним та зрілим вибором. Для сховищ із критично важливими даними та вимогами до цілісності і реплікації ZFS є практично безальтернативним рішенням, незважаючи на підвищені вимоги до пам’яті (рекомендовано від 8 ГБ ОЗП для ARC-кешу). Для систем Apple APFS є єдиним підтримуваним варіантом. Для баз даних з інтенсивним паралельним вводом-виводом XFS часто перевершує ext4 завдяки кращій масштабованості при паралельній роботі.

4.3 Механізми захисту цілісності даних та відновлення після збоїв

Забезпечення цілісності даних є однією з найважливіших функцій сучасних файлових систем. Втрата або пошкодження даних може відбуватися з численних причин: апаратні збої накопичувачів, несподіване вимкнення живлення, програмні помилки, електромагнітні завади або поступова деградація носія — так зване "гниття бітів" (bit rot), коли намагнічування секторів HDD або заряд комірок флеш-пам'яті деградують настільки, що дані стають нечитабельними. За оцінками дослідників, жорсткі диски у виробничих умовах мають частоту невиявлених помилок читання (URE) приблизно одну на кожен петабайт прочитаних даних — для масштабних сховищ ця цифра є вельми реальною загрозою, особливо під час відновлення після збою RAID-масиву, де читається вся поверхня усіх дисків. Сучасні файлові системи використовують цілий арсенал технічних рішень для протидії цим загрозам.

Технологія **copy-on-write** (COW, копіювання при записі) є одним із найпотужніших механізмів захисту цілісності. Замість того щоб перезаписувати дані безпосередньо на місці (in-place update), COW-файлові системи записують нові версії даних у нові вільні блоки, а потім атомарно оновлюють покажчики у дереві метаданих. Стара версія залишається незайманою до завершення операції: якщо система виходить з ладу в середині запису, після відновлення покажчики у дереві все ще вказують на стару, цілісну версію — файлова система залишається у повністю узгодженому стані без жодних журнальних записів. Ця властивість є принципово іншим підходом до гарантування атомарності операцій: журналювання записує "наміри" і може їх "відмотати", тоді як COW ніколи не руйнує старий стан під час створення нового. Для файлових систем, що підтримують знімки, COW є концептуально чистішим і ефективнішим рішенням.

Механізм **моментальних знімків** (snapshots) є природним наслідком COW-архітектури і одним із найцінніших інструментів операційного управління даними. Коли створюється знімок, файлова система просто зберігає посилання на поточний корінь дерева метаданих і позначає пов'язані блоки як "захищені знімком". Наступні операції запису відповідно до принципу COW йдуть у нові блоки, не торкаючись блоків, на які посилається знімок. Таким чином, стара версія дерева метаданих і відповідні блоки даних зберігаються незмінними, формуючи "заморожений" образ файлової системи. Видалення знімку є зворотною операцією: блоки, що більше не посилаються ані активним деревом, ані жодним зі знімків, позначаються як вільні. Знімки широко використовуються для швидкого відновлення: якщо оновлення системи призвело до нестабільності, за лічені секунди можна відновити попередній стан через `zfs rollback` або аналогічну операцію APFS.

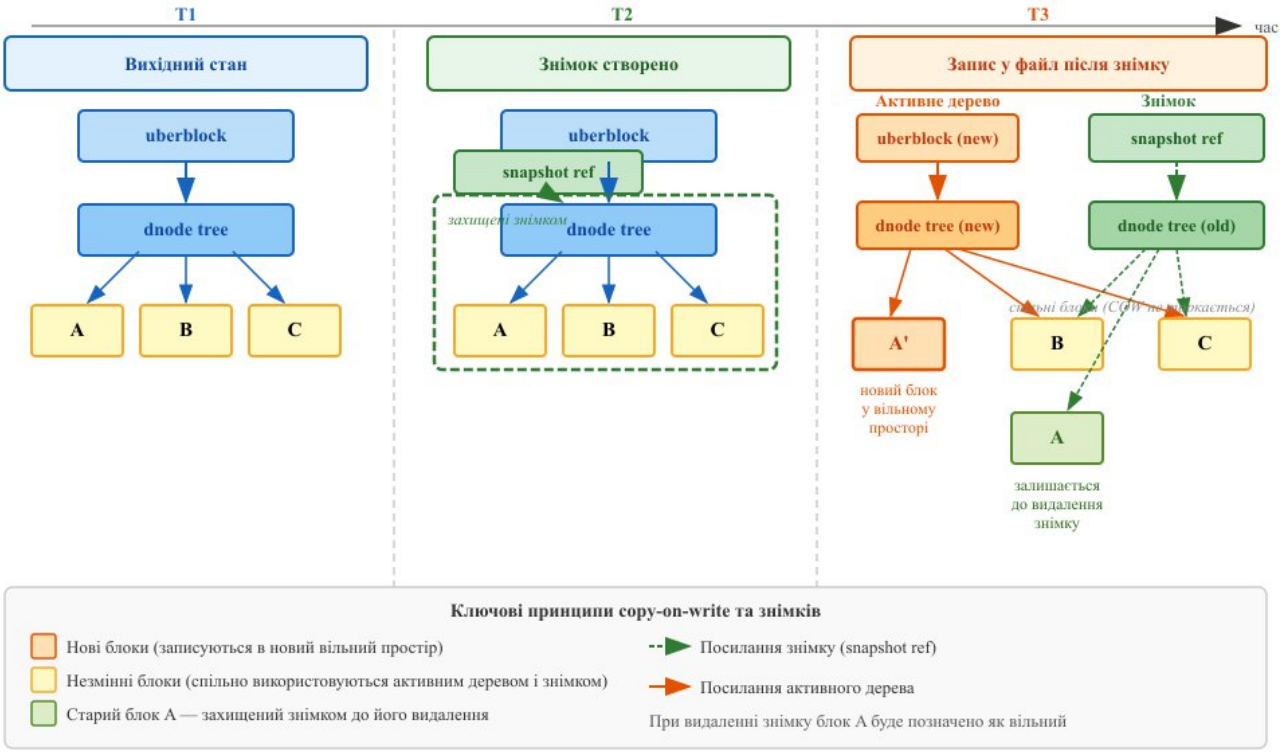

Рисунок 4.3 — Механізм copy-on-write та моментальних знімків у ZFS

Перевірка та відновлення цілісності реалізується по-різному в різних файлових системах. У ZFS кожен блок захищений контрольною сумою, що зберігається не у самому блоці, а у батьківському вузлі дерева метаданих — це принципово важливо, адже якщо б сума зберігалася поруч із даними, пошкодження могло б зачепити обидва. Команда `zpool scrub` запускає фоновий прохід по всьому пулу, читаючи кожен блок і порівнюючи його контрольну суму: при наявності надлишкових дисків пошкоджені блоки автоматично відновлюються з резервних копій, а статистика помилок відображається у `zpool status`. Рекомендується виконувати scrub не рідше разу на місяць, щоб виявляти бітове гниття до того, як воно охопить надлишкові копії. Утиліта `fsck.ext4` для ext4 перевіряє суперблок, таблиці inode та бітові карти блоків на внутрішні суперечності — завдяки журналюванню вона запускається вкрай рідко, переважно у профілактичних цілях або після апаратних збоїв. Для NTFS аналогічну функцію виконує `chkdsk`, який також реконструює пошкоджені записи MFT із їх дзеркальних копій у $MFTMirr.

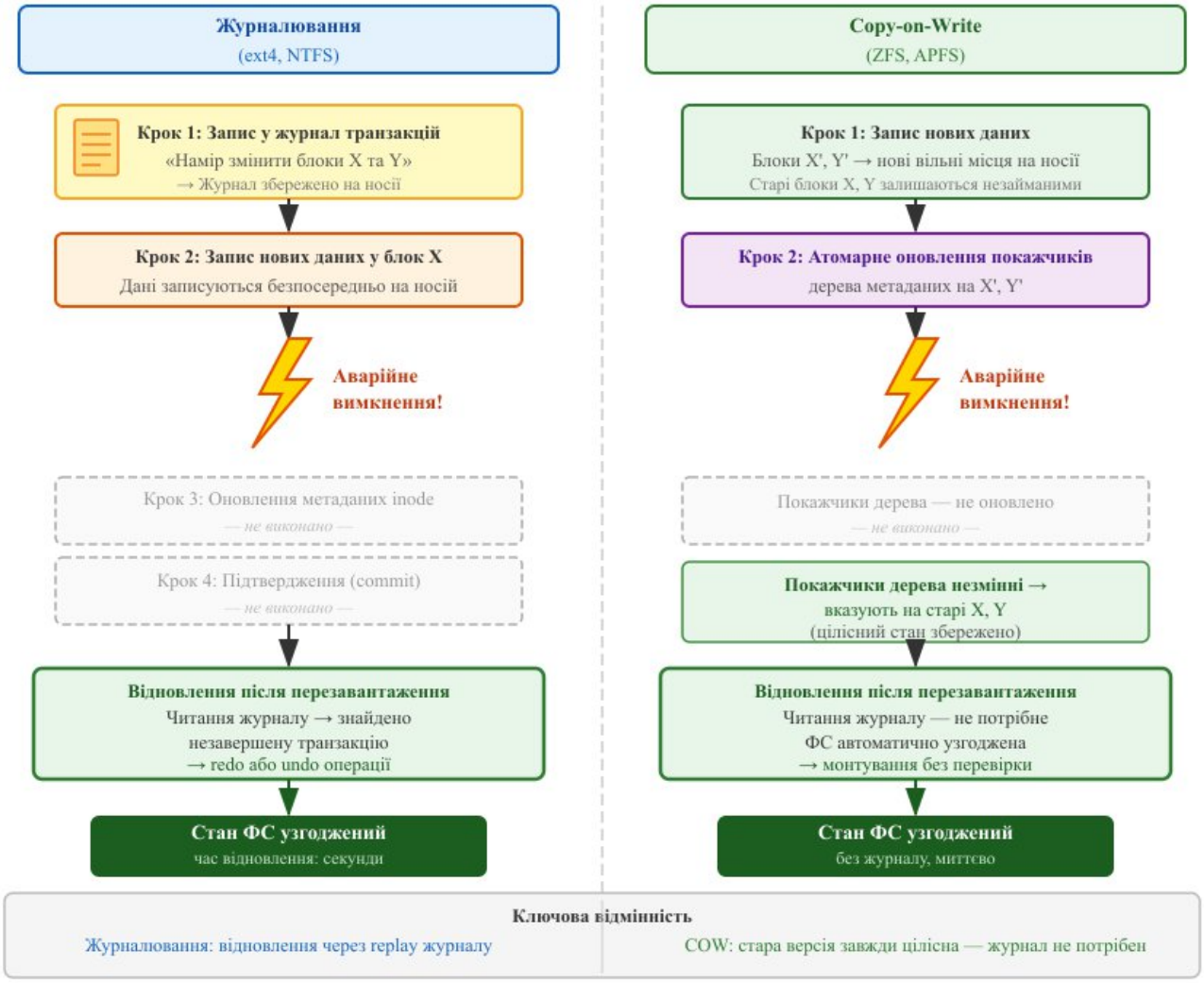

Рисунок 4.4 — Порівняння механізмів журналювання та copy-on-write при аварійному вимкненні

TRIM/DISCARD — механізм взаємодії між файловою системою та SSD-накопичувачем, що прямо впливає як на продуктивність, так і на ресурс носія. Контролер SSD не може перезаписати дані безпосередньо у зайняту комірку флеш-пам’яті — перед записом необхідно виконати стирання цілого блоку флеш (як правило, 256 КБ або більше). Якщо файлова система видаляє файл, але не повідомляє про це контролер, він при наступному записі у цей сектор буде змушений читати весь блок, стирати його та записувати нові дані — це цикл Read-Modify-Write, значно повільніший за простий запис у порожній блок. Команда TRIM (для SATA) або Unmap (для NVMe/SCSI) вирішує проблему: при видаленні файлу ОС повідомляє накопичувачу, які сектори більше не містять актуальних даних, і контролер SSD може завчасно очистити ці блоки у фоновому режимі, підтримуючи постійний пул порожніх блоків. Файлові системи підтримують TRIM у двох режимах: synchronous discard (кожне видалення негайно надсилає TRIM — простіше, але може сповільнити видалення) та periodic fstrim (система раз на тиждень або щодня запускає `fstrim` через systemd-timer — рекомендований підхід для виробничих систем)..

4.4 Організація зберігання: розділи, томи, RAID та VFS

Файлова система функціонує поверх певної абстракції зберігання, яка може бути як простим розділом диска, так і складним програмним RAID-масивом або логічним томом. Розуміння цих рівнів абстракції є необхідним для побудови надійних та масштабованих систем зберігання, оскільки правильна архітектура стеку зберігання не менш важлива, ніж вибір самої файлової системи.

Традиційно фізичні носії розбиваються на **розділи** (partitions). Існують дві основні схеми розбивки: стара MBR (Master Boot Record) та сучасна GPT (GUID Partition Table). MBR обмежена томами до 2 ТБ і дозволяє максимум 4 первинних розділи (або 3 первинних + 1 розширений з необмеженою кількістю логічних). GPT підтримує томи до 9,4 ЗБ і практично необмежену кількість розділів (128 у більшості реалізацій), зберігає резервну копію таблиці у кінці диска, включає захисний CRC32 для виявлення пошкоджень таблиці і є обов'язковою для завантаження у режимі UEFI. Для всіх нових систем GPT є рекомендованою схемою. Типове серверне розбиття диска Linux може включати окремі розділи для `/` (кореневий), `/boot` (завантажувальний), `/home` (домашні каталоги), `/var` (змінні дані, логи, бази даних) та swap — така ізоляція дозволяє запобігти ситуації, коли переповнення одного розділу (наприклад, `/var` логами) не впливає на роботу решти системи.

Логічні менеджери томів (LVM у Linux, Storage Spaces у Windows, diskgroup у Solaris/ZFS) додають гнучкий шар між фізичними носіями та файловими системами. У Linux LVM організований у три рівні: PV (Physical Volume) — окремий носій або розділ, що увійшов до LVM; VG (Volume Group) — об'єднання PV у єдиний пул простору; LV (Logical Volume) — логічний розділ довільного розміру, нарізаний із VG. Перевага LVM — можливість онлайн-розширення LV та VG (додавши новий PV), а також можливість міграції даних між носіями без зупинки системи командою `pvmove`. LV слугують звичайними блоковими пристроями для файлових систем — їх можна форматувати у будь-яку файлову систему, використовувати для swap або як сире блокове сховище для баз даних.

RAID (Redundant Array of Independent Disks) — технологія об'єднання фізичних дисків у масив для підвищення продуктивності та/або надійності. Характеристики основних рівнів RAID наведено у таблиці 4.3.

Таблиця 4.3 — Характеристики основних рівнів RAID

Рівень RAID	Мін. дисків	Надлишковість	Продукт. читання	Продукт. запису	Корисна ємність
RAID 0 (Striping)	2	Відсутня	Висока	Висока	100%
RAID 1 (Mirroring)	2	1 диск	Висока	Нижча	50%
RAID 5	3	1 диск	Висока	Середня	(N-1)/N
RAID 6	4	2 диски	Висока	Нижча за	(N-2)/N

Рівень RAID	Мін. дисків	Надлишковість	Продукт. читання	Продукт. запису	Корисна ємність
				RAID 5	
RAID 10	4	До N/2 дисків	Висока	Середня	50%
RAID-Z1 (ZFS)	3	1 диск	Висока	Середня	(N-1)/N
RAID-Z2 (ZFS)	4	2 диски	Висока	Середня	(N-2)/N
RAID-Z3 (ZFS)	5	3 диски	Висока	Нижча	(N-3)/N

RAID-Z у ZFS усуває відому проблему “write hole” класичного RAID-5/6 — ситуацію, коли після аварійного вимкнення під час запису блок даних і відповідний блок парності опиняються неузгодженими. ZFS завдяки COW-архітектурі записує дані, парність і нові покажчики як єдину атомарну транзакцію, що унеможливлює будь-яку неузгодженість після збою.

VFS (Virtual File System) — абстрактний шар у ядрі, що надає єдиний уніфікований інтерфейс для роботи із різними файловими системами. Завдяки VFS застосунки не мають потреби знати, на якій саме файловій системі зберігаються дані: системні виклики `open()`, `read()`, `write()`, `stat()` завжди звертаються до VFS, а VFS делегує операцію відповідному драйверу. Це дозволяє Linux одночасно монтувати ext4, ZFS, NTFS, NFS, tmpfs (файлова система в ОЗП) та десятки інших, надаючи для всіх єдиний POSIX-інтерфейс. У Windows аналогічну роль виконує I/O Manager ядра NT, хоча Windows традиційно менш відкрита для сторонніх файлових систем.

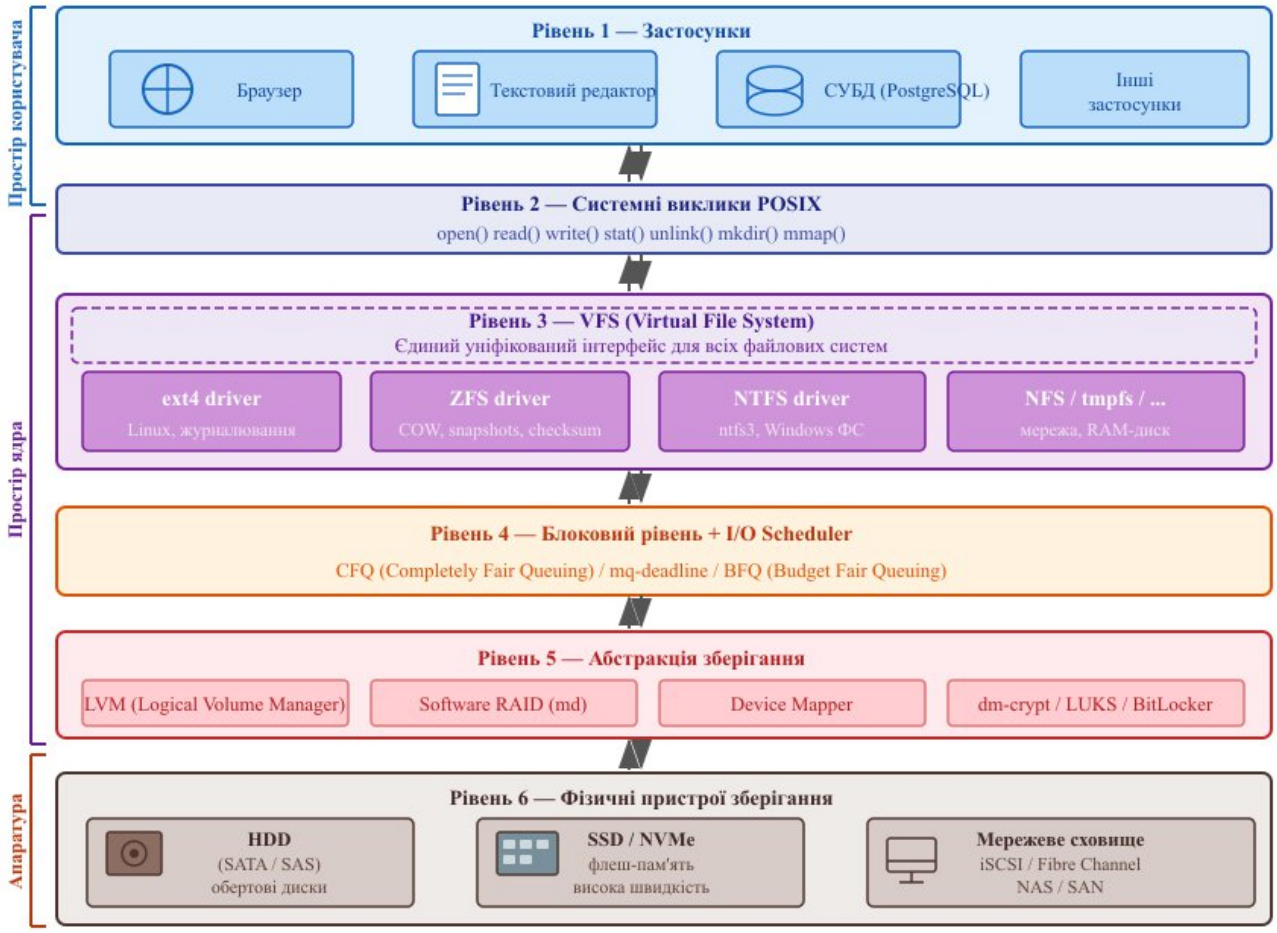

Рисунок 4.5 — Стек зберігання даних від застосунку до фізичного носія

4.5 Файлові системи у контексті резервного копіювання та аудиту

Файлова система не є ізольованим компонентом операційної системи — вона тісно взаємодіє із системою безпеки, механізмами резервного копіювання та підсистемами аудиту. Грамотне розуміння цих взаємозв’язків дозволяє вибудовувати надійні та безпечні інфраструктури, здатні задовольнити як технічні, так і нормативні вимоги (PCI DSS, ISO 27001, HIPAA та інші стандарти інформаційної безпеки).

Резервне копіювання (backup) є невід’ємною частиною будь-якої стратегії захисту даних. Вирізняють кілька типів резервного копіювання, що розрізняються за обсягом копійованих даних і часом відновлення. Повне (full backup) передбачає копіювання всіх файлів без виключень — воно є найпростішим для відновлення (потрібна лише одна резервна копія), але найбільш ресурсомістким. Диференційне (differential) копіює лише файли, змінені після останнього повного копіювання — обсяг зростає з часом, але відновлення вимагає лише двох копій. Інкрементальне (incremental) копіює лише зміни відносно попереднього копіювання будь-якого типу — найефективніше за обсягом і часом резервного копіювання, але відновлення вимагає відтворення всього ланцюжка. Сучасні системи резервного копіювання, такі як Veeam, Bacula або Restic, реалізують дедуплікацію та

стратегії forever-incremental із синтетичними повними копіями, отримуючи переваги інкрементального підходу без ускладненого відновлення.

Файлові системи зі знімками радикально спрощують реалізацію стратегій резервного копіювання. Ключова проблема резервного копіювання живої файлової системи — забезпечення узгодженості (consistency): якщо під час копіювання деякі файли змінюються, резервна копія може містити неузгоджені дані (частину файлів старої версії і частину — нової), що є особливо критичним для баз даних і поштових серверів. Знімок "заморожує" стан файлової системи на певний момент часу, дозволяючи копіювати зі знімку, поки основна система продовжує нормальну роботу — резервна копія отримує узгоджений стан без зупинки системи. У ZFS механізм `zfs send | zfs recv` дозволяє передавати не лише повний знімок, але й інкрементальний потік (`zfs send -i snap1 snap2`) — бінарний потік, що описує лише блоки, які змінилися між двома знімками. Саме так реалізується ефективна асинхронна реплікація між ZFS-вузлами без потреби у зовнішніх інструментах.

Таблиця 4.4 — Порівняння стратегій резервного копіювання

Стратегія	Час створення	Обсяг даних	Час відновлення	Складність відновлення
Повне (Full)	Тривалий	Максимальний	Мінімальний	Низька (1 копія)
Диференційне	Середній	Середній (зростає)	Середній	Низька (2 копії)
Інкрементальне	Мінімальний	Мінімальний	Тривалий	Висока (N копій)
Знімок (Snapshot)	Миттєвий	Мінімальний (COW)	Миттєвий	Низька (вбудована)
ZFS send (інкрем.)	Мінімальний	Лише зміни	Середній	Середня

Аудит файлової системи є критично важливою функцією для забезпечення безпеки та відповідності нормативним вимогам. Аудит дозволяє відслідковувати, хто, коли і які операції виконував із файлами та каталогами — без цього неможливо розслідувати інциденти безпеки або довести відповідність стандартам. У Linux аудит реалізується через підсистему **auditd** та інфраструктуру **inotify/fanotify**. Підсистема auditd функціонує на рівні ядра: правила аудиту задаються через `auditctl` і можуть відстежувати системні виклики на певні шляхи, операції конкретних користувачів, спроби несанкціонованого доступу. Усі відповідні події записуються у структурований журнал `/var/log/audit/audit.log` із полями: time (часова мітка у форматі Unix epoch), pid та ppid (ідентифікатори процесу та батьківського процесу), uid/auid (поточний та реальний UID — auid зберігається навіть при підвищенні привілеїв через sudo), syscall, exe, name (шлях до файлу) та результат операції. Для аналізу журналів використовуються `ausearch` (пошук за критеріями: `-ua 1000 -ts today` знаходить усі дії користувача 1000 сьогодні) та `aureport` (зведені статистичні звіти).

У Windows аудит файлової системи налаштовується через Групові Політики у розділі “Аудит доступу до об’єктів”. Після увімкнення аудиту необхідно для конкретних файлів або каталогів задати SACL (System Access Control List) — список того, чиї і які операції підлягають аудиту. Відповідні події записуються у журнал Windows Security з ідентифікаторами 4663 (звернення до об’єкта), 4656 (запит дескриптора) та 4659 (запит на видалення). Для централізованого збору та кореляції таких подій на підприємствах використовуються SIEM-системи (Splunk, Microsoft Sentinel, ELK Stack), що агрегують журнали з тисяч хостів і дозволяють виявляти підозрілі патерни поведінки в реальному часі.

Зберігання та аналіз розширених атрибутів файлів є ще одним аспектом, що має значення для безпеки. NTFS підтримує **альтернативні потоки даних** (ADS — Alternate Data Streams): до будь-якого файлу можна прикріпити додатковий іменований потік даних, звернувшись до нього як `filename.ext:streamname`. Основний вміст файлу зберігається у потоці із порожнім іменем (`:$DATA`). Ця функціональність іноді використовується зловмисниками для приховування шкідливого коду у потоках легітимних файлів. Разом із тим ADS використовується Windows цілком легітимно: браузери записують у потік `:Zone.Identifier` зону безпеки завантаженого файлу (значення 3 = Internet), що є підставою для появи діалогу безпеки при відкритті. Виявлення ADS можливе через `dir /R` у cmd, `Get-Item -Stream *` у PowerShell або `streams.exe` із пакету Sysinternals.

Квоти дискового простору є механізмом на перетині управління ресурсами та безпеки. Вони дозволяють обмежити максимальний обсяг дискового простору для конкретного користувача або групи, запобігаючи атакам типу “відмова в обслуговуванні” на рівні зберігання, коли один процес або користувач заповнює весь диск. У Linux квоти підтримуються у ext4 і XFS: задаються “soft limit” (при досягненні якого користувач отримує попередження і має grace period — зазвичай 7 днів — для вивільнення місця) та “hard limit” (абсолютна межа). XFS реалізує квоти ефективніше, зберігаючи їх у власних метаданих, тоді як ext4 використовує окремі файли quota. У ZFS квоти задаються атрибутами набору даних (`zfs set quota=100G pool/home/user`) і є невід’ємною частиною архітектури, не вимагаючи додаткових пакетів.

Шифрування на рівні файлової системи є важливим механізмом захисту конфіденційних даних. Повне шифрування диска (FDE) — BitLocker у Windows або LUKS у Linux — шифрує весь том на рівні блокового пристрою, захищаючи від фізичного викрадення носія. Шифрування на рівні файлової системи — EFS у NTFS або fscrypt у ext4 — дозволяє шифрувати окремі файли або каталоги із прив’язкою ключів до облікових записів, захищаючи дані від інших авторизованих користувачів тієї самої системи. APFS та ZFS мають власне нативне шифрування томів, що є більш інтегрованим і зручним порівняно з LUKS. При використанні будь-якого шифрування критично важливо забезпечити окреме, безпечне зберігання резервних копій ключів шифрування у відповідному сховищі (HSM, KMS або захищене сховище паролів), адже втрата ключа означає незворотну втрату зашифрованих даних.

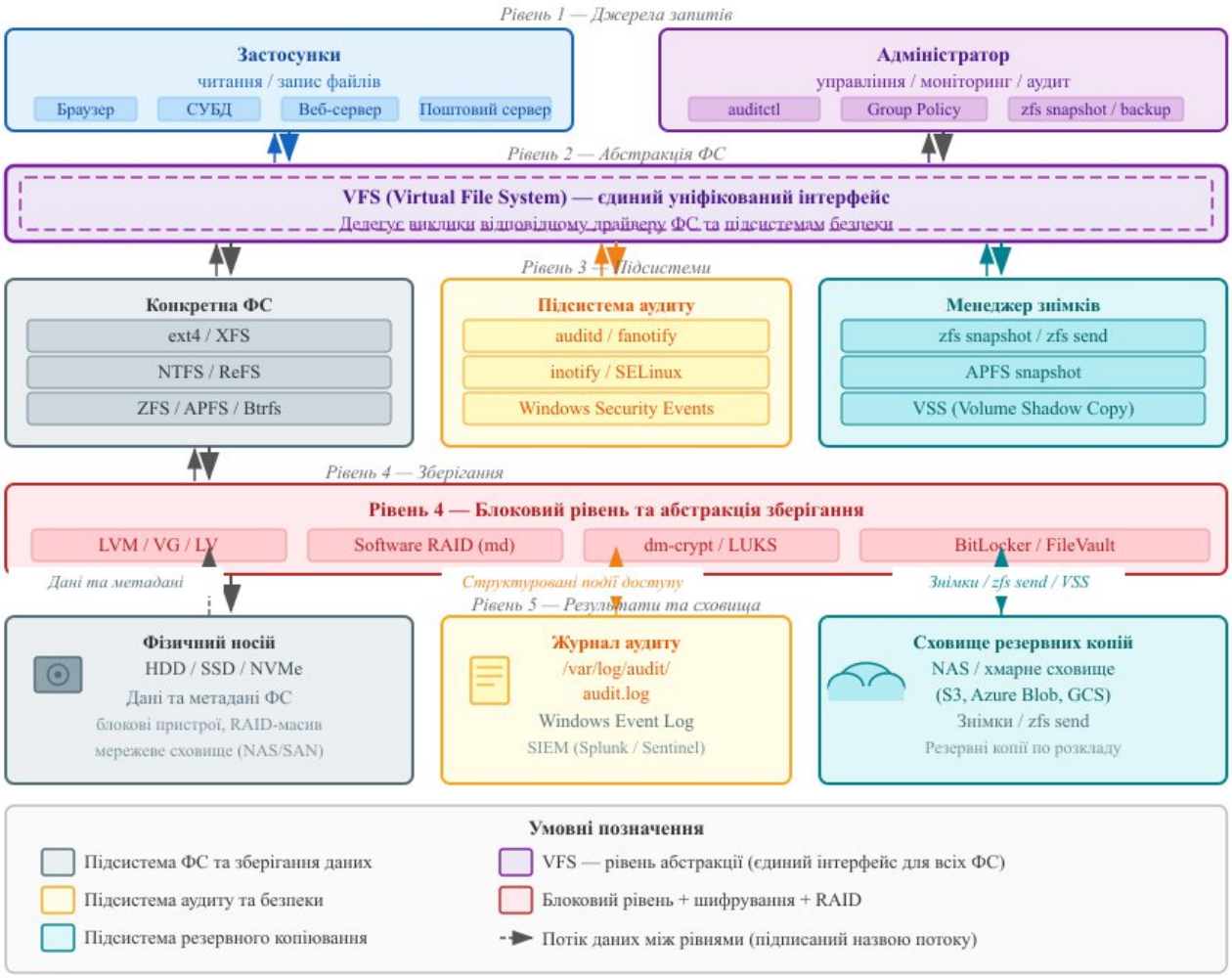

Рисунок 4.6 — Взаємодія компонентів файлової системи із підсистемами безпеки та резервного копіювання

Взаємодія файлової системи із мережевими протоколами додає ще один вимір аналізу безпеки та надійності. Мережеві файлові системи NFS та SMB/CIFS дозволяють монтувати ресурси зберігання на інших вузлах мережі так, ніби вони є локальними, але при цьому виникають специфічні питання: розмежування прав між вузлами (NFS v4 підтримує Kerberos-автентифікацію, а не застарілий UID-based контроль NFS v3), захист від перехоплення трафіку (SMB 3.x підтримує прозоре шифрування каналу командою `Set-SmbServerConfiguration -EncryptData $true`), узгодженість кешованих метаданих на клієнтах. Некоректна конфігурація мережевих файлових систем є частою причиною серйозних інцидентів: відкритий NFS-шер без автентифікації або SMB із застарілим SMBv1 стали векторами атак WannaCry та NotPetya. Регулярний аудит мережевих шерів, відключення застарілих версій протоколів та застосування принципу найменших привілеїв є обов’язковими заходами для будь-якої виробничої інфраструктури.

Таблиця 4.5 — Зведені механізми захисту та аудиту у популярних файлових системах

Механізм	FAT32/exFAT	NTFS	ext4	APFS	ZFS
Журналювання / COW	—	Журналювання	Журналювання	COW	COW
Квоти	—	Так	Так (quota pkg)	—	Так (вбудовані)
Шифрування вбудоване	—	EFS	fscrypt	Так (нативне)	Так (нативне)
Права доступу	—	Повні ACL	POSIX + ACL	POSIX + ACL	POSIX + ACL
Моментальні знімки	—	VSS (зовн.)	—	Так	Так
Перевірка цілісності	—	Часткова	fsck	Часткова	SHA-256 (повна)
Аудит доступу	—	Security Events	auditd	Системний журнал	auditd
TRIM/DISCARD	Так	Так	Так	Так	Так
Альтернативні потоки	—	ADS (повна підт.)	Extended attrs	Extended attrs	Extended attrs
Реплікація	—	DFS / robocopy	rsync	Time Machine	zfs send/recv

Підсумовуючи розгляд теми, варто виокремити кілька ключових висновків. По-перше, файлова система є не просто засобом організації файлів, а повноцінною підсистемою безпеки: вона реалізує контроль доступу, аудит, цілісність та конфіденційність даних. По-друге, жодна конкретна файлова система не є оптимальною для всіх сценаріїв — вибір завжди є компромісом між продуктивністю, надійністю, функціональністю та складністю управління. По-третє, стратегія резервного копіювання та відновлення повинна бути невід’ємною частиною архітектурного проектування системи зберігання, а не додатковою думкою постфактум — правило 3-2-1 (три копії, два різних носії, одна копія поза площадкою) залишається золотим стандартом. Нарешті, сучасний тренд у розвитку файлових систем — перехід від класичного журналювання до copy-on-write із вбудованими знімками, перевіркою цілісності та нативним шифруванням — відображає зростання вимог до надійності та безпеки в критичних інформаційно-комунікаційних системах.

Тема 5. Механізми безпеки та контролю доступу в операційних системах

5.1. Моделі безпеки операційних систем

Безпека операційної системи є фундаментальною властивістю, що визначає здатність системи захищати свої ресурси від несанкціонованого доступу, модифікації чи знищення. Концептуально безпека ОС базується на трьох взаємопов'язаних властивостях, відомих як тріада CIA: конфіденційність (Confidentiality), цілісність (Integrity) та доступність (Availability). Конфіденційність гарантує, що інформація доступна лише авторизованим суб'єктам; цілісність — що дані не можуть бути несанкціоновано змінені; доступність — що авторизовані користувачі можуть отримати доступ до ресурсів у будь-який потрібний момент. Ці три властивості перебувають у певному протиріччі одна з одною: посилення механізмів конфіденційності та цілісності нерідко знижує зручність використання та доступність системи для легітимних користувачів. Розуміння цього балансу є ключовим при проєктуванні та адмініструванні захищених інформаційно-комунікаційних систем.

Для формальної специфікації правил доступу до ресурсів операційних систем було розроблено кілька моделей безпеки. Кожна з них визначає, хто може прийняти рішення про надання доступу, на підставі яких критеріїв це відбувається і як ці рішення зберігаються та виконуються. Вибір моделі безпеки суттєво впливає на гнучкість адміністрування, рівень захищеності та складність реалізації відповідної системи. Розглянемо три найбільш поширені моделі, кожна з яких знайшла широке застосування у реальних операційних системах.

Дискреційна модель контролю доступу (Discretionary Access Control, DAC) є найбільш поширеною і застосовується в більшості традиційних операційних систем, зокрема у UNIX/Linux та Windows. Ключовою характеристикою DAC є те, що власник ресурсу сам вирішує, кому і які права доступу надавати. Наприклад, користувач, який створив файл, може надати доступ до нього іншим користувачам або групам на власний розсуд. У класичній UNIX-моделі права доступу описуються трьома наборами дозволів — для власника, для групи та для всіх інших — кожен з яких включає біти читання (r), запису (w) та виконання (x). Перевагою DAC є його гнучкість та простота управління для кінцевого користувача, проте основний недолік полягає в тому, що якщо обліковий запис власника скомпрометовано, зловмисник отримує всі його права на розподіл доступу, що може призвести до масштабного витоку інформації.

Обов'язкова модель контролю доступу (Mandatory Access Control, MAC) кардинально відрізняється від DAC тим, що рішення про доступ приймаються не власником ресурсу, а централізованою адміністративною політикою, яку користувачі не можуть змінити самостійно. У MAC кожен суб'єкт (процес, користувач) і кожен об'єкт (файл, пристрій) мають мітки безпеки, і доступ надається або забороняється на підставі порівняння цих міток відповідно до

заданих правил. Класичним прикладом MAC-системи є модель Белла-ЛаПадули, де суб'єкти не можуть читати об'єкти з вищим рівнем секретності (no read up) та записувати в об'єкти з нижчим рівнем (no write down), що забезпечує суворий контроль інформаційних потоків. Поряд із моделлю Белла-ЛаПадули існує модель Біба, яка фокусується на цілісності: суб'єкт не може читати дані нижчого рівня цілісності та не може записувати у об'єкти вищого рівня цілісності. У Linux MAC реалізується через підсистеми SELinux та AppArmor, а у Windows — через механізм Mandatory Integrity Control.

Типова ієрархія рівнів секретності, що використовується у MAC-системах, наведена у таблиці 5.2.

Таблиця 5.2 — Типова ієрархія рівнів секретності у MAC-системах

Рівень секретності	Числове значення	Опис	Приклад систем
Top Secret	4	Найвищий рівень, розкриття завдає виключної шкоди	Державні ІС, розвідка
Secret	3	Серйозна шкода при розкритті	Військові командні системи
Confidential	2	Шкода при розкритті	Відомчі інформаційні системи
Unclassified	1	Відкрита або загальнодоступна інформація	Публічні ресурси

Рольова модель контролю доступу (Role-Based Access Control, RBAC) є компромісом між гнучкістю DAC та суворістю MAC і набула широкого поширення у корпоративних середовищах. Замість того щоб призначати права безпосередньо кожному користувачу, RBAC вводить поняття ролі — набору дозволів, що відповідає певній функції в організації. Користувачам призначаються ролі, а не окремі права, що суттєво спрощує адміністрування у великих організаціях: замість управління тисячами індивідуальних пар «користувач — право» адміністратор управляє значно меншою кількістю ролей. RBAC також підтримує принцип поділу обов'язків (separation of duties), що запобігає концентрації критичних повноважень в руках однієї особи — наприклад, одна людина може ініціювати фінансовий платіж, але для його підтвердження потрібна інша особа з відповідною роллю. Розширенням RBAC є модель ABAC (Attribute-Based Access Control), де рішення про доступ приймаються не лише на підставі ролі, а й з урахуванням атрибутів суб'єкта, об'єкта та контексту запиту (наприклад, час доби, IP-адреса, наявність підключення через VPN).

Порівняльні характеристики трьох основних моделей контролю доступу наведено у таблиці 5.1.

Таблиця 5.1 — Порівняльна характеристика моделей контролю доступу

Характеристика	DAC	MAC	RBAC
Хто приймає рішення	Власник ресурсу	Адмін. політика системи	Адміністратор через ролі
Гнучкість для користувача	Висока	Дуже низька	Середня
Складність адміністрування	Низька	Висока	Середня
Стійкість до компрометації	Низька	Висока	Середня
Типові застосування	UNIX, Windows (традиційні)	Військові, держ. ІС	Корпоративні середовища
Реалізація в Linux	chmod/chown, ACL	SELinux, AppArmor	sudo, групи
Реалізація в Windows	DACL в NTFS	MIC, AppLocker	Active Directory ролі
Чи може користувач змінити права?	Так (для своїх ресурсів)	Ні	Ні (тільки адмін)

Принцип найменших привілеїв (Principle of Least Privilege, PoLP) є наскрізним принципом безпеки, що застосовується у всіх моделях. Він вимагає, щоб кожен суб'єкт — процес, користувач або служба — мав лише ті права, які безпосередньо необхідні для виконання його функцій, і не більше. Це обмежує потенційну шкоду від компрометації облікового запису або програми: зловмисник, який отримав контроль над суб'єктом з мінімальними правами, не зможе завдати такої шкоди, як у випадку з привілейованим суб'єктом. На практиці цей принцип реалізується через відмову від роботи під обліковим записом адміністратора в повсякденній діяльності, запуск серверних процесів від імені непривілейованих системних облікових записів та застосування механізму тимчасового підвищення привілеїв лише у разі необхідності. Близьким за змістом є принцип мінімальної поверхні атаки, що вимагає вимикати всі служби, функції та облікові записи, які не використовуються, оскільки кожен додатковий активний компонент є потенційним вектором атаки.

Схема взаємозв'язку між суб'єктами, об'єктами та правилами доступу у різних моделях безпеки наведена на рисунку 5.1.

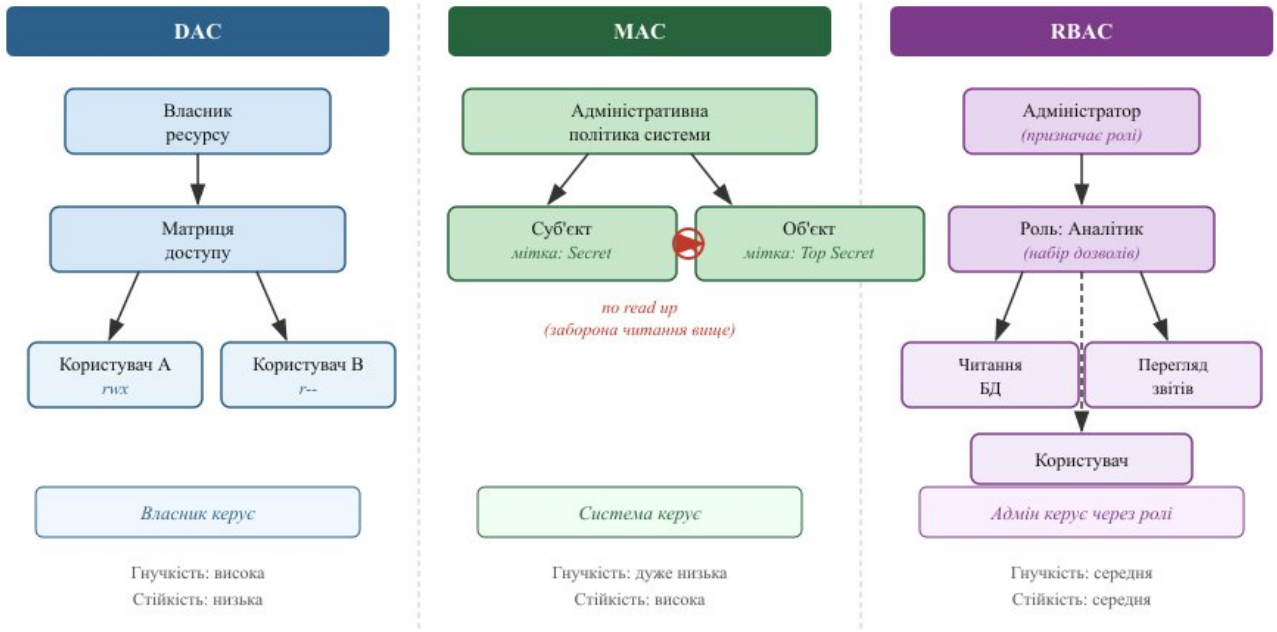

Рисунок 5.1 — Порівняльна схема моделей контролю доступу DAC, MAC та RBAC

5.2. Механізми автентифікації та авторизації

Автентифікація і авторизація є двома нерозривно пов'язаними, але принципово різними процесами, що утворюють основу системи контролю доступу. Автентифікація — це процес підтвердження ідентичності суб'єкта, відповідь на питання «Хто ви?»; авторизація — це процес визначення прав суб'єкта після підтвердження його ідентичності, відповідь на питання «Що вам дозволено робити?». Важливо розуміти, що успішна автентифікація не гарантує автоматичного доступу до ресурсів — вона лише встановлює ідентичність, на підставі якої система потім приймає рішення про авторизацію. Помилкове ототожнення цих двох процесів є поширеним джерелом вразливостей у програмному забезпеченні та системному адмініструванні.

Автентифікація традиційно базується на одному або кількох з трьох факторів: те, що суб'єкт знає (пароль, PIN-код, відповідь на секретне запитання), те, чим суб'єкт є (біометричні дані: відбиток пальця, розпізнавання обличчя, сітківка ока), та те, що суб'єкт має (фізичний токен, смарт-картка, мобільний пристрій). Однофакторна автентифікація, що базується лише на паролі, є найменш надійною, оскільки паролі можуть бути скомпрометовані через перебір, фішинг, витоки з баз даних або шпигунське програмне забезпечення. Двофакторна автентифікація (2FA) суттєво підвищує рівень захисту, комбінуючи два різних фактори: наприклад, пароль і одноразовий код, згенерований мобільним застосунком (TOTP — Time-based One-Time Password). Стандарт TOTP, описаний у RFC 6238, обчислює шестизначний код на основі поточного часу та спільного секретного ключа, термін дії якого становить 30 секунд, що унеможливлює повторне використання перехопленого коду. Сучасні стандарти, зокрема FIDO2/WebAuthn, просувають безпарольну

автентифікацію на основі криптографічних ключів, що усуває цілий клас вразливостей, пов'язаних з паролями.

На рівні операційних систем Linux/Unix автентифікація традиційно реалізується через підсистему PAM (Pluggable Authentication Modules). PAM є модульною системою, яка абстрагує процес автентифікації від конкретних застосунків, дозволяючи змінювати методи автентифікації без модифікації коду програм. Кожен застосунок або системна служба, що вимагає автентифікації, має власний конфігураційний файл у /etc/pam.d/, де описується ланцюжок модулів для чотирьох типів операцій: auth (автентифікація), account (перевірка облікового запису), password (зміна пароля) та session (налаштування сесії). Результати кожного модуля оцінюються відповідно до керуючого прапору, що визначає, чи є невдача цього модуля критичною для всього ланцюжка.

Основні модулі PAM та їх призначення наведено у таблиці 5.3.

Таблиця 5.3 — Основні модулі PAM та їх призначення

Модуль PAM	Призначення	Тип керуючого прапору
pam_unix.so	Автентифікація через локальні паролі (/etc/shadow)	required
pam_ldap.so	Автентифікація через каталог LDAP/Active Directory	sufficient
pam_google_authenticator.so	Перевірка одноразового TOTP-коду (2FA)	required
pam_faillock.so	Блокування облікового запису після N невдалих спроб	required
pam_limits.so	Встановлення ресурсних обмежень сесії (ulimit)	optional
pam_env.so	Встановлення змінних середовища при вході	optional
pam_cracklib.so / pam_pwquality.so	Перевірка складності пароля при його зміні	required
pam_tty_audit.so	Увімкнення аудиту команд у терміналі	optional

У корпоративних середовищах автентифікація та авторизація, як правило, централізуються через каталогові служби. Найпоширенішим рішенням у Windows-середовищах є Active Directory (AD) — ієрархічна база даних об'єктів (користувачів, комп'ютерів, груп, політик), що використовує протокол Kerberos для автентифікації та LDAP для запитів. Об'єкти в AD організовані в ієрархічну структуру: ліс (forest) → домен (domain) → організаційний підрозділ (OU, Organizational Unit). Групові політики (GPO, Group Policy Objects) дозволяють централізовано управляти параметрами безпеки сотень і тисяч комп'ютерів: вимагати певну мінімальну довжину пароля, вмикати екранну заставку з

блокуванням, обмежувати запуск певних застосунків тощо. LDAP (Lightweight Directory Access Protocol) використовується і в Linux-середовищах через OpenLDAP, забезпечуючи централізоване зберігання облікових записів у гетерогенних середовищах.

Схема автентифікації Kerberos у корпоративному середовищі наведена на рисунку 5.2.

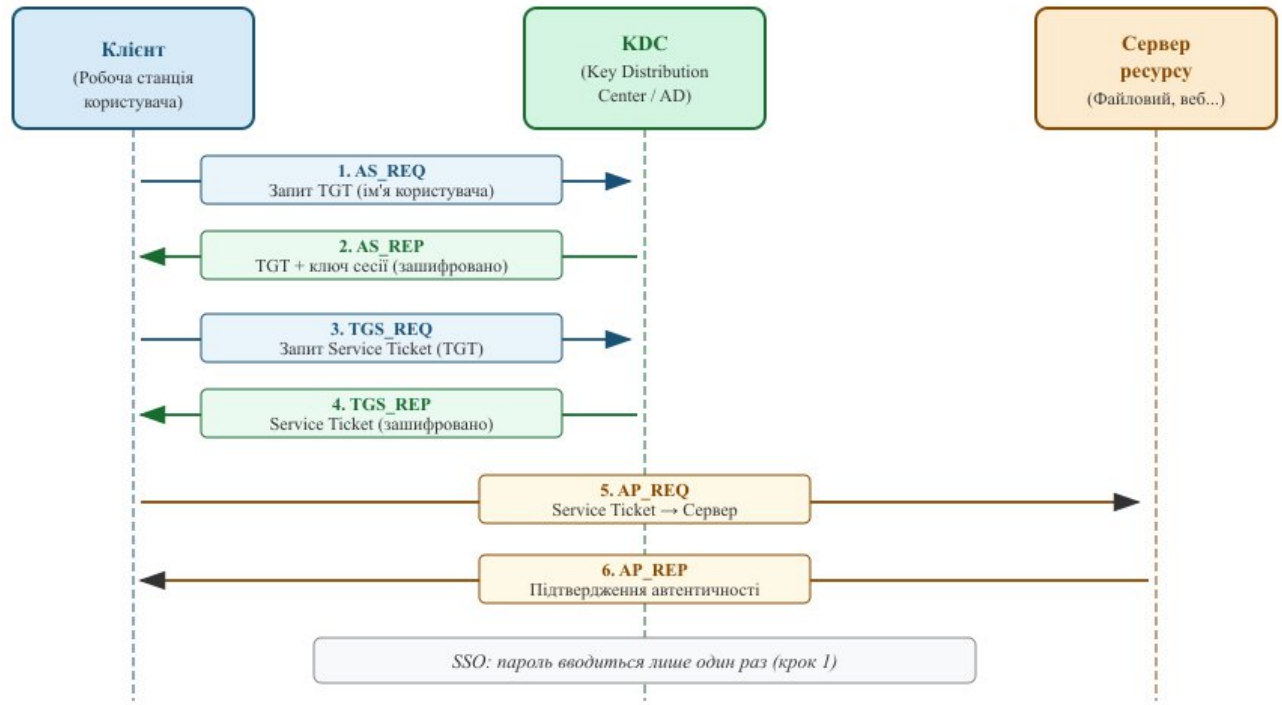

Рисунок 5.2 — Схема автентифікації Kerberos у корпоративному середовищі

На рисунку 5.2 зображено взаємодію між трьома учасниками протоколу Kerberos: клієнтом (робочою станцією користувача), сервером автентифікації KDC (Key Distribution Center, що є частиною контролера домену Active Directory) та цільовим сервером ресурсу. Схема розміщена горизонтально: клієнт ліворуч, KDC у центрі, сервер ресурсу праворуч. Від клієнта до KDC позначено стрілку «1. AS_REQ: запит TGT (ім'я користувача)»; від KDC до клієнта — «2. AS_REP: TGT + ключ сесії (зашифровано ключем користувача)». Далі від клієнта до KDC — «3. TGS_REQ: запит Service Ticket (TGT + назва сервісу)»; від KDC до клієнта — «4. TGS_REP: Service Ticket (зашифровано ключем сервісу)». Нарешті від клієнта до сервера ресурсу — «5. AP_REQ: Service Ticket»; від сервера до клієнта — «6. AP_REP: підтвердження автентичності». Схема ілюструє, як SSO реалізується без повторного введення пароля.

Управління обліковими записами є критичним аспектом безпеки ОС. У Linux інформація про облікові записи зберігається у файлах /etc/passwd (публічна інформація: ім'я, UID, GID, домашній каталог, командний інтерпретатор) та /etc/shadow (хеші паролів та параметри їх старіння, доступні лише для привілейованих процесів). Зберігання паролів у відкритому вигляді є

неприпустимим; замість цього використовуються криптографічні хеш-функції. Сучасні стандарти рекомендують адаптивні хеш-функції, такі як bcrypt, scrypt або Argon2, які навмисно є обчислювально витратними завдяки параметрам вартості (cost factor), що робить атаки методом перебору непрактичними. У Windows функція NTLM-хешування вважається застарілою через атаки типу Pass-the-Hash та Pass-the-Ticket, тому рекомендується її вимкнення на користь виключного використання Kerberos.

Концепція маркера доступу (access token) є ключовим елементом реалізації авторизації на рівні ОС. Після успішної автентифікації операційна система створює маркер доступу — структуру даних, що містить ідентифікатор користувача (SID у Windows, UID у Linux), список груп, до яких він належить, набір привілеїв та іншу контекстну інформацію безпеки. Цей маркер асоціюється з усіма процесами, запущеними від імені даного користувача, і система звіряється з ним при кожному зверненні до захищеного ресурсу — перевіряє, чи є необхідний дозвіл у маркері. Механізм UAC у Windows реалізує концепцію розщепленого маркера: звичайний користувач-адміністратор отримує при вході два маркери — стандартний (filtered token) з обмеженими правами та підвищений (elevated token) з повними правами, причому підвищений маркер використовується лише при явному підтвердженні запиту на підвищення привілеїв. В Linux аналогічна функціональність досягається через sudo, який запускає процес від імені root, попередньо перевіривши права поточного користувача у файлі /etc/sudoers.

5.3. Контроль доступу на рівні файлів, процесів та мережі

Практична реалізація контролю доступу охоплює кілька взаємодоповнювальних рівнів: файлову систему, процеси та міжпроцесну взаємодію, а також мережеві з'єднання. Кожен з цих рівнів має власні механізми захисту, які разом утворюють комплексну систему безпеки операційної системи. Відсутність захисту хоча б на одному з цих рівнів може призвести до компрометації всієї системи, навіть якщо інші рівні добре захищені. Принцип глибокоешелонованої оборони (defense in depth) вимагає саме такого багаторівневого підходу, де кожен захисний рівень є незалежним і не покладається на те, що попередній рівень вже виконав всю необхідну роботу.

Контроль доступу до файлів є найбільш очевидним аспектом безпеки ОС. У Linux традиційна UNIX-модель прав доступу визначає три категорії суб'єктів (власник, група, інші) та три типи дій (читання, запис, виконання). Для файлів «виконання» означає запуск програми, для каталогів — можливість переходу в нього та звернення до об'єктів всередині. Додатково існують спеціальні біти: SUID (Set User ID) дозволяє виконувати програму з привілеями власника файлу замість привілеїв запускаючого користувача — цей механізм використовується командою passwd для запису в /etc/shadow; SGID (Set Group ID) аналогічно встановлює ефективний GID при виконанні або, якщо встановлений на каталог, змушує нові файли успадковувати групу каталогу; Sticky Bit на каталозі

запобігає видаленню файлів іншими користувачами, навіть якщо вони мають права запису у каталог. SUID-програми є потенційним вектором атаки при наявності вразливостей у них, тому їх кількість слід регулярно переглядати командою find / -perm -4000.

Списки контролю доступу (ACL, Access Control Lists) розширюють класичну UNIX-модель, дозволяючи визначати права для довільної кількості користувачів і груп, не обмежуючись трьома стандартними категоріями. У Linux ACL реалізовані через розширені атрибути файлової системи та управляються командами setfacl/getfacl. Наприклад, команда setfacl -m u:bob:r-- /srv/data/report.txt надає користувачу bob право читання конкретного файлу без зміни його власника чи групи. ACL підтримуються файловими системами ext4, XFS, Btrfs та іншими. У Windows аналогом є DACL (Discretionary ACL) — список записів ACE (Access Control Entry), кожен з яких визначає права доступу або явну заборону для конкретного SID. DACL підтримує успадкування: права, встановлені на батьківській теці, автоматично поширюються на дочірні об'єкти, що суттєво спрощує управління правами у великих ієрархічних структурах.

Архітектура системи контролю доступу Linux з кількома шарами захисту наведена на рисунку 5.3.

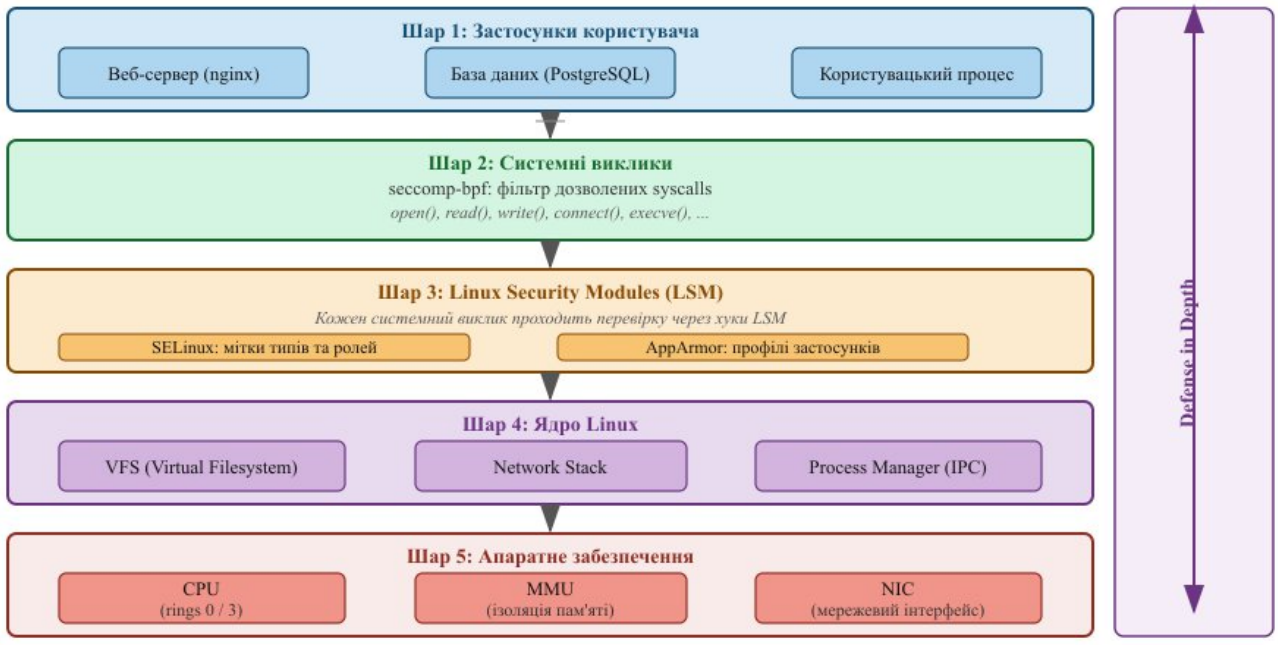

Рисунок 5.3 — Архітектура системи контролю доступу Linux (багаторівневий захист)

На рисунку 5.3 зображено ієрархічну схему шарів захисту операційної системи Linux у вигляді стопки горизонтальних прямокутників. Верхній шар — «Застосунки користувача» містить три блоки: «Веб-сервер», «База даних», «Користувацький процес». Другий шар — «Системні виклики» позначений написом «seccomp-bpf: фільтр дозволених syscalls» з блоком-переліком прикладів: «open(), read(), write(), connect()». Третій шар — «Linux Security Modules (LSM)» містить два паралельних блоки: «SELinux: мітки типів та

ролей» та «AppArmor: профілі застосунків». Четвертий шар — «Ядро Linux» з блоками «VFS», «Network Stack», «IPC». П'ятий, найнижчий шар — «Апаратне забезпечення» з блоками «CPU (rings 0/3)», «MMU (ізоляція пам'яті)» та «NIC». Стрілки між шарами спрямовані вниз (запити) та вгору (відповіді). Праворуч від схеми вертикальний напис «Defense in Depth» із двоголовою стрілкою. Схема ілюструє принцип, за яким кожен запит від застосунку проходить крізь усі шари перевірок перед доступом до апаратних ресурсів.

Захист процесів ґрунтується на концепції ізоляції адресних просторів та контролі міжпроцесної взаємодії. Кожен процес виконується у власному віртуальному адресному просторі, ізольованому від інших процесів апаратним механізмом MMU (Memory Management Unit), що запобігає несанкціонованому читанню або модифікації пам'яті. Системні виклики, що забезпечують міжпроцесну взаємодію (сигнали, pipes, shared memory, Unix sockets), підлягають перевіркам безпеки: в Linux надсилання сигналу процесу дозволено лише якщо обидва процеси мають однакового власника або якщо надсилач має права суперкористувача. Механізм seccomp (Secure Computing Mode) дозволяє процесу обмежити набір системних викликів, доступних йому самому та його нащадкам; у режимі BPF (Berkeley Packet Filter) можна задавати складні правила фільтрації. Браузер Chrome, наприклад, використовує seccomp-bpf для ізоляції процесів рендерингу, обмежуючи їх лише необхідним набором системних викликів.

Мережевий контроль доступу реалізується передусім через брандмауери та мережеві фільтри. У Linux вбудований брандмауер реалізований через netfilter — підсистему ядра, що дозволяє фільтрувати, перенаправляти та модифікувати мережеві пакети на різних стадіях їх обробки (ланцюжки: PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING). Традиційним інструментом управління netfilter є iptables, хоча сучасні дистрибутиви переходять на nftables з більш гнучким та уніфікованим синтаксисом для IPv4, IPv6 та фільтрації за MAC-адресами. Stateful Packet Inspection (SPI) дозволяє відстежувати стан TCP-з'єднань та дозволяти або блокувати пакети на підставі контексту з'єднання, а не лише окремих атрибутів пакету. У Windows аналогом є Windows Defender Firewall з підтримкою правил для конкретних застосунків, портів і профілів мережі (Domain, Private, Public).

Огляд ключових механізмів контролю доступу на різних рівнях у Linux та Windows наведено у таблиці 5.4.

Таблиця 5.4 — Механізми контролю доступу в Linux та Windows за рівнями

Рівень	Linux	Windows
Файлова система (базовий)	chmod/chown, POSIX permissions	NTFS DACL
Файлова система (розширений)	POSIX ACL (setfacl/getfacl)	NTFS ACL, SACL
Обов'язковий контроль	SELinux, AppArmor	Mandatory Integrity Control

(MAC)		
Процеси та ізоляція	namespaces, capabilities, seccomp	Job Objects, процесні токени
Мережа	iptables/nftables, SELinux	Windows Firewall, AppLocker
Ізоляція застосунків	seccomp-bpf, namespaces, cgroups	AppContainer, Windows Sandbox
Підвищення привілеїв	sudo, SUID, capabilities	UAC, Run as Administrator
Контейнеризація	Docker (namespaces + cgroups)	Windows Containers, Hyper-V Isolation

Механізм Linux Capabilities (можливостей) є важливим інструментом для дотримання принципу найменших привілеїв на рівні процесів. Традиційно у UNIX існувало лише бінарне розмежування між привілейованим (root, UID=0) та непривілейованим процесом — все або нічого. Механізм capabilities, введений у ядрі Linux 2.2, розбиває привілеї суперкористувача на 37 незалежних одиниць, кожну з яких можна незалежно надавати або позбавляти. Основні capabilities наведено у таблиці 5.5.

Таблиця 5.5 — Основні Linux Capabilities та їх застосування

Capability	Що дозволяє	Типове застосування
CAP_NET_BIND_SERVICE	Прив'язка до портів < 1024	Веб-сервери (nginx, Apache)
CAP_NET_ADMIN	Управління мережевими інтерфейсами та маршрутами	Брандмауери, VPN-демони
CAP_SYS_PTRACE	Відстеження та зміна інших процесів через ptrace	Відладчики (gdb, strace)
CAP_SYS_ADMIN	Широкий набір адмін. операцій (монтування, namespace)	Системні утиліти
CAP_KILL	Надсилання сигналів довільним процесам	Менеджери процесів
CAP_DAC_OVERRIDE	Обхід перевірок дискреційного контролю доступу	Backup-агенти
CAP_CHOWN	Зміна власника файлу	Інсталятори програм
CAP_SYS_TIME	Зміна системного годинника	NTP-демони

Важливою концепцією безпеки на рівні процесів є також sandbox-ізоляція — технологія обмеження дій процесу рамками суворо визначеного середовища виконання. У Linux пісочниці реалізуються через комбінацію namespaces, cgroups, seccomp та capabilities. Зокрема, механізм namespaces дозволяє ізолювати окремі аспекти системи: PID namespace ізолює простір ідентифікаторів процесів, net namespace — мережевий стек, mount namespace — точки монтування файлових систем, user namespace — ідентифікатори користувачів. Контейнерні рушії, такі як Docker та Podman, активно використовують ці механізми для ізоляції контейнерів. При цьому слід пам'ятати, що контейнери, на відміну від повноцінних віртуальних машин, поділяють ядро хост-системи, тому вразливість ядра може призвести до виходу з контейнера та компрометації хосту.

5.4. Аудит та журналізація подій безпеки

Аудит та журналізація подій є невід'ємними компонентами системи безпеки операційної системи, що виконують кілька критичних функцій: виявлення інцидентів безпеки в режимі реального часу або постфактум, забезпечення доказової бази для розслідування інцидентів та виконання регуляторних вимог (зокрема PCI DSS, HIPAA, ISO 27001, НД ТЗІ). Без належного аудиту неможливо ані виявити атаку, ані провести повноцінне розслідування після її виявлення — зловмисник може роками мати несанкціонований доступ до системи, якщо відсутня система моніторингу. Ефективна система журналізації повинна бути стійкою до маніпуляцій з боку зловмисника, забезпечувати цілісність і незаперечність записів та надавати достатньо деталей для відновлення повного ланцюжка подій (forensic readiness).

У Linux центральним інструментом аудиту безпеки є підсистема auditd — демон аудиту, тісно інтегрований з ядром через netlink-сокет. Ядро Linux містить аудит-підсистему, яка перехоплює системні виклики та інші події і надсилає їх демону auditd для обробки та запису у /var/log/audit/audit.log. Конфігурація аудиту визначається правилами auditctl, що зберігаються у /etc/audit/rules.d/: можна вказати конкретні системні виклики для відстеження (наприклад, execve, open, connect), файли або каталоги для моніторингу змін, дії конкретних користувачів. Кожен запис аудиту містить: тип події (SYSCALL, PATH, EXECVE, USER_AUTH тощо), часову мітку з мікросекундною точністю, UID, EUID, GID, ім'я виконаної програми (comm), аргументи системного виклику та результат. Утиліта ausearch підтримує пошук за ключовими словами, часовими діапазонами та типами подій, а ausreport формує зведені статистичні звіти.

Системний журнал syslog та його наступник systemd-journald фіксують широкий спектр системних подій. Традиційний rsyslog збирає повідомлення від ядра та демонів і маршрутизує їх за парою facility/severity: наприклад, auth.warning позначає попередження від підсистеми автентифікації, kern.crit — критичні повідомлення ядра. Systemd-journald зберігає журнали у структурованому бінарному форматі, що дозволяє ефективно фільтрувати

записи за будь-яким полем: journalctl -u nginx --since yesterday виводить журнал сервісу nginx за вчорашній та сьогоднішній день. Структуровані журнали також підтримують довільні метаданні, що спрощує їх автоматичну обробку SIEM-системами. Ключовим налаштуванням безпеки є пересилання журналів у режимі реального часу на виділений сервер через зашифровані канали (TLS): у разі компрометації хосту зловмисник не зможе видалити вже передані записи.

Схема централізованої системи журналізації та моніторингу безпеки наведена на рисунку 5.4.

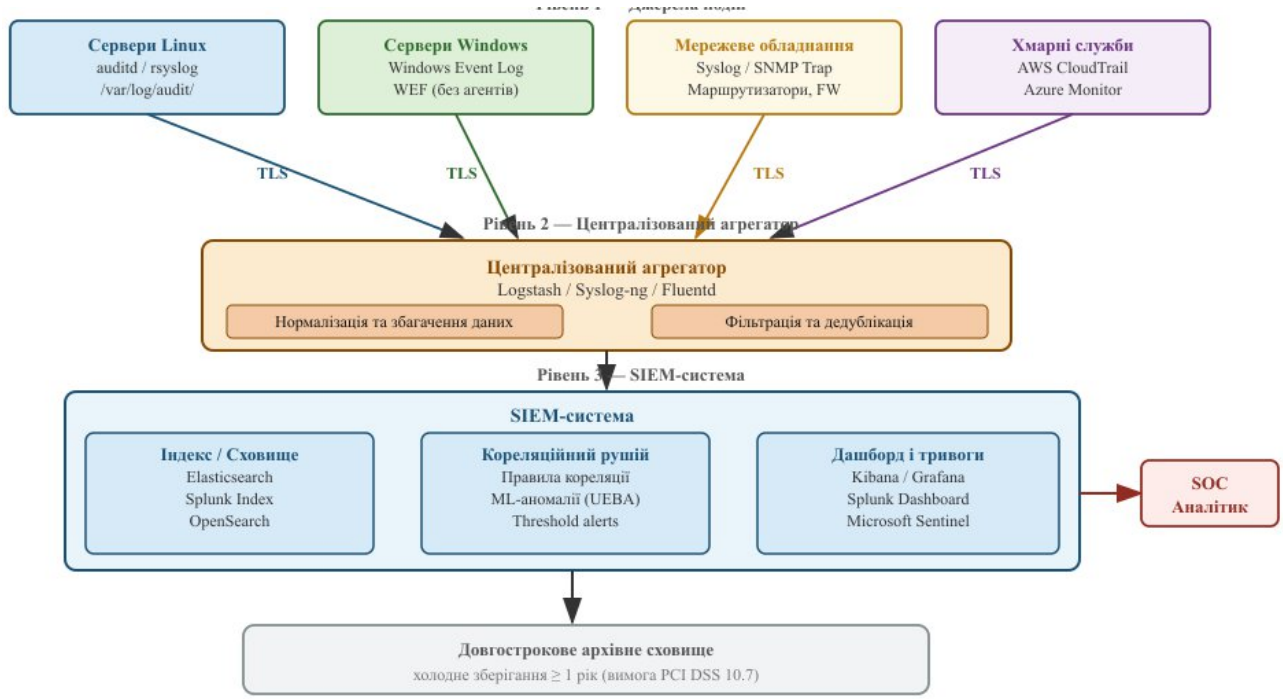

Рисунок 5.4 — Архітектура централізованої системи журналізації та моніторингу безпеки (SIEM)

У Windows аудит реалізується через підсистему Local Security Authority (LSA) та Windows Event Log. Налаштування аудиту здійснюється через Групові Політики: розділ Computer Configuration → Windows Settings → Security Settings → Advanced Audit Policy Configuration містить дев'ять категорій аудиту, кожна з яких може бути окремо увімкнена для успіху, невдачі або обох варіантів подій. Журнали подій Windows поділяються на канали Security, System, Application та численні спеціалізовані журнали Microsoft-Windows-*, де зберігаються деталізовані події окремих підсистем (PowerShell, Task Scheduler, WMI тощо). Журнал PowerShell/Operational є особливо важливим: він фіксує весь виконаний код PowerShell, включно з декодованими base64-рядками та командами, що виконувалися через Invoke-Expression — типовий прийом зловмисників для приховування шкідливого коду.

Ключові події журналу Security Windows та їх значення для виявлення загроз наведені у таблиці 5.6.

Таблиця 5.6 — Ключові події журналу Security Windows та їх значення для безпеки

Event ID	Назва події	Категорія аудиту	Значення для безпеки
4624	Успішний вхід в систему	Logon/Logoff	Базова активність; аномалії — ознака компрометації
4625	Невдала спроба входу	Logon/Logoff	Brute-force; множинні записи — тривожний сигнал
4648	Вхід з явно вказаними обліковими даними	Logon/Logoff	Pass-the-Hash атаки, lateral movement
4672	Призначення спеціальних привілеїв при вході	Privilege Use	Вхід привілейованих облікових записів
4688	Створення нового процесу	Process Tracking	Виявлення аномального запуску, malware
4698	Створення задачі планувальника	Object Access	Механізм persistence зловмисників
4720	Створення облікового запису	Account Management	Несанкціоноване backdoor-акаунти
4732	Додавання члена до привілейованої групи	Account Management	Несанкціоноване підвищення привілеїв
4771	Невдала попередня автентифікація Kerberos	Account Logon	Атаки на AD, password spray
7045	Встановлення нового сервісу	System	Встановлення шкідливих сервісів

Концепція SIEM (Security Information and Event Management) визначає підхід до централізованого збору, нормалізації, кореляції та аналізу журналів з багатьох різнорідних джерел. SIEM-система застосовує правила кореляції для виявлення підозрілих паттернів, що не є очевидними при аналізі окремого джерела. Наприклад, одинична невдала спроба входу є звичайною подією, однак двісті невдалих спроб з однієї IP-адреси протягом хвилини — це очевидна brute-force атака, і SIEM повинна автоматично ініціювати блокування. Більш складні атаки виявляються через ланцюжки подій: успішний вхід о третій ночі → запуск PowerShell → підключення до зовнішнього IP → масовий доступ до файлів із чутливими даними — ця послідовність сигналізує про компрометацію облікового запису та можливий витік даних. Сучасні SIEM-рішення доповнюються алгоритмами машинного навчання для виявлення аномалій поведінки (UEBA, User and Entity Behavior Analytics), що дозволяє виявляти нові типи атак, які не охоплені наявними правилами.

Цілісність та захист самих журналів є критичним аспектом безпеки, на який нерідко не звертають належної уваги. Якщо зловмисник може

модифікувати або видалити журнали після компрометації системи, він може повністю приховати сліди своєї активності, унеможлививши розслідування інциденту. Для захисту журналів застосовується комплекс заходів: жорстке обмеження прав на запис у файли журналів (лише відповідний демон), пересилання журналів у режимі реального часу на ізольований сервер збору з мінімальним доступом, криптографічне підписання блоків записів для виявлення їх підробки, а також централізоване зберігання у SIEM з незмінним (immutable) сховищем. Відповідно до вимог PCI DSS 10.7, організації зобов'язані зберігати журнали аудиту протягом не менше одного року, забезпечуючи негайний доступ до даних за останні три місяці.

Важливим принципом організації аудиту є вибірковість та пріоритизація. Не слід журналювати абсолютно всі події, оскільки це призводить до колосальних обсягів даних та зниження продуктивності системи, а в результаті — до того, що аналітик потоне у шумі та пропустить реально важливі події. Натомість необхідно зосередитися на категоріях подій з найвищою цінністю для безпеки: автентифікаційні події (як успішні, так і невдалі), операції з привілейованими обліковими записами, зміни конфігурації та прав доступу, запуск нових процесів та сервісів, доступ до чутливих файлів та ресурсів, аномальна мережева активність. Ці рекомендації систематизовані у стандартах, зокрема CIS Benchmarks та NIST SP 800-92 «Guide to Computer Security Log Management», які є практичним орієнтиром для побудови ефективної системи аудиту.

Підсумовуючи цю тему, слід підкреслити, що безпека операційної системи є комплексним завданням, яке не може бути вирішене єдиним механізмом або засобом. Ефективний захист ІКС будується на принципі глибокоешелонованої оборони (defense in depth): кожен з розглянутих рівнів — моделі контролю доступу, автентифікація, контроль доступу на рівні файлів, процесів та мережі, аудит та журналізація — доповнює інші, і компрометація одного рівня не призводить до миттєвого захоплення всієї системи. Регулярний перегляд та оновлення конфігурацій безпеки, своєчасне встановлення патчів, мінімізація поверхні атаки через вимкнення непотрібних служб та неухильне слідування принципу найменших привілеїв утворюють основу сталої безпеки операційних систем у виробничому середовищі. Зі зростанням складності сучасних ІКС і постійною еволюцією кіберзагроз відповідальне ставлення до безпеки ОС стає не просто технічною вимогою, а стратегічним пріоритетом для будь-якої організації.

Тема 6. Управління конфігурацією та автоматизація системних завдань

6.1. Роль скриптів та систем автоматизації в підтримці операційної системи

Сучасні інформаційно-комунікаційні системи характеризуються значною складністю та динамічністю: вони охоплюють десятки чи сотні вузлів, різноманітне програмне забезпечення та постійно змінювані вимоги до безпеки і продуктивності. В таких умовах ручне управління конфігурацією стає не лише неефективним, а й небезпечним — людський фактор призводить до помилок, неузгодженостей між вузлами та вразливостей, що важко виявити і усунути. Саме тому управління конфігурацією та автоматизація системних завдань є одними з фундаментальних дисциплін системного адміністрування та інженерії надійності (Site Reliability Engineering, SRE). Курс операційних систем розглядає цю тему як критично важливу для забезпечення стабільності, відтворюваності та безпеки ОС у виробничому середовищі.

Управління конфігурацією (Configuration Management, CM) — це практика підтримки систем у відомому, бажаному та стабільному стані протягом усього їхнього життєвого циклу. Воно передбачає не лише початкове налаштування ОС, а й відстеження змін, їх документування, повторне застосування та відкат у разі потреби. Тісно пов’язана з CM концепція автоматизації системних завдань охоплює планування регулярних операцій — резервного копіювання, оновлення пакетів, сканування журналів, перезапуску сервісів — без безпосередньої участі адміністратора. Разом ці практики формують основу для впровадження підходів Infrastructure as Code (IaC), де стан інфраструктури описується у вигляді коду, зберігається у системах версійного контролю та може бути розгорнутий автоматично.

Особливої актуальності ці підходи набувають в умовах DevOps і DevSecOps — методологій, що передбачають тісну інтеграцію між командами розробки, операцій та безпеки. Традиційна межа між «написати програму» і «розгорнути та підтримувати програму» стає дедалі більш розмитою: сучасні практики вимагають, щоб стан середовища виконання описувався, тестувався та версіонувалось із тією ж ретельністю, що й сам код застосунку. Дослідження State of DevOps, що регулярно проводиться DORA (DevOps Research and Assessment), незмінно демонструє кореляцію між зрілістю практик конфігураційного управління та ключовими показниками ефективності IT-організацій: часом розгортання нових версій, частотою інцидентів та часом відновлення після збоїв. Розуміння принципів управління конфігурацією та автоматизації є тому обов’язковим для будь-якого фахівця у сфері IT, незалежно від того, обіймає він роль системного адміністратора, інженера DevOps, спеціаліста з безпеки чи розробника програмного забезпечення.

Перш ніж з’явились спеціалізовані системи управління конфігурацією, основним інструментом автоматизації в Unix-подібних системах були командні інтерпретатори — shell-скрипти. Bash (Bourne Again Shell) і досі залишається найпоширенішим засобом написання сценаріїв автоматизації в Linux-

середовищах, тоді як PowerShell займає аналогічне місце в екосистемі Windows. Скрипт являє собою послідовність команд, що виконуються інтерпретатором, і може охоплювати будь-які дії: від простого копіювання файлів до складних багатокрокових операцій із розгалуженнями, циклами та обробкою помилок. Перевага скриптів перед ручним виконанням команд полягає в їх відтворюваності — один і той самий скрипт завжди виконує одні й ті ж дії в одному й тому ж порядку, що усуває варіативність, притаманну ручним операціям.

Bash-скрипт починається з так званого «shebang» — рядка `#!/bin/bash`, що повідомляє ядру, яким інтерпретатором слід виконувати файл. Після цього йдуть команди, що можуть використовувати змінні, умовні конструкції (`if`, `case`), цикли (`for`, `while`), функції та підстановку результатів виконання команд. Скрипт може отримувати аргументи з командного рядка через спеціальні змінні `$1`, `$2` тощо, що робить його параметризованим та придатним для різних сценаріїв застосування. Для обробки помилок широко використовуються конструкції перевірки кодів виходу (`$?`), команди `set -e` (завершення при будь-якій помилці) та `set -u` (завершення при використанні неоголошеної змінної). Скрипт, написаний без належної обробки помилок, може продовжувати виконання після збою однієї з операцій, залишаючи систему в непослідовному стані — це одна з найпоширеніших причин тяжких інцидентів, спричинених «безневинними» скриптами обслуговування.

PowerShell, запроваджений Microsoft у 2006 році та відкритий під назвою PowerShell Core (починаючи з версії 6.0), суттєво відрізняється від традиційних командних інтерпретаторів: він працює не з текстовим виводом, а з об’єктами .NET, що дозволяє передавати між командами структуровані дані без потреби в текстовому розборі. Це робить PowerShell особливо потужним для автоматизації в Windows-середовищах: адміністратори можуть керувати Active Directory, Exchange, SQL Server, Azure та локальними ресурсами ОС через єдиний інтерфейс. PowerShell DSC (Desired State Configuration) являє собою вбудований механізм декларативного управління конфігурацією, де адміністратор описує бажаний стан системи, а платформа самостійно приводить реальний стан у відповідність до описаного. PowerShell також підтримує модульну архітектуру: екосистема PowerShell Gallery містить тисячі готових модулів для будь-яких задач адміністрування, що значно прискорює розробку автоматизованих рішень у гетерогенних Windows-середовищах.

Python як мова написання сценаріїв системного адміністрування набуває дедалі більшої популярності завдяки своїй зрозумілості, великій стандартній бібліотеці та широкій екосистемі сторонніх пакетів. Стандартна бібліотека Python містить модулі для роботи з файловою системою (`os`, `pathlib`), мережею (`socket`, `urllib`), процесами (`subprocess`), архівами (`zipfile`, `tarfile`) та форматами конфігураційних файлів (`configparser`, `json`, `yaml`). На відміну від shell-скриптів, Python-сценарії легше тестувати, розширювати та підтримувати, що робить їх кращим вибором для складних задач автоматизації, де логіка виходить за рамки послідовного виконання команд. Крос-платформеність Python — можливість запускати той самий скрипт на Linux, macOS та Windows

— є додатковою перевагою у гетерогенних середовищах, де необхідно підтримувати уніфіковану базу коду для різних ОС.

Незалежно від мови написання, якісний скрипт автоматизації повинен відповідати низці вимог: бути самодокументованим (містити коментарі та описові імена змінних), ідемпотентним (повторний запуск не повинен спричиняти небажаних ефектів), стійким до помилок (перевіряти передумови перед виконанням та коректно обробляти збої) та безпечним (не зберігати паролі у відкритому вигляді, правильно обробляти права доступу до файлів). Документування призначення, параметрів та прикладів використання скрипта є особливо важливим у командному середовищі, де ним користуватимуться інші люди або майбутні версії вас самих. Зберігання скриптів у системі версійного контролю, написання до них автоматизованих тестів та проходження code review перед впровадженням у виробниче середовище — ці практики перетворюють скрипт із одноразового «хака» на надійний інструмент, що може бути частиною виробничої інфраструктури.

З ростом масштабів інфраструктури ручне написання та підтримка скриптів для кожного вузла стає практично нездійсненним. Для вирішення цієї проблеми були розроблені спеціалізовані системи управління конфігурацією та оркестрації. Найпоширенішими серед них є Ansible, Puppet, Chef та SaltStack, кожна з яких реалізує власний підхід до опису та застосування конфігурацій. Спільним для всіх цих систем є поняття «бажаного стану» (desired state): адміністратор описує не послідовність дій, а те, яким має бути кінцевий стан системи, а інструмент самостійно визначає необхідні кроки для досягнення цього стану. Цей декларативний підхід принципово відрізняється від імперативного, характерного для традиційних скриптів, і знаменує концептуальний стрибок у способі мислення про управління системами.

Ansible вирізняється серед конкурентів своєю архітектурою без агентів (agentless): він використовує SSH для підключення до вузлів та не вимагає встановлення додаткового програмного забезпечення на керованих машинах. Конфігурації описуються у форматі YAML у так званих «playbook» (книгах відтворення), що містять набір «plays» — блоків завдань, спрямованих на певну групу вузлів. Ansible виконує завдання послідовно та за замовчуванням є ідемпотентним: більшість вбудованих модулів перевіряють поточний стан і виконують дії лише у разі відхилення від бажаного, повідомляючи про факт зміни через механізм «changed/ok/failed». Ansible Galaxy — централізоване сховище готових ролей та колекцій — дозволяє повторно використовувати перевірені конфігурації від спільноти, значно прискорюючи розробку нових автоматизованих рішень.

Puppet і Chef реалізують модель «тягни» (pull): агент, встановлений на кожному вузлі, з певною переодичністю звертається до центрального сервера, завантажує актуальну конфігурацію та застосовує її. Puppet використовує власну декларативну мову (Puppet DSL), тоді як Chef — мову Ruby та концепцію «cookbook» (кулінарних книг), що описують «рецепти» налаштування. Ця модель добре масштабується та забезпечує постійне відстеження конфігураційного дрейфу — поступового відхилення реального

стану вузлів від заданого еталона. SaltStack поєднує модель «тягни» з можливістю синхронного виклику з керуючого вузла (push), що дозволяє реагувати на події в режимі реального часу. SaltStack також підтримує систему подій та реакторів, завдяки якій система може автоматично відповідати на зміни в інфраструктурі без втручання адміністратора — наприклад, автоматично налаштовувати новий вузол при його реєстрації в системі.

У таблиці 6.1 наведено порівняльну характеристику розглянутих систем управління конфігурацією за ключовими параметрами, що дозволяє зорієнтуватись у виборі інструменту залежно від специфіки завдань.

Таблиця 6.1 — Порівняльна характеристика систем управління конфігурацією

Характеристика	Ansible	Puppet	Chef	SaltStack
Архітектура	Agentless (SSH)	Agent + сервер	Agent + сервер	Agent + сервер (+ push)
Мова опису	YAML (Playbook)	Puppet DSL	Ruby (DSL)	YAML / Jinja2
Модель взаємодії	Push	Pull	Pull	Pull + Push
Ідемпотентність	Вбудована	Вбудована	Вбудована	Вбудована
Складність освоєння	Низька	Середня	Висока	Середня
Масштабованість	Середня	Висока	Висока	Висока
Підтримка Windows	Часткова	Так	Так	Часткова
Веб-інтерфейс	AWX / Tower	Puppet Enterprise	Chef Automate	SaltStack Config
Реакція на події	Обмежена	Обмежена	Обмежена	Розвинута

Вибір інструменту залежить від розміру інфраструктури, наявних компетенцій команди та специфічних вимог до безпеки. В середніх і малих організаціях Ansible часто є оптимальним вибором через низький поріг входу та відсутність необхідності підтримувати інфраструктуру агентів. Для великих підприємств із тисячами вузлів Puppet або Chef забезпечують кращу масштабованість та більш розвинені механізми звітності. Системи CM не є взаємовиключними: наприклад, Ansible може використовуватись для початкового розгортання та встановлення агентів Puppet, тоді як сам Puppet забезпечує подальше постійне підтримання конфігурації. Поширеним патерном у великих організаціях є також використання Terraform для управління інфраструктурою (provision), Ansible — для конфігурування ОС та застосунків, та спеціалізованих систем для оркестрації контейнерів.

6.2. Принципи конфігураційного управління

Одним із центральних принципів конфігураційного управління є ідемпотентність — властивість операції, при якій її багаторазове виконання дає

той самий результат, що й одноразове. Математично це означає, що для функції `f` виконується умова `f(f(x)) = f(x)` для будь-якого `x`. Стосовно системного адміністрування ідемпотентна операція може бути безпечно запущена повторно без ризику небажаних побічних ефектів: наприклад, команда «встановити пакет nginx версії 1.24» є ідемпотентною, оскільки якщо nginx вже встановлений у потрібній версії, вона просто підтвердить цей факт без жодних змін у системі. Ця властивість є критично важливою для автоматизованого управління конфігурацією, оскільки скрипти та playbook запускаються регулярно або тригерно, і кожен запуск має приводити систему до коректного стану незалежно від її поточного становища.

Неідемпотентні операції можуть накопичувати побічні ефекти при повторному виконанні. Класичний приклад — команда «додати рядок до файлу» без перевірки на наявність цього рядка: при кожному повторному запуску в `/etc/hosts` або `/etc/fstab` з'являтимуться нові дублікати, що зрештою може призвести до некоректної поведінки системи або навіть неможливості її завантаження. Аналогічно, створення правила iptables без перевірки його відсутності при кожному перезапуску сервісу накопичуватиме дублікати, що сповільнюють обробку мережевого трафіку. При написанні скриптів та плейбуків адміністратори повинні свідомо забезпечувати ідемпотентність кожної операції — через перевірку поточного стану перед виконанням дії (`if ! grep -q "..."`, перевірку exit code), використання атомарних операцій або ідемпотентних за природою інструментів. Ідемпотентність також тісно пов'язана з концепцією «конвергентної конфігурації»: система CM з часом наближає реальний стан до бажаного та підтримує його, навіть якщо на шляху виникають часткові збої.

Версіонування конфігурацій є логічним продовженням ідеї Infrastructure as Code (IaC): якщо конфігурація інфраструктури описана у вигляді коду, цей код слід зберігати в системі версійного контролю так само, як і будь-який інший програмний код. Використання Git для зберігання конфігурацій дозволяє відстежувати всі зміни — хто, коли і чому вніс зміну, — а також повертатись до будь-якого попереднього стану у разі проблем. Це кардинально змінює підхід до управління IT-інфраструктурою: замість того щоб «заходити на сервер та щось налаштовувати», адміністратор вносить зміну до репозиторію, яка потім автоматично розгортається за встановленим процесом, що забезпечує аудитованість та контрольованість кожної модифікації.

GitOps — сучасна методологія, що розширює принципи DevOps шляхом використання Git-репозиторію як єдиного джерела істини для стану інфраструктури. В GitOps-підході будь-яка зміна конфігурації проходить через стандартний процес code review у вигляді pull request, що забезпечує контроль якості та відповідальність. Автоматизована система (наприклад, ArgoCD або Flux для Kubernetes, або CI/CD-пайплайн для традиційних систем) постійно порівнює фактичний стан інфраструктури з тим, що описано в репозиторії, та при виявленні розбіжностей або автоматично усуває їх, або сигналізує про конфігураційний дрейф. Ця модель значно покращує аудитованість змін і спрощує реагування на інциденти: весь history зберігається в одному місці,

кожна зміна супроводжується описом її причини та автором, а «аварійний відкат» зводиться до повернення репозиторію до попереднього коміту.

Terraform є одним із найбільш поширених інструментів IaC для управління хмарною та локальною інфраструктурою. На відміну від систем CM (Ansible, Puppet), що зосереджуються на конфігурації вже існуючих вузлів, Terraform дозволяє описувати саму інфраструктуру — мережі, сервери, бази даних, балансувальники навантаження — у декларативному форматі HCL (HashiCorp Configuration Language). Terraform підтримує десятки провайдерів (AWS, Azure, GCP, VMware, Kubernetes тощо) та реалізує транзакційну модель змін: перед застосуванням будь-яких дій він будує план змін (`terraform plan`) та запитує підтвердження, що дозволяє переглянути, що саме зміниться. Поєднання Terraform для розгортання інфраструктури та Ansible або Puppet для конфігурування програмного забезпечення є типовим стеком IaC у сучасних організаціях, де перший відповідає за «що запустити», а другий — за «як налаштувати».

На рисунку 6.1 зображено типовий процес GitOps: ланцюжок подій від внесення змін до їх застосування в інфраструктурі. Зліва направо розташовані блоки: «Розробник/Адміністратор» (пише конфігурацію та робить commit), «Git-репозиторій» (зберігає зміни, де виконується code review та merge до main-гілки), «CI-система» (виконує автоматичні тести конфігурації: перевірку синтаксису, лінтинг, тести відповідності базовій лінії безпеки, статус позначається на коміті), «GitOps-оператор» (ArgoCD/Flux, постійно порівнює стан репозиторію зі станом інфраструктури), «Інфраструктура» (сервери або Kubernetes-кластер, де застосовуються зміни). Стрілки позначають напрям: від розробника до репозиторію — «git push», від репозиторію до CI — «тригер», від CI назад до репозиторію — «статус перевірки», від GitOps-оператора до інфраструктури — «sync / apply». Знизу під схемою показано стрілку зворотного зв’язку від інфраструктури до GitOps-оператора із написом «Виявлення дрейфу → alert».

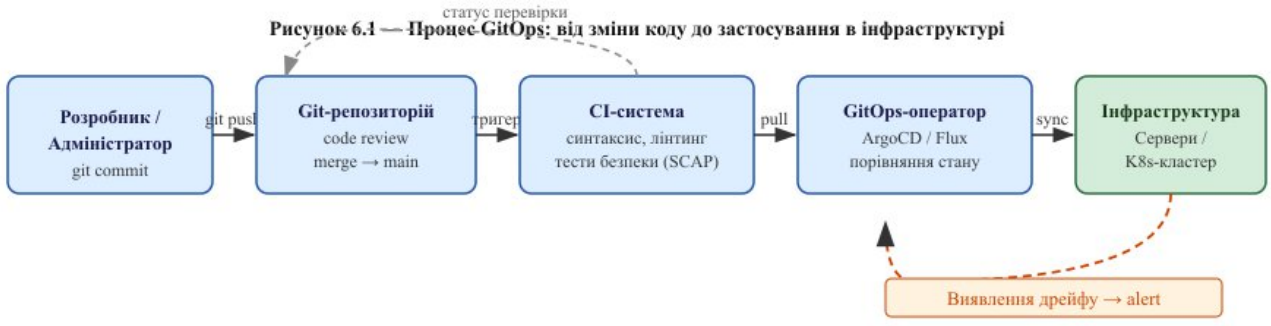

Рисунок 6.1 — Процес GitOps: від зміни коду до застосування в інфраструктурі

Транзакційність змін означає, що будь-яке оновлення конфігурації має відбуватись як атомарна операція: або всі компоненти зміни застосовуються успішно, або система залишається в попередньому стані. Ця концепція запозичена з теорії баз даних, де транзакція гарантує властивості ACID (Atomicity, Consistency, Isolation, Durability). В контексті ОС та системного

адміністрування транзакційність особливо важлива при оновленні взаємозалежних компонентів: якщо оновлення бібліотеки завершилось успішно, але оновлення залежного конфігураційного файлу зазнало невдачі, система може опинитись у непослідовному стані. Класичним прикладом є одночасне оновлення nginx і конфігурації TLS-сертифікатів: часткова зміна може зробити веб-сервер непрацездатним або вразливим, якщо нова версія програми вимагає параметрів, яких ще немає в конфігурації.

Деякі пакетні менеджери та системи управління конфігурацією реалізують транзакційні механізми. Менеджер пакетів RPM у поєднанні з dnf підтримує концепцію «транзакцій», що можна відкотити командою `dnf history undo <id>`. Nix і Guix — функціональні менеджери пакетів — реалізують повну транзакційність через незмінні «покоління» системи: кожна зміна конфігурації створює нове покоління, перемикання між якими є атомарним і оборотним навіть при оновленні ядра. NixOS дозволяє адміністратору описати повну конфігурацію системи в одному декларативному файлі `configuration.nix` та гарантує, що при відкочуванні буде повністю відновлено попередній стан — аж до тих самих версій пакетів та налаштувань сервісів. Цей підхід є особливо привабливим для середовищ із підвищеними вимогами до відтворюваності та надійності.

Конфігураційний дрейф (configuration drift) — явище, при якому реальний стан системи поступово відхиляється від задокументованого або бажаного стану внаслідок ручних змін, неповністю застосованих оновлень або непередбачуваних побічних ефектів. Дрейф є однією з головних причин труднощів при діагностиці інцидентів: якщо реальна конфігурація відрізняється від тієї, що задокументована, адміністратор витрачає значний час на з’ясування фактичного стану системи. Для боротьби з дрейфом застосовуються регулярні перевірки відповідності та автоматична корекція через системи CM — Puppet і SaltStack за замовчуванням перевіряють стан вузла кожні 30 хвилин і автоматично усувають відхилення. Деякі організації дотримуються принципу «незмінної інфраструктури» (immutable infrastructure): замість модифікації працюючого вузла його замінюють новим, побудованим з нуля за актуальними специфікаціями, що повністю виключає можливість накопичення дрейфу.

У таблиці 6.2 наведено порівняльну характеристику ключових принципів конфігураційного управління з прикладами їх реалізації та типовими порушеннями.

Таблиця 6.2 — Принципи конфігураційного управління та їх практична реалізація

Принцип	Визначення	Механізм реалізації	Приклад порушення
Ідемпотентність	Повторне виконання дає той самий результат	Перевірка стану перед дією, декларативні модулі CM	Дублювання рядків у /etc/fstab після повторного запуску скрипта
Версіонування	Зміни відслідковуються у	Git + GitOps-оператори (ArgoCD, Flux)	«Хто змінив конфіг nginx і коли?» —

Принцип	Визначення	Механізм реалізації	Приклад порушення
	VCS		невідомо
Транзакційність	Зміни атомарні, можливий відкат	dnf history, NixOS generations, Terraform plan	Оновлення бібліотеки без конфігу залежного сервісу
Мінімальні привілеї	CM-агент має лише необхідні права	Sudo-правила, service accounts із обмеженими правами	Запуск Ansible від root руйнує права файлів сервісу
Єдине джерело істини	Один репозиторій описує весь стан системи	IaC (Terraform + Ansible), монорепозиторій	Два адміністратори підтримують різні копії конфігурації
Документованість	Причина кожної зміни зафіксована	Commit messages, PR descriptions, ITSM-тікети	Незрозуміле правило в iptables без коментаря чи тікета

6.3. Планування завдань в операційних системах

Одним із найстаріших і найуніверсальніших механізмів планування завдань у Unix-подібних операційних системах є демон cron. Він з'явився ще в ранніх версіях Unix і в майже незмінному вигляді присутній у всіх сучасних Linux-дистрибутивах та macOS. Cron читає конфігурацію з так званих «crontab»-файлів, де кожен рядок описує одне завдання у форматі «розклад команда». Формат розкладу складається з п'яти полів: хвилини (0–59), години (0–23), дня місяця (1–31), місяця (1–12) та дня тижня (0–7, де 0 і 7 означають неділю). Символ `*` у будь-якому полі означає «будь-яке значення», дріб `/` задає крок повторення, а кома дозволяє перерахувати декілька конкретних значень. Наприклад, рядок `0 3 * * 1 /usr/local/bin/backup.sh` означає запуск скрипта резервного копіювання щопонеділка о 3:00 ранку, тоді як `*/15 * * * * /usr/local/bin/health_check.sh` запускатиме перевірку стану системи кожні 15 хвилин.

Кожен користувач системи має власний crontab, доступний через команду `crontab -e`, а системний адміністратор може також розміщувати скрипти у спеціальних директоріях `/etc/cron.daily/`, `/etc/cron.weekly/`, `/etc/cron.monthly/` для автоматичного запуску відповідно щоденних, щотижневих та щомісячних завдань. Системний crontab (`/etc/crontab`) містить додаткове поле, що визначає, від імені якого користувача запускається команда — це важливо з точки зору безпеки, адже завдання слід запускати від імені користувача з мінімально необхідними правами, уникаючи виконання від root без крайньої потреби. Вивід cron-завдань за замовчуванням надсилається на електронну пошту локального користувача, але на практиці більшість скриптів перенаправляють стандартний вивід та вивід помилок у лог-файли за допомогою конструкції `>> /var/log/task.log 2>&1`.

Anacron вирішує проблему пропущених завдань на системах, що не працюють цілодобово. Якщо завдання не було виконане у запланований час через вимкнення комп'ютера, anacron виконає його при наступному запуску

системи з гарантованою мінімальною затримкою, але не частіше, ніж задано в конфігурації. Це особливо важливо для щоденних задач обслуговування на настільних системах та ноутбуках, де машина може бути вимкнена в ніч, коли заплановано щотижневе резервне копіювання або оновлення баз даних антивірусу. Конфігурація anacron у файлі `/etc/anacrontab` задається трьома полями: інтервалом у днях, затримкою запуску після старту системи в хвилинах та назвою завдання. Завдяки записам у файлі-мітці останнього виконання anacron завжди знає, чи необхідно запускати завдання, або воно вже виконувалось у відповідний день.

Systemd-timer — сучасна альтернатива cron у системах з systemd — описує розклад у форматі unit-файлів і поєднує потужність планування із загальною інфраструктурою керування сервісами. Директива `OnCalendar=` дозволяє задавати розклад у зрозумілому форматі (наприклад, `Mon *-*-* 03:00:00` для щопонеділкового запуску о 3:00), тоді як `OnBootSec=` і `OnActiveSec=` задають відносні затримки. Директива `Persistent=true` реалізує функціональність anacron: завдання виконується при наступному старті системи, якщо воно пропустило свій запланований час. Рандомізована затримка запуску (`RandomizedDelaySec`), яку підтримує systemd-timer, є корисною при великій кількості вузлів у кластері — розсіювання запусків у часі запобігає одночасному навантаженню на загальні ресурси (бази даних, NFS, репозиторії оновлень). Перегляд стану всіх активних таймерів командою `systemctl list-timers` і деталізоване журналювання через `journalctl` суттєво спрощують діагностику проблем порівняно з класичним cron.

На рисунку 6.2 схематично зображено архітектуру та взаємодію компонентів планування завдань у Linux-системі. Рисунок розділено на дві вертикальні колонки. Ліва колонка «Cron» зображує файли `/etc/cron.d/`, `~/.config/cron/tabs/` та `/etc/crontab`, що стрілками вказують на блок «Cron-демон (crond)», який стрілкою «fork + exec» веде до блоку «Дочірній процес (bash/python/…)», а той — до блоку «Вивід → /var/log або email». Права колонка «Systemd.timer» зображує файли `*.timer` та `*.service` у `/etc/systemd/system/`, що вказують на блок «systemd (PID 1)», який активує «Сервісний unit», що запускає «Дочірній процес», чий вивід стрілкою «stdout/stderr» прямує до блоку «journald». Між колонками розміщено блок «Адміністратор», від якого стрілки ведуть до обох систем із відповідними командами (`crontab -e` та `systemctl edit`). Знизу обидві колонки з’єднані блоком «Моніторинг» з підписом «перевірка виконання задач».

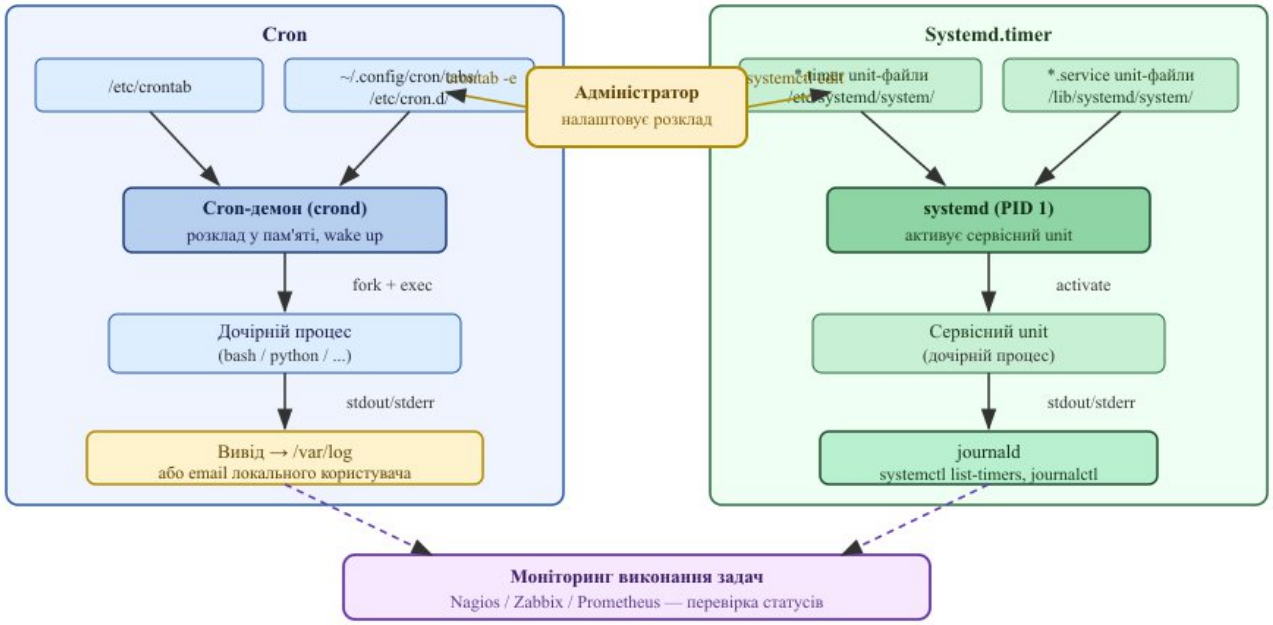

Рисунок 6.2 — Архітектура планування завдань у Linux: порівняння cron та systemd.timer

Windows Task Scheduler (Планувальник завдань) є еквівалентом cron у Windows і надає значно розвинутіший набір можливостей для управління розкладом. Задача (Task) складається з трьох основних компонентів: тригерів (triggers), що визначають умови запуску; дій (actions), що описують виконувані операції; та умов (conditions) і параметрів (settings), що уточнюють поведінку задачі. Тригери можуть бути часовими (за фіксованим розкладом з можливістю повторення всередині вікна часу), подієвими (при вході в систему, запуску Windows, певній події у журналі подій — наприклад, при виявленні конкретного Event ID у Security Log) або комбінованими, що об'єднують кілька умов. Тригер типу «При вході в систему» часто використовується для задач, що налаштовують робоче середовище конкретного користувача, тоді як тригер «При запуску системи» підходить для сервісних задач, що мають виконуватись до входу будь-якого користувача.

Адміністрування Task Scheduler можливе через утиліту командного рядка `schtasks.exe` або через PowerShell-командлети `New-ScheduledTask`, `Register-ScheduledTask`, `Get-ScheduledTask`, `Enable-ScheduledTask` та інші. PowerShell-підхід особливо потужний для масового створення та управління задачами на багатьох комп'ютерах через групові політики (Group Policy Object, GPO) або системи CM. Завдання Task Scheduler зберігаються у форматі XML у директорії `C:\Windows\System32\Tasks\`, що дозволяє їх версіонування та розгортання через стандартні механізми управління файлами або GPO «Immediate Tasks». Права на запуск задач контролюються стандартними механізмами безпеки Windows: задача виконується в контексті зазначеного облікового запису, і рекомендується використовувати спеціальні сервісні акаунти з мінімально необхідними правами замість облікових записів адміністраторів.

Параметр «Run whether user is logged on or not» дозволяє задачі виконуватись у фоновому режимі незалежно від активної сесії, проте вимагає

збереження пароля облікового запису в диспетчері задач. Альтернативою є використання облікового запису типу «SYSTEM» або «Network Service» для задач, що не потребують доступу від імені конкретного користувача. Параметр «Run with highest privileges» надає задачі права UAC-адміністратора — цей параметр слід використовувати лише за необхідності, дотримуючись принципу найменших привілеїв. Налаштування «If the task fails, restart every X minutes» забезпечує стійкість критичних задач, а «Stop the task if it runs longer than X hours» захищає від задач, що «зависли» та блокують ресурси системи.

На рисунку 6.3 зображено структуру задачі Windows Task Scheduler у вигляді ієрархічної схеми.

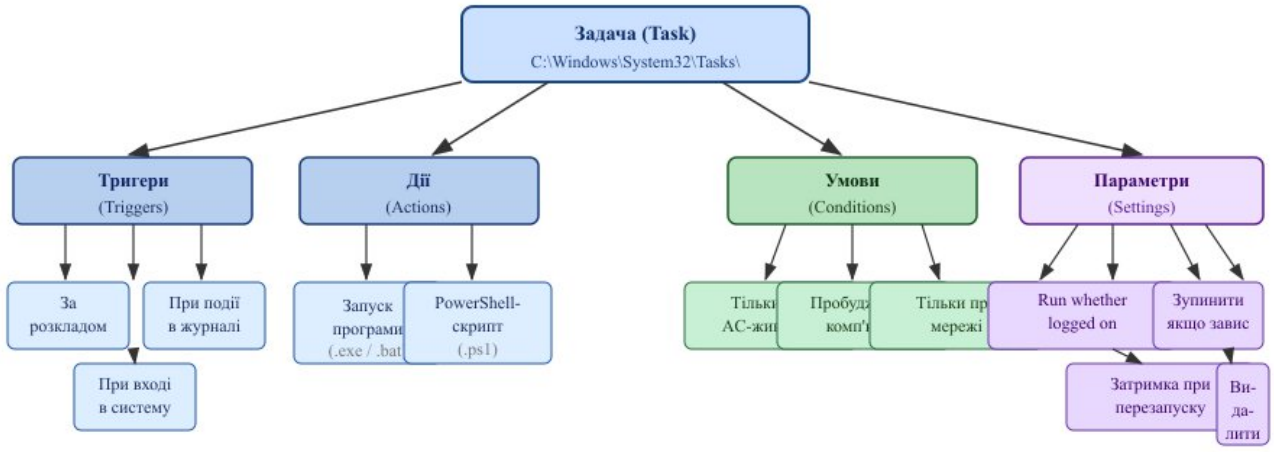

Рисунок 6.3 — Структура задачі Windows Task Scheduler

У таблиці 6.3 наведено порівняльну характеристику основних механізмів планування завдань у різних операційних системах, що допоможе вибрати відповідний інструмент для конкретного сценарію.

Таблиця 6.3 — Порівняльна характеристика механізмів планування завдань

Характеристика	cron (Linux)	systemd.timer	Windows Task Scheduler	Anacron
Тип тригерів	Часовий	Часовий + події systemd	Часовий + системні події Windows	Часовий (з відкладенням)
Інтерфейс	Текстовий файл	Unit-файл (INI-подібний)	GUI + XML + CLI (schtasks)	Текстовий файл
Журналювання	syslog / /var/log	journald	Журнал подій Windows	syslog
Пропущені задачі	Ігноруються	Persistent=true — виконуються	Налаштовується (Run ASAP)	Виконуються при наступному старті
Залежності між задачами	Відсутні	Підтримуються (After=, Requires=)	Умови (Conditions)	Відсутні

Характеристика	cron (Linux)	systemd.timer	Windows Task Scheduler	Anacron
Підтримка ОС	Unix / Linux / macOS	Linux (systemd)	Windows	Unix / Linux
Безпека (привілеї)	Від імені конкретного user	Налаштовується (User=)	Service account / AD-акаунт	Виконується від root
Виконання при вимкненому ПК	Ні	З Persistent=true — так	Так (при наступному старті)	Так

Планувальники завдань знаходять застосування в широкому колі сценаріїв системного адміністрування. Одним із найпоширеніших є автоматичне резервне копіювання: cron-завдання може щоночі запускати скрипт, що архівує важливі директорії командою `tar`, передає архів на віддалений сервер через `rsync` або `rclone`, перевіряє цілісність архіву та надсилає звіт про успішність операції. Ротація та очищення старих логів є ще одним типовим завданням: інструмент `logrotate`, що зазвичай запускається через cron щоденно, стискає та архівує файли журналів, видаляє старі архіви та опціонально надсилає лог-файл до централізованої системи збору подій.

Моніторинг доступності сервісів через заплановані перевірки є важливою складовою проактивного адміністрування. Скрипт, що кожні 5 хвилин перевіряє відповідь веб-сервера, коректність сертифіката TLS, наповненість диску та навантаження ЦПУ, а при виявленні аномалій надсилає сповіщення через email або Slack, є простим прикладом такої автоматизації. Більш складні сценарії включають автоматичне перезапускання зупинених сервісів (`systemctl start` або `net start` у Windows), очищення тимчасових файлів і кешів, що накопичуються з часом, та генерацію регулярних звітів про стан системи. Взаємодія між плануванням завдань та системами моніторингу (Nagios, Zabbix, Prometheus) утворює повноцінний цикл автоматизованого управління: моніторинг виявляє проблему, заплановане завдання усуває її або ескалює до чергового адміністратора.

6.4. Інтеграція автоматизації з політиками безпеки

Своєчасне оновлення програмного забезпечення є одним із найефективніших заходів з кібербезпеки: значна частина успішних атак використовує вразливості, для яких на момент атаки вже існували виправлення, але вони не були своєчасно застосовані. Автоматизація процесу оновлення дозволяє скоротити «вікно вразливості» — проміжок часу між публікацією виправлення та його застосуванням — від тижнів до годин або навіть хвилин. В Ubuntu/Debian автоматичні оновлення реалізуються через пакет `unattended-upgrades`, який конфігурується у файлі `/etc/apt/apt.conf.d/50unattended-upgrades` та дозволяє гнучко вказати, які репозиторії та категорії оновлень застосовуються автоматично, а які потребують ручного затвердження. В

RHEL/CentOS аналогічну функцію виконує `dnf-automatic` із конфігурацією у `/etc/dnf/automatic.conf`, де окремо задаються параметри завантаження, встановлення та відправки сповіщень електронною поштою.

Однак повністю автоматичне оновлення всього програмного забезпечення на виробничих системах пов'язане з ризиками: нові версії пакетів можуть вносити несумісні зміни або нові помилки, що призведуть до збоїв у роботі сервісів. Тому на практиці організації застосовують диференційований підхід: оновлення безпеки застосовуються автоматично та максимально швидко, тоді як мажорні оновлення проходять попереднє тестування в непродуктивному середовищі, затвердження Change Advisory Board та планове розгортання з вікном обслуговування. Для тестування оновлень широко використовується підхід «кільця розгортання» (deployment rings): спочатку оновлення застосовується до невеликої групи менш критичних систем, збирається статистика та відгуки, і лише після підтвердження стабільності оновлення поширюється на решту парку машин.

Windows Server Update Services (WSUS) та його сучасний хмарний аналог Windows Update for Business дозволяють централізовано управляти оновленнями Windows-середовища. Адміністратор може затримати розгортання оновлень на певний час для збору інформації про можливі проблеми від «ранніх адоптерів», визначити кільця (rings) розгортання та повністю виключити певні оновлення. Інтеграція з Active Directory та груповими політиками дозволяє застосовувати різні стратегії до різних організаційних підрозділів: тестові машини можуть отримувати оновлення відразу після їх виходу, тоді як критична серверна інфраструктура — лише після двотижневого карантину та успішного тестування. Microsoft Endpoint Manager (Intune) розширює ці можливості на хмарні та гібридні середовища, дозволяючи централізовано управляти оновленнями навіть для пристроїв, що не підключені до корпоративної мережі.

Сканування вразливостей (vulnerability scanning) — це процес систематичної перевірки систем на наявність відомих вразливостей у програмному забезпеченні, неправильних налаштувань та відхилень від встановлених базових ліній безпеки. Автоматизація цього процесу через заплановані задачі дозволяє забезпечити безперервний моніторинг стану безпеки замість разових перевірок. OpenVAS (Open Vulnerability Assessment System) та його комерційний нащадок Greenbone Vulnerability Manager є одними з найпоширеніших відкритих рішень для сканування вразливостей. Комерційні рішення, такі як Nessus, Qualys та Rapid7 InsightVM, пропонують більш широкі бази даних вразливостей, кращу інтеграцію з тікет-системами та розширені засоби звітності й пріоритизації.

Сканування вразливостей слід інтегрувати в загальний процес управління вразливостями, що включає не лише виявлення, а й пріоритизацію, усунення та верифікацію виправлень. Пріоритизація базується на оцінці критичності вразливості (Common Vulnerability Scoring System, CVSS), наявності публічних експлойтів та важливості ураженого активу для бізнесу. Автоматизація дозволяє регулярно перераховувати пріоритети з урахуванням нових даних та

автоматично призначати завдання з усунення відповідальним командам через інтеграцію з системами відстеження задач (JIRA, ServiceNow тощо). Вразливість із оцінкою CVSS 9.0+, для якої існує публічний експлойт і яка виявлена на інтернет-доступному сервері, потребує негайного реагування — а автоматизована система може надіслати критичне сповіщення відповідальній команді ще до того, як черговий адміністратор прочитає ранковий звіт.

Аудит системних подій — це записування та аналіз значущих подій в операційній системі з метою виявлення підозрілої активності, розслідування інцидентів та підтвердження відповідності нормативним вимогам. В Linux основним інструментом аудиту є підсистема `auditd`, що дозволяє відстежувати системні виклики, зміни файлів, спроби входу в систему та зміни привілеїв. Правила аудиту конфігуруються у файлі `/etc/audit/audit.rules` або через утиліту `auditctl`, а журнали зберігаються у `/var/log/audit/audit.log` у форматі, придатному для парсингу та аналізу. Утиліта `ausearch` дозволяє швидко знаходити події за різними критеріями (час, користувач, тип події, файл), а `aureport` генерує статистичні звіти для регулярного аналізу.

У Windows аудит конфігурується через групові політики (Advanced Audit Policy Configuration) та записується до журналу подій Security у форматі Event Log. Стандарти безпеки, такі як PCI DSS, HIPAA та ISO 27001, висувають конкретні вимоги до того, які події мають журналюватись, як довго зберігатись журнали та як забезпечуватись їхня цілісність. Зокрема, PCI DSS вимагає зберігання аудиторських журналів протягом не менше одного року з можливістю негайного доступу до журналів за останні три місяці. SIEM-системи (Security Information and Event Management), такі як Splunk, IBM QRadar або відкрите рішення Elastic SIEM, агрегують журнали з багатьох джерел, нормалізують їх та застосовують правила кореляції для автоматичного виявлення підозрілих патернів. Автоматизований аналіз журналів значно перевершує можливості ручного перегляду: SIEM може обробляти мільярди подій на добу та сигналізувати про аномалії у режимі, близькому до реального часу.

Інтеграція SIEM з системами автоматичного реагування на інциденти (SOAR — Security Orchestration, Automation and Response) дозволяє не лише виявляти загрози, але й автоматично реагувати на них. Наприклад, при виявленні множинних невдалих спроб входу з однієї IP-адреси SOAR-система може автоматично заблокувати цю адресу у брандмауері, занести її до чорного списку в IPS та одночасно відкрити інцидент у тікет-системі з усіма деталями. При виявленні ознак компрометації облікового запису система може вимагати повторної автентифікації або тимчасово заблокувати обліковий запис до розслідування. Такий підхід скорочує час реагування з годин до секунд, що критично важливо при боротьбі з APT-атаками (Advanced Persistent Threat), де швидкість реагування безпосередньо визначає масштаб можливих наслідків.

На рисунку 6.4 зображено цикл управління вразливостями у вигляді кругової діаграми з чотирма фазами. Перша фаза — «Виявлення» (Discovery) — зображена у верхньому секторі: до неї входять блоки «Сканування вразливостей (OpenVAS/Nessus)» та «Сканування конфігурацій (OpenSCAP)»,

що виконуються автоматично за розкладом. Друга фаза — «Пріоритизація» — містить блоки «CVSS-оцінка», «Наявність експлойту» та «Критичність активу». Третя фаза — «Усунення» — містить блоки «Автоматичне оновлення (критичні)», «Ручне усунення (з тестуванням)» та «Призначення задачі у JIRA/ServiceNow». Четверта фаза — «Верифікація» — містить «Повторне сканування» та «Закриття тікета». Фази з'єднані стрілками по колу, а всередині кола розміщено блок «Неперервний моніторинг (SIEM)» зі стрілками до всіх чотирьох фаз.

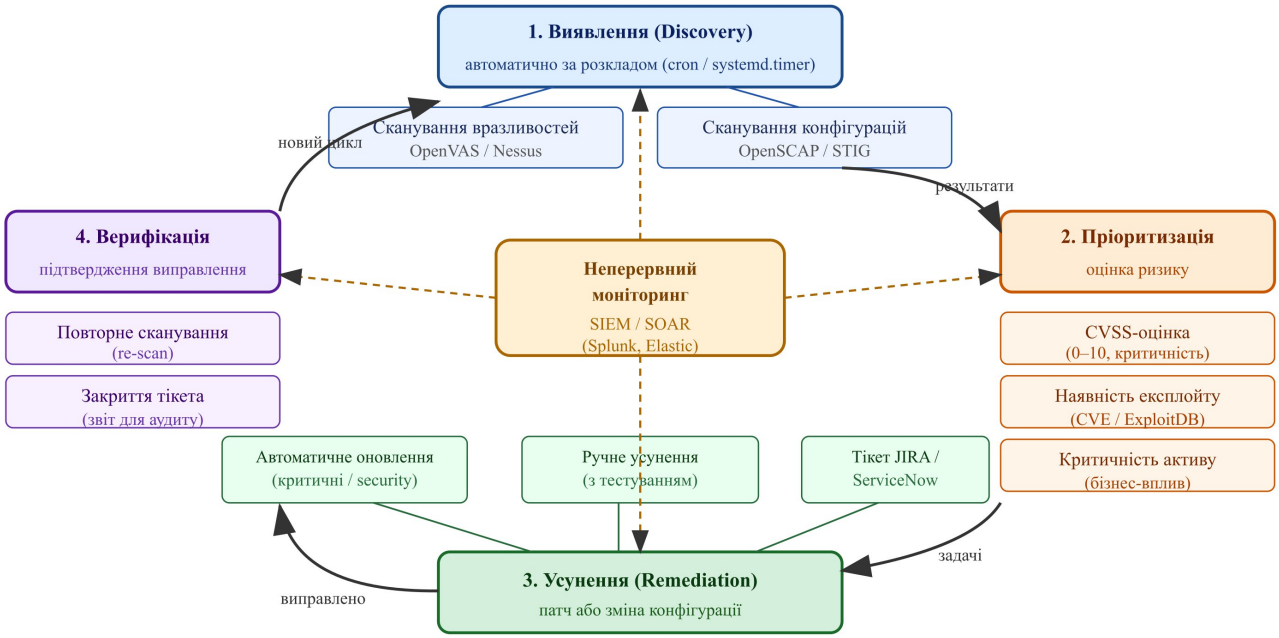

Рисунок 6.4 — Цикл управління вразливостями з інтеграцією автоматизації

Автоматизоване звітування є невід'ємною частиною як операційного управління IT-інфраструктурою, так і забезпечення відповідності нормативним вимогам. Регуляторні стандарти у сфері інформаційної безпеки — GDPR, PCI DSS, HIPAA, SOC 2, ISO/IEC 27001 — вимагають від організацій регулярного документування стану безпеки, проведення внутрішніх аудитів та надання доказів відповідності зовнішнім перевіряючим органам. Ручне збирання такої інформації є трудомістким, схильним до помилок та не забезпечує актуальність даних між аудитами. Автоматизовані інструменти відповідності, такі як OpenSCAP (Security Content Automation Protocol) для Linux або Microsoft Security Compliance Toolkit для Windows, дозволяють перевіряти конфігурацію системи відносно заздалегідь визначених базових ліній безпеки та автоматично генерувати звіти у стандартних форматах.

OpenSCAP реалізує стандарти SCAP (Security Content Automation Protocol), розроблені NIST, та підтримує профілі безпеки для різних стандартів і регуляторних вимог. Запускаючи `oscap xccdf eval` з відповідним профілем (наприклад, PCI-DSS або DISA STIG), адміністратор отримує детальний HTML-звіт про відповідність кожного правила з поясненням, чому воно не виконується, та рекомендаціями щодо усунення. Інтеграція OpenSCAP у pipeline CI/CD дозволяє перевіряти образи ОС на відповідність базовій лінії

безпеки ще до їх розгортання у виробниче середовище, що реалізує принцип «security as code» — безпека перевіряється автоматично і є частиною процесу доставки програмного забезпечення. Результати перевірок можна зберігати в часі та аналізувати тренди: якщо кількість невідповідностей зростає, це може свідчити про конфігураційний дрейф або нові вимоги, що з'явились у стандарті.

Автоматизоване звітування також охоплює операційні показники: доступність сервісів (uptime), завантаженість ресурсів, статистику оновлень, кількість відкритих вразливостей по кожному рівню критичності. Системи моніторингу, такі як Nagios, Zabbix, Prometheus у поєднанні з Grafana, дозволяють збирати ці дані та відображати їх у вигляді інтерактивних дашбордів та регулярних PDF-звітів, що автоматично розсилаються відповідальним особам. Деякі організації впроваджують концепцію «security dashboards» — панелей управління, що у реальному часі відображають ключові показники безпеки (кількість незакритих вразливостей CVSS $\geq$ 7, відсоток систем з актуальними оновленнями, кількість failed login подій за останні 24 години тощо) та є доступними для керівництва для прийняття управлінських рішень.

У таблиці 6.4 наведено огляд основних нормативних стандартів у сфері інформаційної безпеки та вимог, які вони висувають до автоматизованого аудиту і журналювання.

Таблиця 6.4 — Вимоги нормативних стандартів до аудиту та журналювання в ОС

Стандарт	Галузь	Вимоги до журналів	Мінімальний термін зберігання	Вимоги до цілісності
PCI DSS	Платіжні системи	Вхід, зміни конфігурації, адмін-дії, доступ до даних власників карт	1 рік (3 міс. — оперативно)	Захист від модифікації, синхронізація часу (NTP)
HIPAA	Охорона здоров'я (США)	Доступ до захищеної медичної інформації (PHI)	6 років	Тільки авторизований доступ до журналів
ISO/IEC 27001	Будь-яка	Відповідно до оцінки ризиків	Визначається організацією	Захист від несанкціонованої зміни
GDPR	ЄС (обробники персональних даних)	Доступ до персональних даних, зміни, витоки	Не менше ніж термін обробки	Забезпечення можливості розслідування порушень
SOC 2	SaaS-компанії	Відповідно до Trust Services Criteria	Визначається сферою аудиту	Незалежна перевірка (аудитори)

Ескалація сповіщень — механізм, при якому нереагована аномалія з часом сповіщає все вищих рівнів ієрархії відповідальних осіб — є важливим елементом автоматизованого управління інцидентами. Якщо черговий

адміністратор не підтвердив отримання критичного сповіщення протягом 15 хвилин, система автоматично надсилає повторне сповіщення та ескалює його до менеджера команди. Таким чином, автоматизація системних завдань та управління конфігурацією утворюють цілісну екосистему, що забезпечує не лише стабільну роботу ОС, але й відповідність вимогам безпеки та регуляторним стандартам — без залежності від людського фактора у рутинних, але критично важливих завданнях.

Тема 7. Віртуалізація, контейнеризація та управління сучасними IT-середовищами

7.1. Принципи віртуалізації та класифікація гіпервізорів

Віртуалізація є однією з фундаментальних технологій сучасної інформаційної інфраструктури, що дозволяє абстрагувати апаратні ресурси від програмного забезпечення, яке їх використовує. Ідея не є новою: перші концепції віртуалізації з'явилися ще у 1960-х роках у мейнфреймах IBM серії System/360, де виникла потреба одночасно обслуговувати кількох користувачів, надаючи кожному ілюзію виключного доступу до машини. Проте справжній розквіт технології у контексті серверних і клієнтських систем розпочався на початку 2000-х, коли компанії VMware, Microsoft та Xen Project запропонували комерційні та відкриті рішення для платформи x86. Сьогодні без віртуалізації неможливо уявити хмарні обчислення, центри обробки даних, середовища розробки та тестування, а також системи аварійного відновлення. Показово, що понад 90% навантажень у хмарних середовищах Amazon AWS, Microsoft Azure та Google Cloud виконується у тій чи іншій формі ізольованих обчислювальних середовищ — від класичних VM до спеціалізованих мікровіртуальних машин.

В основі віртуалізації лежить спеціалізований програмний або програмно-апаратний компонент — гіпервізор (hypervisor), що також відомий як монітор віртуальних машин (VMM, Virtual Machine Monitor). Гіпервізор відповідає за створення та управління ізольованими обчислювальними середовищами, кожне з яких отримує власний набір віртуальних ресурсів: процесор, оперативну пам'ять, мережеві інтерфейси, дискові пристрої. Фізична машина, на якій працює гіпервізор, називається хостом (host), а кожне із запущених у ньому ізольованих середовищ — гостьовою системою (guest) або віртуальною машиною (VM, Virtual Machine). Важливо розуміти, що гостьова операційна система в загальному випадку не знає, що виконується у віртуальному середовищі — вона взаємодіє з абстракцією апаратного забезпечення, яку надає гіпервізор, і поводиться так, наче має справу зі справжнім «залізом».

Класифікація гіпервізорів будується передусім за рівнем, на якому вони функціонують відносно апаратного забезпечення та операційної системи хоста. Гіпервізори першого типу (Type 1, bare-metal hypervisor) встановлюються безпосередньо на фізичне обладнання та самі виконують функції операційної системи для керування апаратними ресурсами. До цієї категорії належать VMware ESXi, Microsoft Hyper-V (у серверному виконанні), Citrix XenServer та KVM (Kernel-based Virtual Machine), який є модулем ядра Linux. Гіпервізори другого типу (Type 2, hosted hypervisor) функціонують поверх існуючої операційної системи хоста і залежать від неї для отримання доступу до апаратних ресурсів. Представниками цього типу є VMware Workstation, Oracle VirtualBox та QEMU у класичному режимі роботи без KVM. Різниця між типами суттєво впливає на продуктивність, накладні витрати та сфери застосування: Type 1 домінує в ЦОД і хмарних платформах завдяки

мінімальним накладним витратам та незалежності від стороннього ПЗ, тоді як Type 2 залишається незамінним для локальних середовищ розробника та навчальних цілей.

У таблиці 7.1 наведено порівняльну характеристику гіпервізорів першого та другого типів за ключовими параметрами.

Таблиця 7.1 — Порівняння гіпервізорів Type 1 та Type 2

Параметр	Type 1 (Bare-metal)	Type 2 (Hosted)
Рівень встановлення	Безпосередньо на апаратуру	Поверх ОС хоста
Продуктивність	Висока (мінімальні накладні витрати)	Нижча (додатковий рівень ОС хоста)
Ізоляція	Повна	Залежить від ОС хоста
Типові представники	VMware ESXi, Hyper-V, KVM	VirtualBox, VMware Workstation
Область застосування	Серверна інфраструктура, ЦОД	Розробка, тестування, навчання
Складність адміністрування	Висока	Відносно низька
Захищеність	Висока	Залежить від рівня захисту хоста
Вимоги до апаратних ресурсів	Залежать від навантаження VM	Хост ОС + навантаження VM

Рисунок 7.1 ілюструє архітектурну схему гіпервізорів обох типів. На схемі зображено два рядково розташованих блоки.

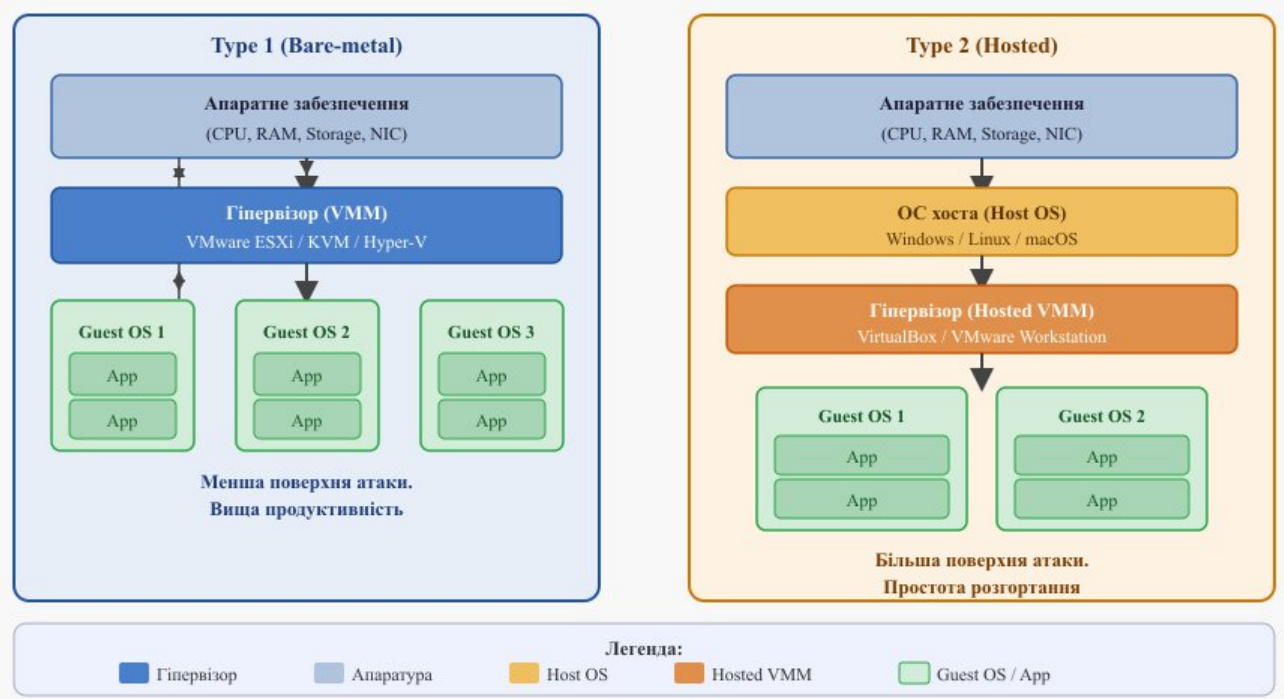

Рисунок 7.1 — Архітектурне порівняння гіпервізорів Type 1 та Type 2

Апаратна підтримка віртуалізації є ключовим чинником продуктивності сучасних гіпервізорів. Технології Intel VT-x (Virtualization Technology for IA-32, IA-64, and Intel 64) та AMD-V (AMD Virtualization) розширюють архітектуру процесора, додаючи новий привілейований режим роботи — VMX root mode для гіпервізора та VMX non-root mode для гостьових систем. Це дозволяє гостьовому ядру виконувати привілейовані інструкції безпосередньо на процесорі без дорогої програмної емуляції або бінарної трансляції. Додатково технології Intel VT-d та AMD-Vi забезпечують апаратну підтримку прямого призначення пристроїв гостьовим системам (passthrough), а також апаратну ізоляцію DMA-транзакцій через IOMMU (Input-Output Memory Management Unit), що значно підвищує як продуктивність, так і безпеку. Зокрема, GPU passthrough дозволяє надати гостьовій VM прямий доступ до дискретної відеокарти без втрати продуктивності, що є критичним для робочих станцій з вимогливими графічними додатками або для задач машинного навчання у VM.

Паравіртуалізація є окремим підходом, при якому гостьова операційна система модифікується для безпосередньої взаємодії з гіпервізором через спеціальний інтерфейс — гіпервиклики (hypercalls). Цей підхід, реалізований у Xen Project, дозволяє уникнути накладних витрат на перехоплення привілейованих інструкцій, оскільки гостьова система свідомо делегує їх виконання гіпервізору. Недолік паравіртуалізації — необхідність модифікації ядра гостьової ОС, що обмежує сумісність і ускладнює підтримку немодифікованих (особливо пропрієтарних) систем. Більшість сучасних рішень використовують гібридний підхід: апаратна підтримка забезпечує базову ізоляцію та перехоплення критичних операцій, а паравіртуальні драйвери — virtio у KVM, PVSCSI та VMXNET у VMware — прискорюють операції введення-виведення, надаючи гостьовій ОС більш ефективний шлях до фізичних пристроїв порівняно з повною апаратною емуляцією. Завдяки virtio-драйверам мережева продуктивність VM на KVM практично не поступається «голому залізу», а затримки при операціях з дисковою підсистемою знижуються у рази.

Консолідація серверів — одне з найважливіших економічних обґрунтувань впровадження гіпервізорної віртуалізації у корпоративних ЦОД. До появи масштабованої x86-віртуалізації більшість серверних застосунків розгорталися на виділених фізичних машинах через несумісність програмного стеку, вимоги до ізоляції або проблеми з надійністю. Середній рівень завантаження фізичного сервера при цьому рідко перевищував 10–15%, що вело до колосального марнування капітальних витрат і витрат на споживання електроенергії та охолодження. Консолідація кількох десятків фізичних серверів на одному потужному хості з гіпервізором дозволяє підвищити рівень завантаження до 60–80%, скоротити займану площу та споживання енергії у ЦОД у рази, а також суттєво спростити процедури резервного копіювання та управління конфігурацією. Слід, однак, враховувати ризик «шумного сусіда» (noisy neighbor): VM з нерівномірним навантаженням може негативно впливати на інші VM через конкуренцію за спільні ресурси — зокрема кеш процесора та пропускну спроможність шини пам’яті — тому гіпервізори надають механізми

обмеження та резервування ресурсів (resource pools, reservations, limits) для гарантування якості обслуговування.

7.2. Контейнеризація як альтернатива повній віртуалізації

Контейнеризація являє собою принципово інший підхід до ізоляції обчислювальних середовищ, який часто розглядають як легковагову альтернативу традиційній віртуалізації. На відміну від гіпервізорного підходу, де кожна гостьова система запускає власне ядро ОС та потребує повного набору системних бібліотек, контейнери спільно використовують ядро операційної системи хоста, ізолюючи лише простір імен процесів, файлову систему, мережевий стек та ресурсні ліміти. Така архітектура радикально знижує накладні витрати: контейнер стартує за секунди (а не хвилини, як повноцінна VM), споживає значно менше оперативної пам’яті та дискового простору, і дозволяє розмістити на одному фізичному сервері набагато більше ізольованих середовищ. На практиці сервер, що здатен запустити 10–15 VM, може одночасно обслуговувати сотні або навіть тисячі контейнерів зі схожим сумарним навантаженням, що зумовлює надзвичайно щільне використання апаратних ресурсів у хмарних платформах.

Технологічна основа контейнерів у Linux спирається на два ключові механізми ядра. Простори імен (namespaces) забезпечують ізоляцію різних ресурсів: pid-namespace ізолює дерево процесів (процеси в контейнері бачать лише свої pid, починаючи з 1), net-namespace виділяє окремий мережевий стек зі своїми інтерфейсами та таблицями маршрутизації, mnt-namespace ізолює точки монтування файлової системи, uts-namespace надає контейнеру власне ім’я хоста, ipc-namespace ізолює міжпроцесну взаємодію (черги повідомлень, семафори), а user-namespace дозволяє відображати UID/GID всередині контейнера на інші значення зовні. Control groups (cgroups) версії 2 забезпечують обмеження та облік споживання ресурсів: CPU-час (включаючи квоти та ваги між контейнерами), обсяг оперативної пам’яті з жорсткими та м’якими лімітами, смугу пропускання мережі та кількість операцій вводу-виводу. Разом ці два механізми утворюють ядро будь-якого контейнерного рушія та існували у ядрі Linux задовго до появи Docker, забезпечуючи технологічну основу для LXC (Linux Containers) — попередника сучасних контейнерних рішень.

Технологія Docker, запропонована у 2013 році компанією dotCloud, зробила контейнери доступними для широкого загалу, надавши зручний інструментарій для збирання образів, їх розповсюдження через реєстри та керування життєвим циклом контейнерів. Ключовою інновацією Docker став Union File System (OverlayFS у сучасних системах): образ контейнера складається з незмінних шарів (layers), що містять окремі фрагменти файлової системи, і лише верхній записуваний шар є унікальним для кожного запущеного контейнера. Завдяки цьому безліч контейнерів, що базуються на одному образі, можуть спільно використовувати незмінні шари на диску, суттєво знижуючи використання сховища. Файл Dockerfile описує

послідовність команд для збирання образу і стає артефактом «інфраструктури як коду» (Infrastructure as Code), що дозволяє версіонувати, перевіряти та аудитувати середовище виконання застосунку так само, як і його вихідний код.

Варто чітко розуміти відмінності між контейнерами і повноцінними VM, адже обидві технології вирішують схожі завдання ізоляції, але зовсім різними засобами. У таблиці 7.2 наведено детальне порівняння за ключовими характеристиками, яке допомагає обрати відповідний підхід для конкретного сценарію.

Таблиця 7.2 — Порівняння контейнерів та віртуальних машин

Характеристика	Контейнери	Віртуальні машини
Ізоляція ядра ОС	Спільне ядро хоста	Власне ядро у кожній VM
Час запуску	Секунди (< 1–5 с)	Хвилини (1–5 хв)
Накладні витрати пам’яті	Мінімальні (десятки МБ)	Значні (сотні МБ – ГБ)
Рівень ізоляції	Простори імен ядра	Апаратна / гіпервізорна
Портабельність	Висока (образ = артефакт)	Обмежена (великі файли VM)
Безпека	Потребує додаткового налаштування	Вища за замовчуванням
Підтримувана ОС гостя	Лише Linux (на Linux-хості)	Будь-яка ОС
Щільність розміщення	Сотні–тисячі на хості	Десятки на хості
Типові інструменти	Docker, Podman, containerd	VMware, KVM, Hyper-V
Формат опису середовища	Dockerfile, docker-compose.yml	OVF/OVA, Vagrantfile

Поряд з Docker у сучасній контейнерній екосистемі активно застосовуються й альтернативні рушії. Podman від Red Hat пропонує схожий з Docker CLI, але принципово відмінну архітектуру: відсутній центральний демон із підвищеними правами, а контейнери запускаються безпосередньо від імені поточного користувача (rootless containers), що суттєво зменшує поверхню атаки — компрометація рушія не дає зловмисникові привілеї root на хості. Containerd та CRI-O є низькорівневими рушіями, сумісними зі специфікацією OCI (Open Container Initiative), і призначені переважно для використання у складі Kubernetes, а не для безпосередньої роботи розробника. Sandbox-контейнери, такі як gVisor від Google та Kata Containers, поєднують переваги контейнерів та VM: кожен контейнер отримує власне мікроядро або легковагову VM, що забезпечує значно глибшу ізоляцію системних викликів від хостового ядра та наближає рівень безпеки контейнерів до гіпервізорної ізоляції.

Оркестрація контейнерів — управління їх розгортанням, масштабуванням та відновленням у розподілених кластерах — вирішується за допомогою спеціалізованих систем. Kubernetes (K8s), розроблений у Google на основі внутрішнього проєкту Borg та переданий до Cloud Native Computing Foundation у 2014 році, став де-факто стандартом у цій галузі. Він оперує абстракціями

вищого рівня, ніж окремий контейнер: Pod — мінімальна одиниця розгортання, що може містити кілька тісно пов’язаних контейнерів зі спільним мережевим простором імен і томами; Deployment — описує бажаний стан групи ідентичних Pod та управляє їх rolling-оновленням; Service — стабільний мережевий ендпоінт зі власною IP-адресою, що балансує трафік між Pod-ами; Namespace — логічний розділ кластера для організації ресурсів за командами або середовищами. Kubernetes безперервно звіряє поточний стан кластера з описаним у маніфестах (desired state), автоматично перезапускаючи збійні поди, розподіляючи навантаження між вузлами та виконуючи оновлення без простою, що суттєво підвищує операційну надійність порівняно з ручним управлінням сервісами. Рисунок 7.2 ілюструє ієрархію абстракцій Kubernetes.

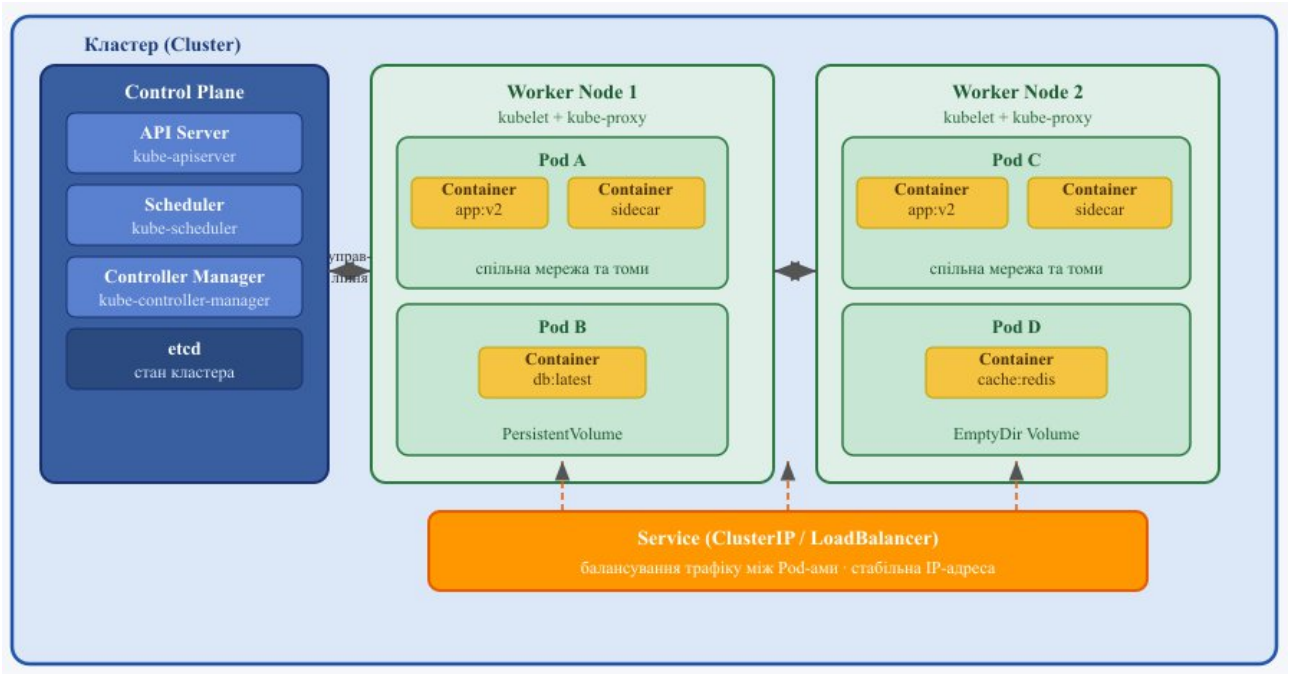

Рисунок 7.2 — Ієрархія абстракцій Kubernetes: кластер, вузли, Pod-и та Service

7.3. Безпека віртуальних середовищ та контейнерів

Безпека у контексті віртуалізації є багаторівневою проблемою, яка охоплює як специфічні загрози для гіпервізорного шару та контейнерного середовища, так і класичні загрози безпеці операційних систем, що продовжують діяти всередині ізольованих середовищ. Помилкове припущення, ніби «ізоляція = безпека», є однією з найнебезпечніших хибних концепцій у сфері DevOps та хмарних обчислень. Реальна захищеність середовища визначається глибиною ізоляції, якістю налаштувань безпеки та своєчасністю усунення вразливостей на всіх рівнях стеку: від апаратного мікрокоду до прикладного програмного забезпечення в контейнері. Модель спільної відповідальності (shared responsibility model), що використовується хмарними провайдерами, добре ілюструє цей принцип: провайдер відповідає за безпеку фізичної інфраструктури та гіпервізора, тоді як клієнт несе відповідальність за безпеку гостьових ОС, контейнерів та застосунків.

Найбільш критичною загрозою у контексті гіпервізорної віртуалізації є VM Escape (побіг з VM) — ситуація, коли зловмисний код, виконуючись всередині гостьової системи, використовує вразливість у гіпервізорі або у пристроях вводу-виводу, щоб вийти за межі VM та отримати доступ до хостової системи або інших VM. Вразливості у коді емуляції пристроїв є типовим вектором таких атак: вразливість VENOM (CVE-2015-3456) у контролері FDC (Floppy Disk Controller), реалізованому у QEMU, дозволяла виконати довільний код на хості з будь-якої VM на основі QEMU — що охопило KVM, Xen та інші рішення на базі QEMU. Атака Cloudburst (2009) продемонструвала можливість виходу з VM VMware через вразливість у відображенні 3D-графіки. Гіпервізори першого типу мають меншу поверхню атаки порівняно з Type 2, оскільки не несуть повного стеку операційної системи хоста — ESXi, наприклад, містить лише кілька мільйонів рядків коду порівняно з десятками мільйонів у Linux або Windows, що суттєво скорочує потенційну кількість вразливостей.

Атаки через побічні канали (side-channel attacks) являють собою клас загроз, специфічних для віртуалізованих середовищ, де різні VM або контейнери спільно використовують фізичні ресурси. Атаки Spectre та Meltdown, розкриті у 2018 році командами Google Project Zero та незалежними дослідниками, продемонстрували, що вразливості у мікроархітектурних оптимізаціях сучасних процесорів — спекулятивному виконанні та кеші — можуть бути використані для витягування довільних даних із пам’яті інших процесів або навіть ядра ОС. У хмарних середовищах, де на одному фізичному сервері можуть співіснувати VM різних клієнтів (multi-tenant), це набуває особливої критичності: теоретично один клієнт міг зчитувати дані іншого клієнта через спільний процесорний кеш. Для мітигації застосовуються патчі мікрокоду процесора, оновлення ядра ОС (KPTI — Kernel Page Table Isolation), відключення Hyper-Threading на критичних системах, а також механізм retpoline для запобігання атакам типу branch target injection. Технологія Confidential Computing — AMD SEV (Secure Encrypted Virtualization) та Intel TDX (Trust Domain Extensions) — є сучасною апаратною відповіддю на цей клас загроз: пам’ять кожної VM шифрується апаратно, і навіть компрометований гіпервізор не зможе прочитати вміст пам’яті гостьових систем.

У таблиці 7.3 наведено основні загрози безпеці та відповідні механізми захисту для різних типів віртуалізованих середовищ.

Таблиця 7.3 — Загрози та механізми захисту у віртуалізованих середовищах

Загроза	Вражений рівень	Механізм захисту
VM Escape	Гіпервізор / емуляція пристроїв	Оновлення гіпервізора, мінімізація емульованих пристроїв
Side-channel (Spectre/Meltdown)	Апаратний (CPU)	Патчі мікрокоду, KPTI, відключення HT
Container breakout	Ядро ОС / простори	Rootless containers, seccomp,

Загроза	Вражений рівень	Механізм захисту
	імен	AppArmor/SELinux
Ескалація привілеїв	Контейнер / VM	Принцип найменших привілеїв, USER директива у Dockerfile
Атаки на ланцюг поставок	Образи контейнерів	Перевірка підписів, сканування образів (Trivy, Clair)
DoS через ресурсне виснаження	Оркестратор / ядро	cgroups, LimitRange, ResourceQuota у Kubernetes
Мережева атака між VM/контейнерами	Мережевий рівень	Network Policies, мікросегментація, Service Mesh
Витік даних через пам’ять VM	Апаратний рівень	AMD SEV, Intel TDX (Confidential Computing)
Витік даних через образ контейнера	Шар образу	Мінімальні базові образи, distroless, SBOM

Ескалація привілеїв (privilege escalation) у контейнерних середовищах є ще одним суттєвим вектором атаки. Якщо контейнер запущено від імені root (uid 0) і система не налаштована належним чином, зловмисник, який отримав контроль над процесом всередині контейнера, потенційно може отримати привілеї root на хостовій системі — наприклад, через монтування директорії хоста або використання прапорця –privileged. Кращою практикою є запуск контейнерів від імені непривілейованого користувача (директиви USER у Dockerfile, параметри –user у Docker run), використання rootless-контейнерів (Podman), а також застосування seccomp-профілів для обмеження доступних системних викликів та AppArmor/SELinux-міток для мандатного контролю доступу. Важливо розуміти, що навіть якщо контейнер не запущений як root, деякі Linux capabilities — такі як CAP_NET_ADMIN або CAP_SYS_PTRACE — можуть надати достатні привілеї для зловмисних дій, тому мінімізація набору capabilities (параметри –cap-drop ALL –cap-add у Docker run) є обов’язковою частиною hardening контейнерного середовища.

Безпека ланцюга поставок (supply chain security) стала особливо актуальною проблемою з поширенням контейнерних технологій. Публічні реєстри образів, такі як Docker Hub, містять мільйони образів, серед яких чимало містять застарілі або вразливі компоненти, а деякі — навмисно впроваджений шкідливий код. Рекомендованою практикою є використання мінімалістичних базових образів (distroless від Google, Alpine Linux), регулярне сканування образів на вразливості за допомогою інструментів Trivy, Clair або Snyk, підписання образів за допомогою Notary/Cosign (проєкт Sigstore) та застосування політик допуску в Kubernetes (OPA Gatekeeper, Kyverno), що забороняють розгортання непідписаних або вразливих образів. Не менш важливою є перевірка Software Bill of Materials (SBOM) — декларації всіх компонентів, що входять до складу образу, яка дозволяє швидко визначити, чи зачіпає нова CVE вашу інфраструктуру, і автоматизувати відповідь на інцидент.

Мережева ізоляція у контейнерних кластерах потребує окремої уваги, оскільки за замовчуванням у більшості конфігурацій Kubernetes усі поди у

кластері можуть вільно спілкуватися між собою, що суперечить принципу найменших привілеїв. Network Policies — ресурс Kubernetes, що дозволяє декларативно описати дозволений мережевий трафік між подами та неймспейсами — є базовим інструментом мікросегментації. Для реалізації Network Policies необхідний сумісний CNI-плагін (Container Network Interface): Calico, Cilium або Antrea. Cilium, що базується на технології eBPF (extended Berkeley Packet Filter), надає можливості фільтрації трафіку не лише на рівні IP/порт, але й на рівні HTTP/gRPC методів, забезпечуючи так звану «layer 7» мережеву безпеку безпосередньо в ядрі Linux без накладних витрат sidecar-проксі. Для сервісів, що потребують взаємної автентифікації та шифрування трафіку між мікросервісами, застосовується Service Mesh (Istio, Linkerd), що автоматично забезпечує mTLS (mutual TLS) між усіма подами без змін у коді застосунків. Рисунок 7.3 ілюструє багаторівневу модель безпеки контейнерного середовища.

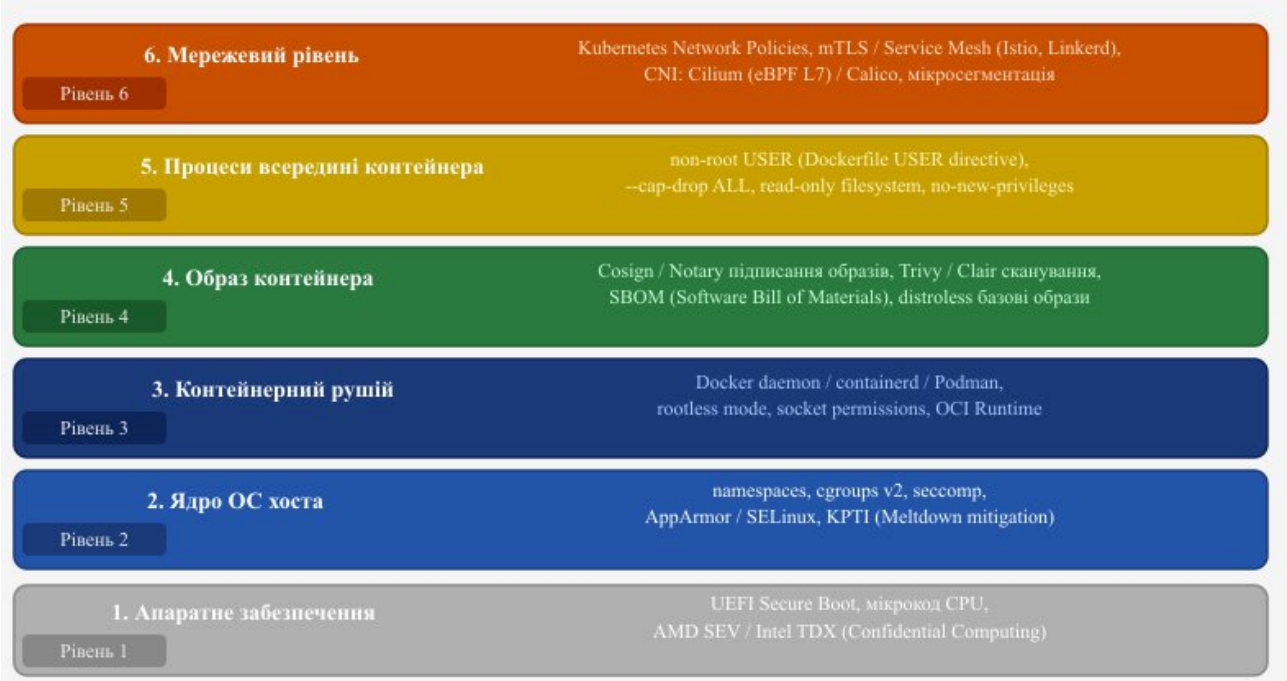

Рисунок 7.3 — Багаторівнева модель безпеки контейнерного середовища

7.4. Роль віртуальних систем у тестуванні, моделюванні загроз та резервному копіюванні

Ізольованість та відтворюваність є найціннішими властивостями віртуальних середовищ з точки зору забезпечення якості програмного забезпечення та кібербезпеки. Тестування у VM або контейнерах гарантує, що результати не залежатимуть від специфіки конкретного фізичного середовища, а після завершення тестів середовище може бути повністю очищено або повернуто до початкового стану. Це вирішує давню проблему «але у мене все працює», яка виникала через відмінності між середовищами розробника, тестувальника та продуктивним сервером. Контейнери Docker у поєднанні з Docker Compose дозволяють описати всю інфраструктуру застосунку — бекенд,

бази даних, черги повідомлень, кеш — у єдиному файлі docker-compose.yml та відтворити її на будь-якій машині однією командою docker compose up, що є практичним втіленням принципу Infrastructure as Code.

Практики DevOps та CI/CD (Continuous Integration / Continuous Delivery) неможливо уявити без контейнеризації. Платформи безперервної інтеграції — Jenkins, GitLab CI, GitHub Actions, CircleCI — використовують контейнери як ізольовані середовища виконання для кожного кроку конвеєра: збирання артефактів, запуску автоматичних тестів, статичного аналізу коду, сканування безпеки (SAST, DAST, SCA). Кожен запуск конвеєра стартує з чистого образу, що виключає «забруднення» середовища попередніми запусками та усуває клас помилок, пов'язаних з накопиченим станом середовища виконання. Після завершення тестування готовий образ контейнера публікується у реєстрі та автоматично розгортається у staging або production середовищі, забезпечуючи повторюваний та аудитований процес доставки від коміту розробника до запущеного застосунку за лічені хвилини. Рисунок 7.4 ілюструє типову CI/CD-pipeline з використанням контейнерів.

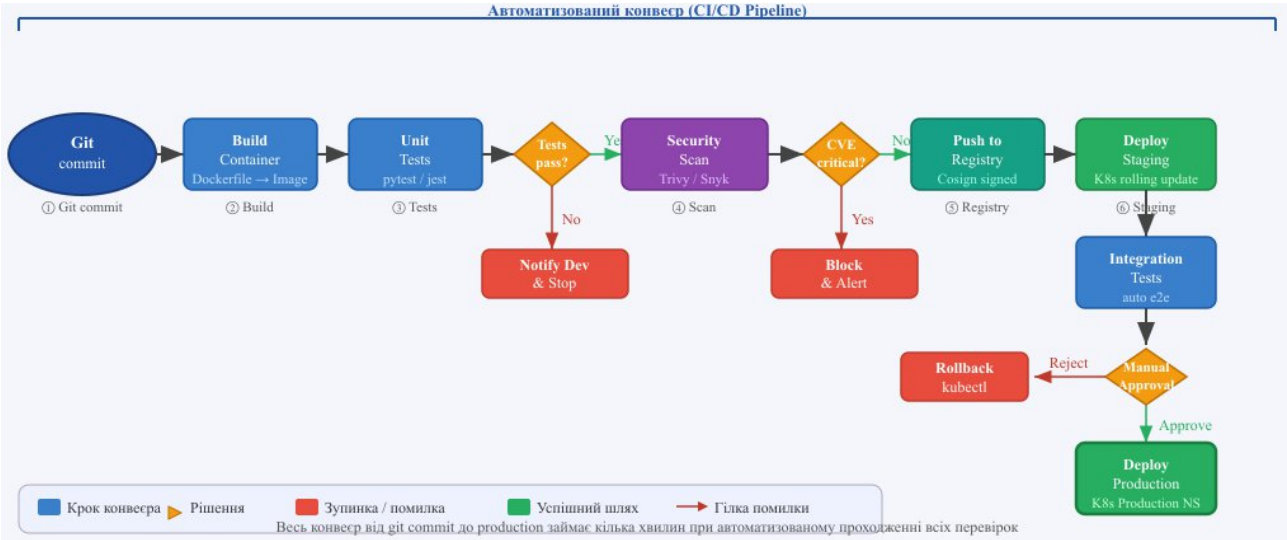

Рисунок 7.4 — CI/CD-pipeline з використанням контейнерів та Kubernetes

Моделювання кіберзагроз та проведення навчань з реагування на інциденти — ще одна важлива область застосування віртуальних середовищ. Кіберполігони (cyber ranges) — спеціалізовані платформи для відпрацювання навичок атаки та захисту — будуються виключно на базі гіпервізорів та контейнерів, що дозволяє відтворити реалістичну корпоративну інфраструктуру у повній ізоляції від виробничих мереж. Платформи HackTheBox, TryHackMe, PentesterLab та корпоративні рішення на базі VMware vSphere або OpenStack надають дослідникам безпеки ізольовані машини з навмисно введеними вразливостями для відпрацювання технік тестування на проникнення. Фахівці Blue Team отримують можливість налаштовувати системи виявлення вторгнень, SIEM-рішення та засоби реагування у безпечному середовищі, де можна без наслідків відтворити реальні кібератаки — включаючи атаки типу ransomware, APT-кампанії та атаки на Active Directory, — а потім

проаналізувати логи та вдосконалити детектуючі правила. Підготовка до сертифікацій з кібербезпеки (OSCP, CEH, CISM) так само цілком і повністю спирається на лабораторні середовища на базі VM.

Знімки стану (snapshots) є одним із найпотужніших інструментів управління VM, що не має прямого аналога у фізичній інфраструктурі. Знімок фіксує повний стан VM у певний момент: вміст оперативної пам'яті, стан диска, конфігурацію пристроїв та стан процесора. Це дозволяє миттєво повернути VM до відомого стабільного стану після невдалого оновлення, зараження шкідливим програмним забезпеченням або будь-якої іншої небажаної зміни. У контексті безпеки знімки активно використовуються для аналізу шкідливого програмного забезпечення (malware analysis): дослідник запускає підозрілий зразок у VM, спостерігає за його поведінкою за допомогою інструментів динамічного аналізу (Cuckoo Sandbox, ANY.RUN), після чого відновлює чистий стан без будь-яких слідів зараження. Платформи автоматизованого аналізу шкідливого ПЗ повністю побудовані на цій концепції: для кожного зразка автоматично запускається чиста VM зі знімка, виконується зразок, збираються мережеві артефакти та записи системних викликів, а потім VM скидається назад до чистого стану.

Слід однак пам'ятати, що знімки не є повноцінним резервним копіюванням — вони зберігаються разом з VM, займають дисковий простір на тому ж сховищі та зникають при її видаленні. Довгий ланцюжок знімків може суттєво знизити продуктивність VM, оскільки кожна операція запису потребує перенаправлення на поточний delta-файл знімка. Тому використання знімків для відновлення після катастроф потребує доповнення резервним копіюванням на зовнішні сховища. Повноцінне резервне копіювання у середовищах з гіпервізорами набуває принципово нових можливостей завдяки механізму Changed Block Tracking (CBT): замість того щоб щоразу знімати повний образ диска VM, система відстежує лише ті блоки, що змінилися з часу попереднього резервного копіювання, і зберігає лише різницю (incremental backup). У поєднанні з можливістю створення знімка VM без її зупинки — hot backup через VMware VADP або Hyper-V VSS — це дозволяє забезпечити Recovery Point Objective (RPO) у кілька годин навіть для найнавантаженіших систем без будь-якого переривання сервісу. У таблиці 7.4 наведено порівняння ключових метрик резервного копіювання та відновлення для різних підходів.

Таблиця 7.4 — Порівняння підходів до резервного копіювання та відновлення

Підхід	RPO	RTO	Обсяг зберігання	Типові інструменти
Повне резервне копіювання (Full backup)	Години–доби	Години	Велике (повний образ)	Veeam, Commvault
Інкрементне (CBT)	Години	Хвилини–години	Мале (лише зміни)	Veeam, Nakivo
Знімок (локальний	Хвилини	Секунди	Середнє (delta-	VMware,

Підхід	RPO	RTO	Обсяг зберігання	Типові інструменти
Snapshot)			файли)	Hyper-V
Реплікація VM між ЦОД (DR)	Хвилини	Хвилини	Велике (повний дубль VM)	vSphere Replication, Zerto
Velero (Kubernetes)	Години	Хвилини–години	Мале (маніфести + PV)	Velero + MinIO/S3
Знімок + копіювання offsite	Хвилини–години	Хвилини	Помірне	Zerto, Cohesity

Рішення Veeam Backup & Replication, VMware vSphere Data Protection та Commvault реалізують інкрементний підхід для VM, тоді як Velero є популярним інструментом резервного копіювання для Kubernetes-кластерів, зберігаючи не лише дані Persistent Volumes, але й маніфести всіх ресурсів кластера у об'єктному сховищі (S3-сумісне сховище: MinIO, AWS S3, Google Cloud Storage). Реплікація VM між географічно розподіленими ЦОД (Disaster Recovery) є складним, але критично важливим сценарієм. Технологія VMware vSphere Replication дозволяє асинхронно реплікувати VM на відновлювальний сайт з мінімальним RPO у 5 хвилин, а у разі збою первинного ЦОД — виконати аварійне перемикання (failover) за лічені хвилини за допомогою VMware Site Recovery Manager, що автоматизує послідовність запуску сервісів з урахуванням залежностей. У хмарних середовищах Amazon AWS, Microsoft Azure та Google Cloud концепція Availability Zones та Multi-Region розгортань забезпечує реплікацію ресурсів між фізично відокремленими ЦОД, наближаючи RTO до нуля для найкритичніших сервісів.

Моніторинг та спостережуваність (observability) у віртуалізованих і контейнерних середовищах потребують спеціалізованих інструментів, оскільки традиційні агенти на основі постійно запущених процесів погано вписуються у модель ефемерних контейнерів, що існують лише кілька секунд або хвилин. Спостережуваність у сучасному розумінні складається з трьох стовпів: метрики (показники продуктивності у часі), логи (структуровані записи подій) та трейси (розподілені ланцюжки виконання запитів між мікросервісами). Стек Prometheus + Grafana + Loki + Jaeger/Tempo став де-факто стандартом відкритого ПЗ для Kubernetes-середовищ: Prometheus використовує модель pull — він сам опитує endpoint-и метрик у pod-ів, а CustomResource ServiceMonitor автоматично виявляє нові контейнери без ручного налаштування. Збір логів вирішується за допомогою DaemonSet-агентів Fluentd або Fluent Bit, що запускаються на кожному вузлі кластера та перехоплюють потоки stdout/stderr усіх контейнерів і пересилають їх у централізоване сховище (Elasticsearch, Loki, OpenSearch). Ця архітектура забезпечує повноцінну видимість стану кластера без необхідності встановлення агентів всередині кожного образу, що є принципово важливим для ефективного реагування на інциденти та ретроспективного аналізу.

Підсумовуючи, варто наголосити, що вибір між повноцінною VM та контейнером не є питанням «кращого» або «гіршого» підходу — обидві технології органічно доповнюють одна одну у сучасній IT-інфраструктурі. Гіпервізорна віртуалізація забезпечує сильну апаратну ізоляцію та підтримку різнорідних операційних систем — Windows, Linux, FreeBSD можуть спільно існувати на одному фізичному сервері, обслуговуватися єдиною командою і мати уніфіковані процедури резервного копіювання та відновлення. Контейнеризація забезпечує щільне пакування навантажень, надзвичайну портабельність та швидкість ітерацій розробки і розгортання, роблячи мікросервісну архітектуру практично реалізованою. Зрілі організації, як правило, поєднують обидві технології: Kubernetes-кластери розгортаються на VM (а не на «голому залізі»), отримуючи переваги апаратної ізоляції вузлів при збереженні гнучкості контейнерів для мікросервісних застосунків, а тенденція до впровадження Confidential Computing — VM з апаратно-зашифрованою пам’яттю (AMD SEV, Intel TDX) — знаменує наступний крок у розвитку технологій ізоляції та захисту обчислювальних середовищ.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Жураковський Б. Ю. Комп'ютерні мережі. Частина 1. [навчальний посібник] / Б. Ю. Жураковський, І.О. Зенів. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 336 с.
2. Матвєєва Н.О. Основи роботи та програмування в операційній системі Linux: навчальний посібник /Н.О. Матвєєва, В.С. Хандецький, О.В. Спірінцева - Д.: ЛІРА, 2018, −156 с.
3. Харченко В. П., Знаковська Є. А., Бородін В. А. Операційні системи та системи програмування: навч. посіб /В. П. Харченко, Є. А. Знаковська, В. А. Бородін – К.: Вид-во Нац. авіац. ун-ту «НАУ-друк», 2012.– 360с.
4. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.
5. Погребняк Б. І. Операційні системи : навч. посібник / Б. І. Погребняк, М. В. Булаєнко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2018. – 104 с.
6. Операційні системи: [Електронний ресурс]: навч. посіб. для студ. спеціальності 123 «Комп'ютерна інженерія» / В. Г. Зайцев, І. П. Дробязко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.
7. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts / A. Silberschatz, P. B. Galvin, G. Gagne. – 10th ed. – Hoboken, NJ : John Wiley & Sons, 2018. – 976 p.
8. Tanenbaum A. S., Bos H. Modern Operating Systems / A. S. Tanenbaum, H. Bos. – 4th ed. – Boston : Pearson Education, 2015. – 1136 p.
9. Stallings W. Operating Systems: Internals and Design Principles / W. Stallings. – 9th ed. – Boston : Pearson Education, 2018. – 864 p.
10. Arpaci-Dusseau R. H., Arpaci-Dusseau A. C. Operating Systems: Three Easy Pieces / R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau. – Madison, WI : Arpaci-Dusseau Books, 2018. – 720 p.
11. Love R. Linux Kernel Development / R. Love. – 3rd ed. – Indianapolis : Addison-Wesley Professional, 2010. – 464 p.