МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДРОГОБИЦЬКИЙ ДЕРЖАВНИЙ ПЕДАГОГІЧНИЙ УНІВЕРСИТЕТ
імені ІВАНА ФРАНКА

Кафедра фізики та інформаційних систем

«До захисту допускаю»
завідувач кафедри фізики
та інформаційних систем,
кандидат фізико-математичних наук,
доцент _____ Віталій ГОЛЬСЬКИЙ
«____»_____2026 р.

РОЗРОБЛЕННЯ ЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ

Спеціальність: 122 Комп'ютерні науки

Кваліфікаційна робота

на здобуття кваліфікації – *бакалавр з комп'ютерних наук*

Автор роботи: ВЕЖДЕЛ Андрій Іванович_____
Підпис

Науковий керівник: кандидат фізико-математичних наук, доцент
КАРПИН Дмитро Степанович _____
підпис

Дрогобич, 2026

Дрогобицький державний педагогічний університет
імені Івана Франка

Зав. кафедрою

_____ _____
(підпис) (дата)

Завдання

на підготовку кваліфікаційної бакалаврської роботи

1. Тема: ***«Розроблення ядра операційної системи»***
2. Керівник ***кандидат фізико-математичних наук, доцент Карпин Д.С.***
3. Студент ***Веждел Андрій Іванович***
(прізвище, ім'я, по батькові)
4. Перелік питань, що підлягають висвітленню у кваліфікаційній роботі
   1. Аналіз предметної області розроблення ядер операційних систем.
   2. Огляд існуючих операційних систем, навчальних і дослідницьких платформ та визначення вимог до розроблюваного ядра операційної системи.
   3. Вибір середовища, мови програмування, бібліотек, інструментів збірки та засобів тестування.
   4. Проєктування архітектури ядра операційної системи та його основних підсистем.
   5. Реалізація підсистем керування пам'яттю, обробки переривань, графічного виводу та введення з пристроїв.
   6. Реалізація інтерфейсу системних викликів, байткодової віртуальної машини та прикладного фреймворку.
   7. Тестування, діагностика та перевірка працездатності ядра операційної системи в середовищі QEMU.
5. Список рекомендованої літератури

   1. TheRustProgrammingLanguage / S. Klabnik, C. Nichols.https://doc.rust-lang.org/book/
   2. RustEmbeddedBook / RustEmbeddedWorkingGroup.https://docs.rust-embedded.org/book/
   3. BareMetalRust / Rustcommunity.https://docs.rust-embedded.org/
   4. QEMU EmulatorDocumentation.https://www.qemu.org/docs/

6. Етапи підготовки роботи

№	Назва етапу	Термін виконання	Термін звіту перед керівником, кафедрою
1.	Аналіз предметної області розроблення ядер операційних систем.	Жовтень 2025	01.11.2025
2.	Огляд існуючих операційних систем, навчальних і дослідницьких платформ.	Листопад 2025	20.11.2025
3	Вибір мови програмування, бібліотек, інструментів збірки та середовища тестування.	Грудень 2025	10.12.2025
4.	Проєктування архітектури ядра операційної системи.	Грудень 2025	20.12.2025
5.	Реалізація основних підсистем ядра	Лютий-травень 2026	10.05.2026

7. Дата видачі завдання _____
8. Термін подачі роботи керівнику _____
9. З вимогами до виконання кваліфікаційної роботи і завданням ознайомлений

_____
(підпис студента)

10. Керівник _____
(підпис)

ЗАХИСТ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Тема: «РОЗРОБЛЕННЯ ЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ».

Оцінка за стобальною шкалою: _____

Оцінка за національною чотирибальною шкалою: _____

Коротка мотивація захисту:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____ Голова ЕК _____ _____

Дата підпис прізвище, ім’я

Секретар ЕК _____ _____

підпис прізвище, ім’я

АНОТАЦІЯ

Веждел А. І. Розроблення ядра операційної системи. Кваліфікаційна робота, Дрогобицький державний педагогічний університет імені Івана Франка, Дрогобич, 2026.

У кваліфікаційній роботі розглянуто проблему розроблення експериментального ядра операційної системи для архітектури x86_64. Актуальність теми зумовлена потребою у вивченні принципів побудови низькорівневого системного програмного забезпечення, механізмів керування пам’яттю, обробки переривань, взаємодії з апаратними пристроями та організації базового середовища виконання програм.

Метою роботи є розроблення ядра операційної системи мовою Rust у режимі no_std з реалізацією базових підсистем, необхідних для запуску, ініціалізації та функціонування мінімального системного середовища. У роботі обґрунтовано вибір мови Rust як засобу системного програмування, що поєднує можливість низькорівневого доступу до апаратних ресурсів із засобами підвищення безпеки роботи з пам’яттю.

У межах роботи реалізовано інфраструктуру збірки та запуску ядра в середовищі QEMU, використано завантажувач для передачі керування ядру, налаштовано базові структури процесора, таблиці дескрипторів, обробники апаратних переривань і процесорних винятків. Розроблено підсистему керування фізичною та віртуальною пам’яттю, механізми сторінкової адресації, heap-алокатор, графічний вивід через framebuffer, засоби введення з клавіатури та миші PS/2, а також інтерфейс системних викликів.

Ключові слова: ядро операційної системи, Rust, no_std, x86_64, QEMU, керування пам’яттю, framebuffer, переривання, системні виклики, PS/2, байткодова віртуальна машина.

ABSTRACT

Vezhdel A. I. Development of an Operating System Kernel. Qualification thesis, Drohobych Ivan Franko State Pedagogical University, Drohobych, 2026.

The qualification thesis addresses the development of an experimental operating system kernel for the x86_64 architecture. The relevance of the topic is determined by the need to study the principles of low-level system software development, memory management mechanisms, interrupt handling, interaction with hardware devices, and the organization of a basic runtime environment for programs.

The aim of the thesis is to develop an operating system kernel using the Rust programming language in no_std mode and to implement the basic subsystems required for booting, initialization, and operation of a minimal system environment. The thesis substantiates the choice of Rust as a system programming language that combines low-level access to hardware resources with mechanisms that improve memory safety.

As part of the work, the kernel build and launch infrastructure was implemented using the QEMU environment, and a bootloader was used to transfer control to the kernel. Basic processor structures, descriptor tables, hardware interrupt handlers, and processor exception handlers were configured. The work also includes the development of physical and virtual memory management subsystems, paging mechanisms, a heap allocator, graphical output through a framebuffer, input processing from PS/2 keyboard and mouse devices, and a system call interface.

Keywords: operating system kernel, Rust, no_std, x86_64, QEMU, memory management, framebuffer, interrupts, system calls, PS/2, bytecode virtual machine.

ЗМІСТ

ВСТУП

Актуальність теми:

Операційні системи є основним шаром програмного забезпечення, який забезпечує взаємодію між прикладними програмами та апаратним забезпеченням. Незважаючи на десятиліття розвитку в цій галузі, проблема створення мінімалістичних і прозорих ядер з контрольованою архітектурою залишається актуальною, особливо щодо вбудованих систем, Інтернету речей (IoT)-пристроїв і спеціалізованих обчислювальних платформ, де єдині рішення на основі Linux надто складні.

На користь актуальності роботи є ще один аргумент щодо вибору мови Rust як засобу реалізації. Rust не використовує збирач сміття, виключаючи широкий спектр помилок управління пам’яттю на рівні компілятора. Зростаюча роль Rust у системному програмуванні демонструється ініціативами Redox OS, підсистемою Rust у ядрі Linux та інтеграцією Rust у системні компоненти Microsoft.

Розробка ядра операційної системи з нуля дозволяє досліджувати всі рівні системного стеку, від управління фізичною пам’яттю до побудови прикладного фреймворку, не залежаючи від зовнішніх "blackbox". Це формує практичний досвід і розуміння реальних компромісів між продуктивністю, безпекою та комплексністю.

Мета кваліфікаційної роботи є в розроблення ядра операційної системи для архітектури x86_64, реалізована мовою Rust у режимі no_std. Ядро містить підсистеми управління віртуальною та фізичною пам’яттю, обробку апаратних переривань і процесорних винятків, графічний вивід через framebuffer, обробку вводу з пристроїв PS/2, інтерфейс системних викликів, а також вбудовану віртуальну машину з байткодом.

У ході роботи було вирішено такі завдання: розроблено інфраструктуру збірки на базі cargonightly з кастомним таргетом x86_64-duxa_os.json та автоматизованим запуском у QEMU. Реалізовано підсистему управління

пам’яттю, яка включає GlobalFrameAllocator, який базується на атомарному CAS-циклі, чотирирівневих таблицях сторінок і п’яти алгоритмів heap-алокатора. Система обробки переривань, заснована на PIC 8259A, може працювати з таймером, клавіатурою та мишею. Розроблено графічну підсистему на основі framebuffer BGR 32-бітів, яка включає тіньовий буфер, відстеження змін у тильній ділянці та хешування FNV рядків пікселів. Реалізовано власну стековубайткодову віртуальну машину (VM), яка має фреймворк застосунків із 22 типами інструкцій і трьома вбудованими програмами, а також подієвою моделлю вводу. Застосування системи власності Rust та безпечних абстракцій над апаратними ресурсами дозволяє створювати повнофункціональне ядро операційної системи без традиційних помилок управління пам’яттю, властивих C-коду. Реалізація цих компонентів демонструє це.

Об’єктом дослідження є процес розробки ядра операційної системи на архітектурі x86_64 без стандартної бібліотеки. Процес включає реалізацію апаратних абстракцій, підсистем управління пам’яттю, обробки переривань і прикладного фреймворку.

Предметом дослідження є інженерні підходи та архітектурні рішення, які застосовуються під час розробки ядра операційної системи на мові Rust; це включає механізми управління фізичною та віртуальною пам’яттю; обробка апаратних переривань і винятків процесора; графічний вивід за допомогою framebuffer; взаємодія з пристроями PS/2; та інтерпретація байткодових програм у просторі ядра.

Методи дослідження: у роботі використано методи аналізу архітектурних підходів до побудови операційних систем, моделювання структури ядра, алгоритмізації низькорівневого рендерингу через framebuffer, проєктування підсистем керування пам’яттю, експериментальної реалізації компонентів у середовищі Rust no_std, а також тестування ядра в QEMU.

В рамках дослідження особливу увагу приділено застосуванню методів безпечного керування пам’яттю Rust і тому, як вони впливають на надійність і коректність системних компонентів ядра.

Завдання, винесені на кваліфікаційну роботу:

Проаналізувати сучасні підходи до розробки низькорівневих компонентів операційних систем та обґрунтувати вибір мови Rust. Розробити інфраструктуру збірки ядра з кастомною цільовою платформою, підтримкою no_std та автоматизованим запуском у QEMU. Реалізувати систему управління пам'яттю, що охоплює фізичний алокатор фреймів, чотирирівневу сторінкову адресацію та декілька алгоритмів heap-алокатора. Побудувати систему обробки апаратних переривань та процесорних винятків з підтримкою DoubleFault на окремому IST-стеку. Реалізувати графічну підсистему на базі framebuffer з тіньовим буфером та оптимізованим рендером. Розробити драйвери PS/2 клавіатури та миші з кільцевими буферами. Реалізувати інтерфейс системних викликів через вектор IDT 120. Розробити власну стековубайткодову ВМ та фреймворк застосунків з трьома вбудованими програмами і подієвою моделлю вводу.

Структура роботи:

Робота складається зі вступу, трьох основних розділів, висновків та списку використаних джерел. Перший розділ містить опис архітектури системи, аналіз вимог, огляд існуючих рішень та технологічний стек. Другий розділ визначає функціональні та нефункціональні вимоги до ядра. Третій розділ детально описує реалізацію ключових підсистем: управління пам'яттю, обробку переривань, графічну підсистему, байткодову ВМ та фреймворк застосунків.

Результати роботи доповідались на XIII Міжнародній науково-практичній конференції студентів та викладачів факультету фізики, математики, економіки та інноваційних технологій «Актуальні проблеми сучасної науки» (м. Дрогобич, 27–30 квітня 2026 р.).

За результатами роботи підготовлено наукову статтю: Карпин Д., Карпин А., Гарбич-Мошора О., Веждел А. «Методи проєктування та реалізації безпечного ядра операційної системи на rust у середовищі no_std для архітектури x86_64», опубліковану в журналі «Наука і техніка сьогодні» у 2026 році, № 5(59).

РОЗДІЛ І. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис

Розроблювана операційна система базується на мінімалізмі, модульності, безпечній роботі з пам’яттю та передбачуваній поведінці в середовищі низькорівневої системи. Система створюється без середовища виконання, бібліотеки чи інших службових компонентів. Ядро створюється мовою Rust у конфігурації no_std, що дозволяє зберігати гарантії типобезпечності, одночасно контролюючи роботу з апаратними ресурсами.

Підсистема завантаження (Bootinfrastructure):

Оскільки реалізація завантажувача не входить у межі даної роботи, операційна система використовує готовий завантажувач (bootloadercrate), адаптований для Rust. Завантажувач відповідає за:

- підготовку базового середовища виконання у режимі protectedmode або longmode (x86_64),
- передавання керування ядру,
- передачу мапи фізичної пам’яті та ключових структур,
- запуск ядра на визначеній точці входу.

Ядро отримує лише базові гарантії 64-бітного середовища виконання, після чого самостійно ініціалізує всі необхідні системні структури.

Система обробки помилок та винятків (ExceptionHandling):

Архітектура ОС включає повноцінну підтримку апаратних винятків процесора x86_64.

Підсистема складається з:

- таблиці дескрипторів переривань (IDT),
- набору обробників для кожного типу винятку,
- механізму виводу діагностичної інформації через framebuffer,
- окремого стеку для подвійних помилок (DoubleFaultStack),

- ізольованого обробника DoubleFault, який запобігає TripleFault і некоректному завершенню роботи системи.

Такий підхід дозволяє гарантувати стабільну роботу у випадку будь-яких помилок, включаючи ті, що викликані некоректними доступами до пам’яті або збоями системних інструкцій.

Підсистема переривань (InterruptHandlingSubsystem):

Архітектура обробки переривань включає:

- ініціалізацію та конфігурацію контролера переривань (PIC або APIC),
- реєстрацію обробників апаратних IRQ,
- механізм протидії "спам-перериванням" (EOI - EndOfInterrupt),
- підтримку обробки таймера PIT/APIC,
- реалізацію драйвера клавіатури PS/2.

Модуль побудований з урахуванням суворого контролю за доступом до спільних даних і забезпечує безпечні точки входу до ядра через низькорівневі апаратні події.

Підсистема пам’яті (MemoryManagementSubsystem):

Це одна з найважливіших частин архітектури, спрямована на безпечне використання фізичної та віртуальної пам’яті.

Підсистема включає такі компоненти:

- Фізичний менеджер пам’яті (PhysicalMemoryManager)

Забезпечує:

  - роботу з мапою пам’яті, переданою завантажувачем,
  - алокацію/деалокацію фізичних сторінок,
  - виділення пам’яті для стеків, таблиць сторінок тощо.

- Віртуальна пам’ять (PagingSystem)

Передбачає:

  - створення та керування ієрархією таблиць сторінок,

- організацію адресного простору ядра,
- ізоляцію критичних ділянок пам’яті,
- надання API для безпечного мапінгу та відмапінгу сторінок.

- Динамічний алокаторHeap-пам’яті

  Забезпечує:

  - реалізацію алокатора у режимі no_std,
  - підтримку глобального алокатора через #[global_allocator],
  - можливість тестування альтернативних алгоритмів (bumpallocator, linkedlistallocator, slab і т.д.).

Підсистема вводу (InputSubsystem):

Підсистема вводу обробляє два фізичних пристрої: PS/2 клавіатуру (IRQ1) та PS/2 мишку (IRQ12). Обидва підключені до одного контролера PS/2 і обслуговуються через апаратні переривання з буферизацією у кільцевих буферах по 256 байт. Підсистема забезпечує:

- обробку низькорівневихscan-кодів,
- формування ASCII-послідовностей,
- буферизацію натискань,
- взаємодію з драйвером переривань,
- базовий API для використання іншими компонентами ядра.
- декодування 3-байтових пакетів PS/2 миші (dx, dy, стан кнопок);
- інвертування осі Y для відповідності екранним координатам;
- доставку MouseEvent у подієву систему AppHost.

Відображення та візуалізація роботи системи:

Графічний вивід реалізовано через framebuffer у форматі BGR 32-bit, наданий завантажувачем. Система використовує shadowbuffer у RAM та тайловий механізм детекції змін (тайли 32x32 пікселі з хешуванням рядків) для мінімізації операцій запису у відеопам'ять. Зокрема:

- shadowbuffer зберігає повний кадр у RAM; у VRAM передаються тільки змінені тайли 32x32 пікселі;
- система підтримує тему оформлення (Theme) з сімома параметрами кольорів: background, text, accent, surface, border, muted, on_accent;
- конвеєр рендеру (RenderList) збирає команди Clear, FillRect, FillRoundedRect, StrokeRect, Text і виконує їх пакетно на FramebufferWriter.

Підсистема байткодової віртуальної машини:

Ядро містить власну стековубайткодовувіртуальну машину, що виконує програми, представлені як вектор інструкцій з мітками. Віртуальна машина підтримує 22інструкції, розбитих на категорії: стекові операції (Push, Dup, Swap, Drop), арифметика та порівняння (Add, Sub, Mul, Div, Mod, Neg, Eq, Neq, Gt, Lt), керування потоком (Jmp, Jz, Jnz, Halt), локальна пам'ять (Load/Store по 256 слотах) та графіка (SetPixel, FillRect, ClearScr, Present). Обмеження середовища: стек 1024 значення i64, вихідний буфер 8192 байт.

Підсистема застосунків та користувацький інтерфейс:

AppHost управляє трьома вбудованими застосунками через систему вкладок: TerminalApp (F1), LogsApp (F2), EditorApp (F3). Кожен застосунок реалізує трейтApp з методами init(), on_event(), layout(), collect_render(), collect_overlay() та focus_blocks(). Переключення вкладок виконується клавішами F1/F2/F3, Alt+Tab або Alt+1..9. TerminalApp розпізнає команди: help, test, test_paging, test_memory, test_asm, vm_help, vm_demo, vm_run, echo, info, clear, exit.

Підсистема системних викликів:

Інтерфейс системних викликів визначає 18 номерів (вектор переривань 120). З них повністю реалізовано 6: write(1), getpid(24), sleep(60), gettime(61), mmap(40), brk(42). Виклики fork(21) та exec(22) реалізовані частково. Решта (read, open, close, exit, wait, munmap, kill, signal, chdir, mkdir) повертають ENOSYS або виконують spin-loop. Диспетчер читає SyscallContext зі стеку переривання; помилки кодуються у стилі POSIX.

1.2. Визначення вимог до розроблюваної платформи

Функціональні вимоги

Підсистема завантаження та ініціалізації:

- Система повинна коректно отримувати керування від завантажувача в режимі *x86_64 longmode*.
- Ядро повинно ініціалізувати базові структури процесора (GDT, IDT).
- Система повинна отримувати карту пам’яті та framebuffer від завантажувача (bootloadercrate).
- Після завершення ініціалізації система повинна переходити до основної логіки ядра.

Підсистема графічного виводу (FramebufferOutput)

- ОС повинна виконувати виведення інформації через framebuffer, наданий завантажувачем.
- Повинна підтримуватись побудова базових графічних примітивів (піксель, лінія, прямокутник).
- Система повинна підтримувати рендеринг тексту через растровий або bitmap-шрифт.
- Повинен бути реалізований базовий графічний консольний інтерфейс для логів і діагностики.
- Вивід через framebuffer використовується для:
    - повідомлень ініціалізації,
    - відображення помилок,
    - debug-інформації,
    - результатів тестів.

Механізм обробки критичних помилок

- ОС повинна мати глобальний panichandler, який виводить діагностику через framebuffer.

- Повинна існувати можливість безпечного завершення або зупинки системи після фатальної помилки.
- Система повинна відображати трасування стеку, код помилки та контекст.

Обробка апаратних винятків

- ОС повинна підтримувати стандартні винятки x86_64 (divisionerror, pagefault, invalidopcode тощо).
- Для кожного винятку повинен бути реалізований окремий обробник із виводом інформації на екран.
- Повинна бути правильно сформована таблиця IDT.
- Має бути реалізований DoubleFaulthandler на окремому стеку (IST).
- Система повинна уникати TripleFault.

Обробка апаратних переривань

- Підтримка PIC або APIC (мінімум - PIC).
- Реалізація переривання клавіатури (IRQ1).
- Коректне надсилання EndOfInterrupt (EOI).
- Наявність таймера PIT для задач синхронізації.
- Механізм реєстрації або підміни обробників.

Підсистема пам’яті

- *Фізична пам’ять*
  - Отримання карти фізичної пам’яті від завантажувача.
  - Реалізація менеджера фізичних сторінок.
  - Виділення пам’яті під стек, таблиці сторінок, буфери.
- *Віртуальна пам’ять*
  - Увімкнення пейджингу.
  - Реалізація 4-рівневої ієрархії таблиць сторінок.
  - API для map/unmap сторінок.

    - Ізоляція адресного простору ядра.
- *Heap-пам'ять*
    - Глобальний алокатор для alloc.
    - Підтримка декількох алгоритмів алокації.
    - Оптимізація під системне програмування.

Підсистема вводу

- Підтримка PS/2 клавіатури через IRQ1.
- Розпізнавання scan-кодів.
- Перетворення scan-кодів у ASCII/Unicode.
- Буфер введення і API для читання клавіш.

Підсистема тестування

- Юніт-тести в no_std (де можливо).
- Інтеграційні тести в QEMU.
- Автоматизація збірки та запуску тестів.
- Вивід результатів тестів через framebuffer або serialport.

Інструменти збірки та запуску:

Створення build-скрипту для:

- Збірки ядра,
- Створення target,
- Формування дискового образу,
- Автоматичного запуску в QEMU.
- Підтримка параметрів debug/release.
- Генерація ISO/IMG-образу, що може бути завантажений як ОС.

1.3. Огляд конкурентів

На сьогоднішній день існує низка операційних систем та дослідницьких платформ, що реалізують різні підходи до архітектури ядра, модульності та управління пам'яттю. Ці проєкти слугують як готовими рішеннями для

практичного використання, так і моделями для навчання та експериментів у галузі системного програмування. Значна частина сучасних ініціатив спрямована на підвищення надійності, безпеки та передбачуваності роботи системи, зокрема шляхом застосування мов програмування із гарантіями безпечного керування пам’яттю, таких як Rust.

Попри те, що жоден з наявних проєктів повністю не відповідає унікальним вимогам та цілям розроблюваної платформи, аналіз актуальних рішень дозволяє визначити їхні сильні та слабкі сторони, виділити ключові архітектурні підходи та зіставити їх із концепціями, покладеними в основу цієї роботи. Саме тому доцільним є огляд найбільш релевантних аналогів і конкурентів, що займають вагоме місце у сфері дослідження операційних систем.

Розглянемо основні з них:

1. Redox OS

- ***Особливості:***Redox OS - операційна система, написана на Rust, з мікрокернелем, драйверами у вигляді ізольованих служб, власним файловим форматом і пакетним менеджером.
- ***Конкурентні переваги:*** Надійність завдяки Rust, POSIX-сумісність, активна спільнота, системні компоненти вже готові (RedoxFS, Orbital GUI).
- ***Обмеження:*** Досить велика та складна кодова база; складно використовувати як мінімалістичну дослідницьку платформу; високий поріг входу в архітектуру.

2. xv6 (MIT)

- ***Особливості:*** Навчальна операційна система, що реалізує основні механізми UNIX у простому вигляді. Написана на C, дуже компактна й широко використовується для навчання.
- ***Конкурентні переваги:*** Ідеальна для вивчення принципів ОС, чітка і проста структура ядра, мінімум залежностей.

- ***Обмеження:*** Написана на C, немає захисту від помилок пам’яті; немає підтримки сучасного обладнання; складно масштабувати до реальної ОС.

3. Minix 3

- ***Особливості:***Мікрокернельна операційна система, де більшість драйверів та серверів працює в user-space. Створена Таненбаумом для надійності та досліджень.
- ***Конкурентні переваги:*** Архітектура мікрокернеля, дуже висока надійність, самовідновлення драйверів, мінімальний код ядра.
- ***Обмеження:*** Старий стек технологій (C), менш активний розвиток, обмежена апаратна сумісність.

4. SerenityOS

- ***Особливості:*** Хобі-ОС з ретро-естетикою, монолітним ядром, власним браузером, файловою системою, юзерспейсом. Відома завдяки високій продуктивності та активності розробників.
- ***Конкурентні переваги:*** Дуже багатий user-space, сучасні інструменти, швидкий розвиток, активна спільнота.
- ***Обмеження:*** Написана на C++ без повної безпеки пам’яті; складна архітектура для навчальної мети; великий обсяг коду.

5. Tock OS

- ***Особливості:*** ОС для вбудованих систем, написана на Rust, з мікрокернельним підходом і суворою ізоляцією процесів.
- ***Конкурентні переваги:*** Сильна безпека, Rust, розроблена спеціально для малих пристроїв, строгі моделі ізоляції.
- ***Обмеження:*** Не призначена для загального призначення; дуже специфічна сфера застосування; обмежений функціональний набір.

6. BareMetal OS

- ***Особливості:*** Високопродуктивна ОС з мінімальним монолітним ядром на x86_64, орієнтована на максимальну швидкість.
- ***Конкурентні переваги:*** Неймовірно маленький розмір ядра, чистий низькорівневий код.
- ***Обмеження:*** Написана на Assembly+C; немає захисту пам’яті; дуже обмежений стек функцій.

1.4. Технологічні аспекти

У процесі розробки операційної системи було використано низку технологій, інструментів та мов програмування, які забезпечують можливість створення високонадійного та безпечного низькорівневого програмного забезпечення. Вибір технологічного стеку ґрунтувався на вимогах до мінімалістичності, контролю над пам’яттю, ізоляції компонентів та можливості запуску в середовищі без стандартної бібліотеки.

Мова програмування Rust

Rust був обраний основною мовою реалізації через:

a. Модель володіння та позичання, що унеможливлює більшість помилок роботи з пам’яттю.

b. Відсутність**GC**(Garbagecollector), що робить систему передбачуваною у плані продуктивності.

c. Можливість роботи в режимі no_std, необхідному для операційних систем.

d. Наявність механізмів безпечної абстракції над апаратними ресурсами.

2. **Бібліотеки**

У розробці використовувалися бібліотеки, серед яких:

a. ***bootloader***- завантажувач, що відповідає за ініціалізацію середовища, підготовку пам’яті та передачу керування ядру;

b. ***x86_64***- набір інструментів для роботи з таблицями сторінок, перериваннями, регістрами та апаратними структурами CPU;

c. ***spin***, ***lazy_static***- для синхронізації та статичної ініціалізації в умовах no_std;

d. ***uart_16550***- для виводу логів у серійний порт.

e. ***pic8259*** - ініціалізація та ремапування контролера переривань PIC 8259A (IRQ0-7 на вектори 32-39, IRQ8-15 на 40-47);

f. ***embedded-graphics*** - растровий рендер тексту через монопросторовийbitmap-шрифт FONT_10X20 (10x20 пікселів);

g. ***os-terminal*** - відображення термінального виводу у framebuffer з підтримкою кольорів та прокрутки.

Середовище віртуалізації

Для запуску та тестування системи буловикористаногіпервізорQEMU - основне середовище для налагодження, яке дозволяє відстежувати переривання, читати вміст регістрів та використовувати серійний порт для дебагу.

Інструменти побудови та автоматизації

Для компіляції та створення повноцінного завантажуваного образу використовувалися:

h. Власний***build-скрипт***, який автоматизує:
   - компіляцію ядра,
   - генерацію дискового образу,
   - запуск у QEMU,
   - очищення артефактів;

i. ***cross-compilationtoolchain*** із власним x86_64 таргетом.

Низькорівневі протоколи та стандарти

У ході реалізації були використані наступні стандарти апаратної взаємодії:

j. ***x86_64 LongMode***– режим роботи процесора, у якому функціонує система;

k. ***InterruptDescriptorTable (IDT)***– таблиця переривань;

l. ***GlobalDescriptorTable (GDT)***– таблиця сегментів;

m. ***Сторінкова адресація (paging)***– основа системи керування пам’яттю;

n. ***Framebuffer протокол***– для графічного текстового виводу.

Тестування та діагностика

Для забезпечення надійності системи застосовувалися:

o. ***Integration-тести*** у qemu з автоматичним завершенням по серійному порту;

p. ***Panic-handler*** з виводом діагностичної інформації у framebuffer;

q. ***Обробники CPU винятків***, що дозволяють детально діагностувати помилки.

РОЗДІЛ ІІ. ВИМОГИ

2.1. Функціональні вимоги

Функціональні вимоги визначають основні функції та можливості, які повинні підтримувати ядро.

Підсистема завантаження та ініціалізації:

- Система повинна коректно отримувати керування від завантажувача в режимі *x86_64 longmode*.
- Ядро повинно ініціалізувати базові структури процесора (GDT, IDT).
- Система повинна отримувати карту пам’яті та framebuffer від завантажувача (bootloadercrate).
- Після завершення ініціалізації система повинна переходити до основної логіки ядра.

Підсистема графічного виводу (FramebufferOutput)

- ОС повинна виконувати виведення інформації через framebuffer, наданий завантажувачем.
- Повинна підтримуватись побудова базових графічних примітивів (піксель, лінія, прямокутник).
- Система повинна підтримувати рендеринг тексту через растровий або bitmap-шрифт.
- Повинен бути реалізований базовий графічний консольний інтерфейс для логів і діагностики.
- Вивід через framebuffer використовується для:
    - повідомлень ініціалізації,
    - відображення помилок,
    - debug-інформації,
    - результатів тестів.

Механізм обробки критичних помилок

- ОС повинна мати глобальний panichandler, який виводить діагностику через framebuffer.
- Повинна існувати можливість безпечного завершення або зупинки системи після фатальної помилки.
- Система повинна відображати трасування стеку, код помилки та контекст.

Обробка апаратних винятків (Exceptions)

- ОС повинна підтримувати стандартні винятки x86_64 (divisionerror, pagefault, invalidopcode тощо).
- Для кожного винятку повинен бути реалізований окремий обробник із виводом інформації на екран.
- Повинна бути правильно сформована таблиця IDT.
- Має бути реалізований DoubleFaulthandler на окремому стеку (IST).
- Система повинна уникати TripleFault.

Обробка апаратних переривань

- Підтримка PIC або APIC (мінімум - PIC).
- Реалізація переривання клавіатури (IRQ1).
- Коректне надсилання EndOfInterrupt (EOI).
- Наявність таймера PIT для задач синхронізації.
- Механізм реєстрації або підміни обробників.

Підсистема пам’яті (MemoryManagement)

Фізична пам’ять

- Отримання карти фізичної пам’яті від завантажувача.
- Реалізація менеджера фізичних сторінок.
- Виділення пам’яті під стек, таблиці сторінок, буфери.

Віртуальна пам’ять

- Увімкнення пейджингу.

- Реалізація 4-рівневої ієрархії таблиць сторінок.
- API для map/unmap сторінок.
- Ізоляція адресного простору ядра.

Heap-пам’ять

- Глобальний алокатор для alloc.
- Підтримка декількох алгоритмів алокації.
- Оптимізація під системне програмування.

Підсистема вводу (KeyboardInput)

- Підтримка PS/2 клавіатури через IRQ1.
- Розпізнавання scan-кодів.
- Перетворення scan-кодів у ASCII/Unicode.
- Буфер введення і API для читання клавіш.
  Підтримка PS/2 миші через IRQ12 з декодуванням пакетів та буферизацією.
  Доставка подій миші (dx, dy, buttons) у підсистему застосунків.

Віртуальна машина

- Віртуальна машина повинна виконувати програми з інструкцій стеку, арифметики, керування потоком та графіки.
- Віртуальна машина повинна ізолювати виконання програм від ресурсів ядра через строго визначений набір інструкцій.

Фреймворк застосунків

- Фреймворк застосунків повинен підтримувати щонайменше 3 вкладки з незалежними застосунками.
- AppHost повинен маршрутизувати події клавіатури і миші до поточного активного застосунку.

Підсистема тестування

- Юніт-тести в no_std (де можливо).
- Інтеграційні тести в QEMU.
- Автоматизація збірки та запуску тестів.
- Вивід результатів тестів через framebuffer або serialport.

Інструменти збірки та запуску

- Створення build-скрипту для:
- Збірки ядра,
- Створення target,
- Формування дискового образу,
- Автоматичного запуску в QEMU.
- Підтримка параметрів debug/release.
- Генерація ISO/IMG-образу, що може бути завантажений як ОС.

2.2. Нефункціональні вимоги

Нефункціональні вимоги визначають характеристики якості, яким має відповідати розроблювана операційна система. Вони регламентують продуктивність, надійність, безпеку, масштабованість та інші властивості, що впливають на експлуатаційні характеристики платформи.

Вимоги до продуктивності

1. Система повинна забезпечувати мінімальні накладні витрати при виконанні базових операцій ядра, включно з обробкою переривань, перемиканням контексту (за наявності), обробкою винятків та доступом до пам’яті.
2. Механізм керування пам’яттю має забезпечувати швидке створення та модифікацію таблиць сторінок.

3. Вивід у framebuffer повинен виконуватись у режимі реального часу з мінімальною затримкою.

Надійність та відмовостійкість

1. Ядро повинно коректно обробляти виняткові ситуації (pagefault, invalidopcode, division-by-zero тощо), не допускаючи переходів у невизначений стан.
2. Для критичних ситуацій має бути реалізований окремий обробник ***doublefault***, який уникає потрійного збою та повного перезавантаження системи.
3. При виникненні помилок система повинна відображати діагностичне повідомлення у framebuffer, забезпечуючи можливість аналізу причин аварії.
4. Усі некритичні помилки мають бути оброблені без зупинки роботи ядра.

Безпека та цілісність

1. Система має повністю уникати небезпечних операцій роботи з пам’яттю, використовуючи механізми Rust: модель ownership, borrow-checker, строгі типи та перевірку меж.
2. Код ядра повинен бути ізольований від зовнішніх бібліотек, що не підтримують no_std, щоб запобігти неконтрольованим викликам або алокаціям.
3. Доступ до фізичної пам’яті повинен здійснюватися тільки через безпечні абстракції.
4. Механізм пейджингу повинен запобігати можливості виконання даних (W^X), якщо це передбачено дизайном.

Масштабованість та розширюваність

1. Архітектура має бути модульною, що дозволить додати драйвери пристроїв, підтримку багатопоточності або планувальника в майбутніх версіях.
2. Механізм алокації пам’яті повинен дозволяти заміну реалізацій (buddy-allocator, slab, bumpallocator тощо) без змін у логіці роботи ядра.

3. Компоненти системи повинні мати чіткі, ізольовані залежності, що дозволяє розширювати функціональність без переписування базових частин ОС.

Портованість

1. Система має працювати на емуляторах QEMU та Bochs без зміни архітектури ядра.
2. Код ядра повинен бути написаний таким чином, щоб спростити перенесення на інші архітектури (наприклад, ARM64) при мінімальних змінах нижчих апаратних шарів.

Тестованість

1. Ядро повинно мати можливість запуску в автоматизованому режимі через QEMU з поверненням exit-коду по серійному порту.
2. Усі ключові компоненти (обробники переривань, алокатор, пейджинг) повинні мати окремі сценарії інтеграційного тестування.
3. Panic-handler повинен забезпечувати інформативний вивід діагностики для аналізу помилок.

Простота супроводу та модифікації

1. Кодова база повинна використовувати сучасні стандарти Rust, включно з ідіоматичним оформленням та відокремленням unsafe-блоків.
2. Компоненти системи повинні мати зрозумілу структурну організацію: розмежування між апаратно-залежними та апаратно-незалежними модулями.
3. Код має бути документований і доповнений коментарями для складних апаратних операцій.

РОЗДІЛ ІІІ. РЕАЛІЗАЦІЯ

3.1. Архітектура системи

Архітектура розробленої операційної системи є модульною, багаторівневою та орієнтованою на максимальну безпеку, передбачуваність і контроль над роботою апаратних ресурсів. Система побудована з мінімальним набором залежностей, використовує no_std середовище та базується на строгій системі типів мови Rust, що дозволяє уникнути більшості низькорівневих помилок, пов’язаних із доступом до пам’яті.

Структура ядра поділена на логічні підсистеми, кожна з яких відповідає за окрему частину функціонування ОС - від ініціалізації апаратних механізмів і обробки переривань до роботи з пам’яттю та побудови графічного інтерфейсу поверх framebuffer.

Основні компоненти архітектури:

1. Підсистема ядра (core/kernel)

Ця частина системи відповідає за базову логіку початкової ініціалізації ОС.

У ній реалізовано:

- запуск ядра після завантаження bootloader-ом;
- встановлення глобальних структур даних;
- налаштування мінімального середовища виконання;
- підготовку системи до роботи графічних, пам’яткових та апаратних модулів.

Підсистема ізольована від конкретних драйверів і забезпечує фундамент, на якому працюють інші блоки.

2. Підсистема обробки переривань (core/interrupts)

Цей модуль реалізує:

- ***IDT (InterruptDescriptorTable);***
- обробники апаратних та програмних переривань;

- механізм обробки виключень x86_64;
- окремий обробник ***doublefault***з власним стеком;
- виведення діагностичної інформації під час критичних помилок.

Модуль має прямий доступ до низькорівневих інструкцій та містить небагато unsafe-вставок, ізольованих у добре задокументовані частини.

3. Підсистема пам’яті (memory)

Компонент відповідає за організацію та управління пам’яттю:

- ініціалізація фізичної пам’яті на основі карти пам’яті від bootloader’a;
- побудова таблиць сторінок та активація ***paging****;*
- механізм безпечного доступу до фізичної та віртуальної пам’яті;
- реалізація динамічного виділення пам’яті (heapallocator) - з можливістю підміни кількох стратегій (bump, linked-list, slab тощо).

Пам’ять ізольована від підсистеми пристроїв і може розвиватися незалежно від інших модулів.

4. Підсистема пристроїв

Цей блок містить драйвери та засоби взаємодії з апаратними компонентами:

4.1. Framebuffer

Реалізує:

- низькорівневий доступ до графічного буфера;
- відображення тексту та графічних елементів;
- примітиви малювання (пікселі, лінії, прямокутники);
- рендеринг елементів UI.

4.2. Input

Містить підтримку:

- обробки PS/2 клавіатури;
- читання raw-подій;
- можливість інтеграції з майбутньою підсистемою подій.

4.3. Mouse cursor

Надає базову логіку:

- переміщення курсору;
- рендерингу його позиції у framebuffer.

4.4. Drivers

Реалізовані драйвери: ps2_keyboard.rs обробляє скан-коди із підтримкою розширених кодів (0xE0), модифікаторів Shift/Ctrl/Alt та стрілок; ps2_mouse.rs декодує 3-байтові пакети, обчислює delta X/Y у форматі two'scomplement та нормалізує координати. Обидва використовують lock-free кільцевий буфер 256 байт з атомарними покажчиками.

5. Підсистема системних викликів (syscalls)

Інтерфейс системних викликів охоплює 18 задекларованих номерів, що обробляються через вектор IDT 120. Диспетчер dispatch_syscall() зчитує SyscallContext зі стеку переривання та маршрутизує виклик. На поточному етапі повністю функціонують 6 викликів: write, getpid, sleep, gettime, mmap, brk. Призначення підсистеми:

- визначити API взаємодії між ядром і майбутніми користувацькими програмами;
- закласти основу для розмежування привілеїв між режимами виконання.

6. Підсистема користувацького інтерфейсу

Сучасний модуль, що формує графічне середовище:

6.1. Theme, Widgets

- система стилів та кольорів;
- UI-компоненти (вікна, текстові елементи, виджети).

6.2. Байткодова віртуальна машина (vm_module)

Власна стековавіртуальна мащина з 22 типами інструкцій, включно з графічними. Підтримує стек до 1024 значень i64, 256 локальних змінних (Load/Store) та вихідний буфер 8192 байт. Метки переходів зберігаються у

BTreeMap для переходів за O(log n). Виконує анімації та обчислення без прямого доступу до ресурсів ядра.

6.3. Термінал

- рендеринг тексту та графічних примітивів у framebuffer;
- обробка команд;
- взаємодія з input-підсистемою.

7. Підсистема додатків

Цей рівень містить прикладні програми, які працюють поверх ядра:

- TerminalApp- інтерактивний термінал з 14 вбудованими командами;
- LogsApp- відображення журналу подій ядра у реальному часі;

EditorApp- текстовий редактор з підтримкою курсору та вводу.

Ця частина модульна і може бути розширена сторонніми програмами без зміни коду ядра.

8. Інфраструктурні компоненти

Скрипти збірки та запуску

Система постачається зі спеціальними інструментами, які:

- компілюють ядро Rust у формат ELF;
- конвертують його у завантажувальне образ;
- запускають ОС у QEMU;
- забезпечують автоматизоване тестування.

Тестове середовище

Дозволяє:

- запуск ядра в режимі тестів;
- відправку exit-кодів для CI;
- перевірку роботи переривань, пейджингу та UI.

3.1.1.Порівнева таблиця моделіархітектури DuxOS

Рівень	Назва	Компоненти	Відповідальність
7	Застосунки	TerminalApp, LogsApp, EditorApp	Кінцеві програми системи
6	Інтерфейс	AppHost, Theme, RenderList	Вкладки, маршрутизація подій, конвеєр рендеру
5	Байткодова ВМ	VM Runtime, BytecodeDecoder	Ізольоване виконання байткодових програм
4	Системні виклики	SyscallDispatcher (вектор 120)	API взаємодії програм з ядром
3	Драйвери	Framebuffer, PS/2 Keyboard, PS/2 Mouse	Взаємодія з апаратними пристроями
2	Управління пам'яттю	FrameAllocator, PageTables, Heap	Фізична та віртуальна пам'ять
1	Переривання	GDT, IDT, PIC 8259A, TSS	Апаратні переривання та процесорні винятки
0	Апаратне забезпечення	x86_64 CPU, BIOS, QEMU	Фізична платформа виконання системи

Порівнева модель наочно показує розподіл відповідальності між підсистемами. Кожен рівень звертається виключно до рівня нижче через строго визначений інтерфейс, що дозволяє змінювати реалізацію окремих компонентів без впливу на інші підсистеми.

3.2. Таблиця взаємодії підсистем ядра

Джерело	Ціль	Механізм взаємодії
IRQ1 (клавіатура)	PS/2 KeyboardDriver	Читання порту 0x60, кільцевий буфер 256 байт
IRQ12 (миша)	PS/2 Mouse Driver	Читання порту 0x60, 3-байтовий пакет протоколу
IRQ0 (таймер)	TIMER_TICKS	Атомарний лічильник тіків (compare-exchange)
PS/2 KeyboardDriver	AppHost	AppEvent::KeyPress з полями ch, ctrl, alt, shift, arrow
PS/2 Mouse Driver	AppHost	AppEvent::Mouse(MouseEvent) з dx, dy, buttons

Джерело	Ціль	Механізм взаємодії
AppHost	RenderList	Виклики collect_render() та collect_overlay() у кожному кадрі
RenderList	Framebuffer	Команди FillRect, StrokeRect, Text, Clear
Syscall IDT 120	Dispatcher	dispatch_syscall(SyscallContext) за номером виклику
Dispatcher	MemorySubsystem	brk(), mmap(), munmap() для керування пам'яттю програм
VM Program	Framebuffer	SetPixel, FillRect, ClearScr, Present через VM Runtime

3.3. UML Діаграма Використання (UseCaseDiagram)

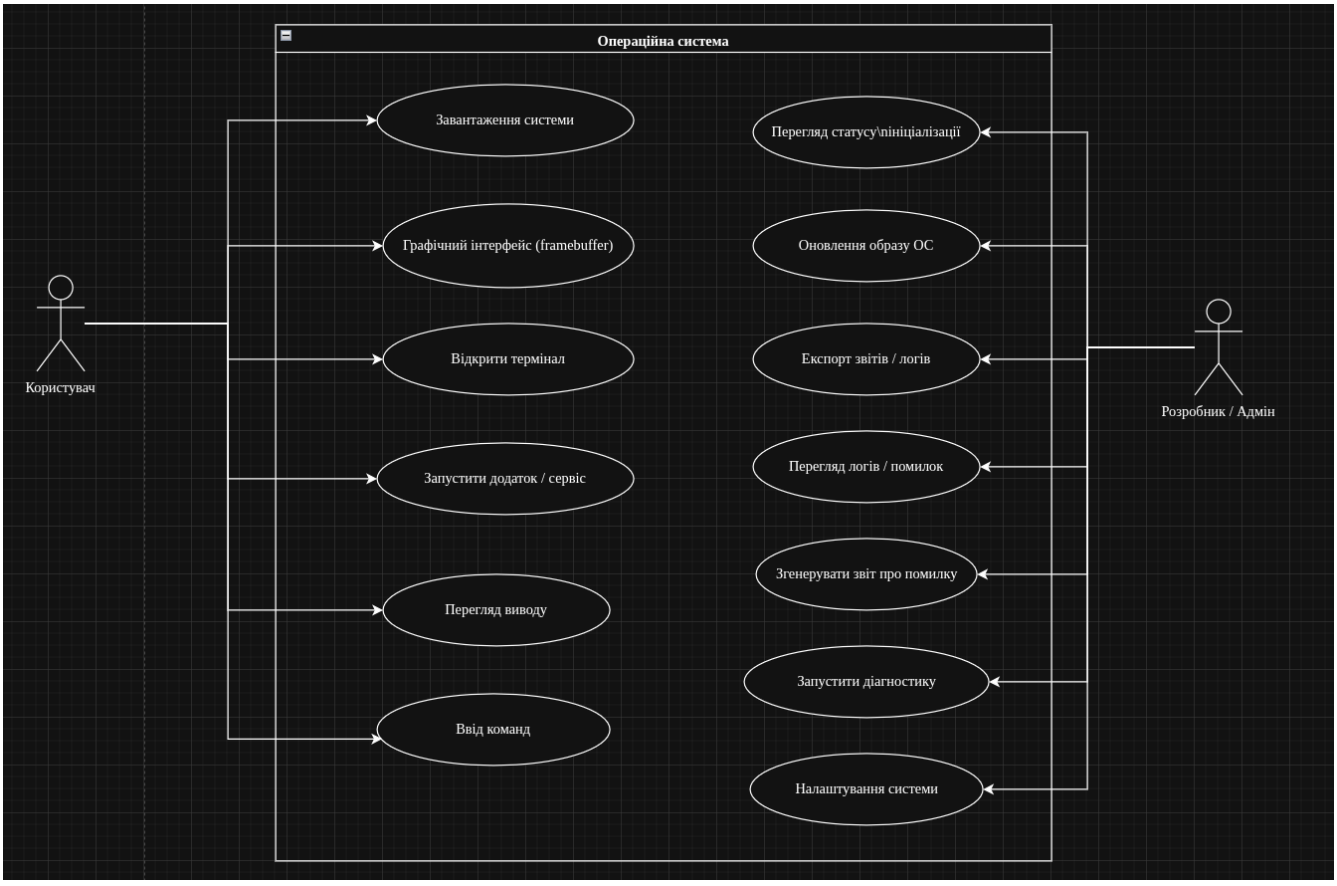

рис. 1 «*UseCaseDiagram*»

3.4 Матриця акторів і варіантів використання DuxOS

Варіант використання	**Користувач**	**Розробник / Адміністратор**	**Система**
1. Завантаження системи			X
2. Перегляд статусу ініціалізації		X	
3. Відображення графічного інтерфейсу	X	X	
4. Відкрити термінал	X	X	
5. Ввід команд у терміналі	X		
6. Перегляд виводу команд	X		
7. Запустити застосунок / сервіс	X	X	
8. Перегляд системних логів		X	
9. Перегляд помилок ядра		X	
10. Запустити діагностику		X	
11. Згенерувати звіт про помилку		X	
12. Оновлення образу ОС		X	
13. Налаштування параметрів системи		X	

У системі визначено три актори. Користувач взаємодіє з системою через термінал: вводить команди, переглядає результати та перемикається між вкладками. Розробник або адміністратор має ширші можливості: перегляд журналів ядра, запуск діагностики та аналіз помилок. Система виконує деякі дії автоматично, зокрема завантаження та ініціалізацію підсистем після передачі керування від завантажувача.

Користувач

Звичайний користувач ядра операційної системи, який взаємодіє з нею через графічний інтерфейс або термінальний застосунок. Має доступ до базових можливостей: запуск програм, введення команд, перегляд системного виводу.

Розробник / Адміністратор

Користувач із розширеними повноваженнями, що має доступ до системних журналів, діагностики, інструментів аналізу помилок і процедур оновлення операційної системи.

Можливості:

1. Завантаження системи

Описує запуск операційної системи після роботи завантажувача: ініціалізація ядра, структури пам’яті, драйверів, системних таблиць, переривань і початкових служб.

2. Перегляд статусу ініціалізації

Розробник може переглядати внутрішній стан системи під час завантаження: повідомлення ядра, помилки, інформацію про ідентифіковане обладнання та виконані системні кроки.

3. Графічний інтерфейс (framebuffer)

Відображення базового графічного середовища через framebuffer без апаратного прискорення. Забезпечує виведення вікон, елементів управління та текстової інформації.

4. Оновлення образу ОС

Розробник може завантажувати нову версію ядра або системного образу, перевіряти її цілісність та проводити операцію оновлення.

5. Відкрити термінал

Користувач отримує доступ до текстового термінального застосунку, який дозволяє вводити команди, переглядати вивід та взаємодіяти з системою на низькому рівні.

6. Експорт звітів / логів

Розробник може отримати системні журнали, дампи помилок або діагностичну інформацію для подальшого аналізу.

7. Запустити додаток / сервіс

Користувач може запускати вбудовані програми (написані для ОС) або системні задачі. Запуск здійснюється через механізм taskmanagement у ядрі.

8. Перегляд логів / помилок

Розробник отримує доступ до буфера логів, детальних звітів про винятки, помилки ядра або драйверів.

9. Перегляд виводу

Користувач переглядає результат виконання команд або роботу програм у терміналі чи графічних елементах інтерфейсу.

10. Згенерувати звіт про помилку

Система формує структуру даних, що містить опис винятку, стан стеку, регістрів, контекст задачі й іншу інформацію, важливу для налагодження.

11. Запустити діагностику

Розробник запускає спеціальні тести або перевірки системи (пам’яті, переривань, диспетчера задач, драйверів тощо).

12. Ввід команд

Користувач вводить текстові команди в терміналі; команда обробляється системним викликом та виконується відповідним сервісом ядра.

13. Налаштування системи

Адмін може змінювати конфігураційні параметри ОС: поведінку драйверів, параметри диспетчера задач, графічні налаштування тощо.

3.5. Деталі реалізації ключових підсистем

1. Система управління пам'яттю

Управління пам'яттю в DuxOS організовано у три окремих рівні: фізична пам'ять, віртуальна пам'ять та динамічна heap-пам'ять. Кожен рівень має власний модуль з чітко визначеним інтерфейсом.

Фізична пам'ять. Карта фізичної пам'яті передається завантажувачем у вигляді масиву регіонів типу MemoryRegionKind::Usable. GlobalFrameAllocator використовує атомарний покажчик NEXT_PHYSICAL_FRAME для послідовної алокації 4-KiB фреймів. Алокація виконується через CAS-цикл (compare_exchange_weak) з атомарним інкрементом, що забезпечує коректність навіть при майбутній багатоядерній конфігурації.

2. Розподіл адресного простору DuxOS

Область пам'яті	Початкова адреса	Розмір / Тип	Призначення
Legacy BIOS зона	0x0000_0000	1 MB	Зарезервовано апаратурою
Frameallocator	0x0010_0000	до 16 MB	Алокація 4-KiB фізичних фреймів
mmap регіон	0x2000_0000	Динамічний	Анонімне відображення (системний виклик mmap)
Programbreak	0x4444_4444_0000	Динамічний	Розширення купи програм (системний виклик brk)
Kernelheap	BSS-сегмент ядра	256 MB статичний буфер	FixedSizeBlockAllocator для ядра
Kernelcode / data	Визначено bootloader	Розмір ELF-образу	Завантажений завантажувачем

Сторінкова адресація. Реалізовано 4-рівневу ієрархію таблиць сторінок (P4, P3, P2, P1) відповідно до специфікації x86_64. Функція map_single_page() приймає віртуальну адресу та номер фізичного фрейму, обходить чотири рівні таблиць і встановлює прапори доступу. Проміжні таблиці створюються автоматично при потребі. Прапор NO_EXECUTE встановлюється для сторінок даних за замовчуванням і знімається лише для виконуваних сторінок.

Heap-алокатор. Порівняння реалізованих алгоритмів наведено у таблиці 3.5.

3. Порівняння алгоритмів heap-алокатора

Алгоритм	Alloc	Dealloc	Фрагментація	Область застосування
BumpAllocator	O(1)	N/A	Висока	Ніколи не деалокує; тільки ініціалізація
LinkedListAllocator	O(n)	O(n)	Середня (злиття	Fallback для

Алгоритм	Alloc	Dealloc	Фрагментація	Область застосування
			блоків)	блоків > 2048 байт
FixedSizeBlockAllocator	O(1)	O(1)	Низька (фіксовані класи)	Основний алокатор ядра

Основним алокатором є FixedSizeBlockAllocator з класами блоків: 8, 16, 32, 64, 128, 256, 512, 1024, 2048 байт. Кожен клас підтримує власний зв'язаний список вільних блоків. Для запитів понад 2048 байт або нестандартних розмірів використовується LinkedListAllocator як fallback з операцією злиття суміжних вільних блоків.

4. Система обробки переривань

Таблиця дескрипторів переривань (IDT) налаштована для 256 векторів. PIC 8259A ремапований: IRQ0-7 отримують вектори 32-39, IRQ8-15 отримують вектори 40-47. Вектор 120 зарезервовано для програмних системних викликів.

5. Ключові вектори переривань та їх обробники

Вектор	Назва виключення / переривання	Обробник	Час EOI
0	DivideError	divide_error_handler	N/A
3	Breakpoint	breakpoint_handler	N/A
6	InvalidOpcode	invalid_opcode_handler	N/A
8	DoubleFault	double_fault_handler (IST0, стек 4 KB)	N/A
13	GeneralProtectionFault	gpf_handler	N/A
14	PageFault	page_fault_handler (аналіз CR2)	N/A
32 (IRQ0)	Timer PIT	timer_interrupt_handler	Після обробника
33 (IRQ1)	PS/2 Keyboard	keyboard_interrupt_handler	До обробника

Вектор	Назва виключення / переривання	Обробник	Час EOI
44 (IRQ12)	PS/2 Mouse	mouse_interrupt_handler	Після обробника
120	Syscall	syscall_handler	До обробника

DoubleFaulthandler виконується на окремому стеку розміром 4096 байт, індекс якого зазначено у InterruptStackTable (IST) TSS. Це гарантує коректну роботу обробника навіть при переповненні основного стеку ядра і виключає перехід у некерований стан triplefault. Час надсилання End-of-Interrupt (EOI) відрізняється залежно від обробника: для клавіатури та системних викликів EOI надсилається до виконання логіки, для таймера та миші - після.

6. Віртуальна машина

DuxOS включає власну стековубайткодовувіртуальну машину. Вона виконує програми, що представлені як вектор інструкцій з мітками переходів, та підтримує арифметичні обчислення, керуючі конструкції та графічні операції через framebuffer.

7. Система інструкцій віртуальної машини

Категорія	Інструкція	Опис
Стек	Push(i64)	Помістити 64-бітну константу на стек
Стек	Dup	Дублювати вершину стеку
Стек	Swap	Поміняти місцями два верхні елементи
Стек	Drop	Видалити вершину стеку
Арифметика	Add, Sub, Mul, Div, Mod	Бінарні арифметичні операції над i64
Арифметика	Neg	Унарне заперечення вершини стеку
Порівняння	Eq, Neq, Gt, Lt	Логічне порівняння; результат 1 або 0
Керування	Jmp(label)	Безумовний перехід за міткою програми
Керування	Jz(label), Jnz(label)	Умовні переходи за вершиною стеку
Керування	Halt	Зупинка виконання програми
Пам'ять	Load(u8), Store(u8)	Читання/запис локальних змінних (256

Категорія	Інструкція	Опис
		слотів)
Виведення	Print	Вивести вершину стеку як ціле число
Графіка	SetPixel	Намалювати піксель (x, y, color)
Графіка	FillRect	Заповнити прямокутник (x, y, w, h, color)
Графіка	ClearScr	Очистити весь екран
Графіка	Present	Передати тіньовий буфер у відеопам'ять

Обмеження середовища виконання: стек на 1024 елементи типу i64, 256 слотів локальних змінних та вихідний буфер розміром 8192 байт. Для міток переходів програма зберігає BTreeMap від рядка до індексу інструкції, що дозволяє виконувати переходи за O(log n). Вбудовані приклади демонструють: підрахунок суми ряду та анімацію відбивного прямокутника на framebuffer.

Графічна підсистема та фреймбуфер

Framebuffer надається bootloader-ом і являє собою масив пікселів у форматі BGR 32-bit (4 байти на піксель). DuxOS реалізує тіньовий буфер (shadowbuffer) в оперативній пам'яті ядра для мінімізації запису у відеопам'ять.

Ключові оптимізації рендеру:

- Тіньовий буфер: усі операції малювання виконуються у RAM-копії. У VRAM передається лише змінена частина.
- Блочне відстеження змін: екран розбито на тайли 32x32 пікселі. Тайл позначається як змінений при будь-якій модифікації пікселя всередині.
- Хешування рядків: для кожного рядка пікселів обчислюється хеш. Рядок передається у VRAM тільки якщо його хеш змінився відносно попереднього кадру.

Конвеєр рендеру реалізований через RenderList: кожен застосунок формує список команд (Clear, FillRect, FillRoundedRect, StrokeRect, Text), після чого AppHost виконує їх пакетно на FramebufferWriter. Теми оформлення (Theme)

визначають палітру кольорів через структуру з полями text, background, accent, surface, border, muted, on_accent.

Фреймворк застосунків та системні виклики

AppHost керує трьома вкладками: Terminal (F1), Logs (F2), Editor (F3). Кожна вкладка містить один застосунок, що реалізує трейтApp.

ТрейтApp визначає контракт застосунку:

init() - ініціалізація початкового стану застосунку.

on_event(AppEvent) ->bool - обробка подій (натискання клавіш, тік таймера, рух миші).

layout(Rect) - розрахунок меж розміщення на екрані.

collect_render(theme, out) - формування списку команд рендеру.

collect_overlay(theme, out) - рендер елементів поверх основного вмісту.

focus_blocks() - список блоків фокусу для навігації клавіатурою.

Переключення між вкладками виконується клавішами F1, F2, F3 або Alt+Tab. Клавіші Alt+1 .. Alt+9 дозволяють прямий перехід до конкретної вкладки. Миша підтримується через AppEvent::Mouse з координатами delta X, delta Y та станом трьох кнопок (ліва, права, середня). TerminalApp підтримує команди: help, test, test_paging, test_memory, test_asm, vm_help, vm_demo, vm_run, echo, info, clear, exit.

3.6. Текстовий асемблер та байткодовий парсерВМ

Окрім виконання вбудованих програм, поданих безпосередньо у вигляді вектора інструкцій, віртуальна машина DuxOS підтримує запуск програм, написаних користувачем у текстовій формі. Така можливість реалізована командою терміналу vm_run, яка приймає вихідний код у мнемонічному поданні (інструкції розділяються символом «;») та виконує його. Це перетворює віртуальну машину з простого інтерпретатора фіксованих прикладів на

повноцінне середовище для написання й налагодження невеликих програм безпосередньо у просторі ядра.

Конвеєр обробки користувацької програми складається з трьох послідовних етапів. На першому етапі функція parse_program() виконує лексичний та синтаксичний розбір тексту й будує структуру Program, що містить вектор інструкцій та таблицю міток. На другому етапі сформований байткод передається у середовище виконання Vm, де відбувається його послідовна інтерпретація. На третьому етапі результат виконання (числовий вивід або графічна побудова) передається у підсистему рендеру або у термінальний застосунок.

Парсер реалізовано за схемою двох проходів. Під час першого проходу аналізуються рядки, що завершуються двокрапкою, вони трактуються як визначення міток і заносяться до таблиці переходів (BTreeMap від рядка до індексу інструкції). Під час другого проходу кожен рядок розпізнається як мнемоніка інструкції з відповідними аргументами, а посилання на мітки замінюються перевіреними індексами. Така модель дозволяє використовувати мітки до їх оголошення у тексті програми (forwardreference), що є типовим для циклів та умовних переходів.

Важливою властивістю парсера є детальна діагностика помилок. Замість загального повідомлення про невдале опрацювання парсер повертає типізовану помилку ParseError з зазначенням номера рядка та контексту, що суттєво спрощує налагодження.

Опис ситуації	Контекст	Код помилки
Вхідна програма не містить жодної інструкції	-	EmptyProgram
Повторне оголошення мітки з однаковим ім’ям	рядок, мітка	DuplicateLabel
Ім’я мітки не відповідає правилам іменування	рядок, мітка	InvalidLabel
Невідома мнемоніка інструкції	рядок, мнемоніка	UnknownInstruction
Інструкція потребує аргумент, який	рядок, мнемоніка	MissingArgument

відсутній		
Передано більше аргументів, ніж очікувалось	рядок, мнемоніка	TooManyArguments
Аргумент не є коректним цілим числом i64	рядок, значення	InvalidInteger
Некоректна ціль переходу	рядок, ціль	InvalidTarget
Перехід на неоголошену мітку	рядок, мітка	UnknownLabel

Таблиця 3.6 - Категорії синтаксичних помилок парсера віртуальної машини

Завдяки такому підходу помилка у користувацькій програмі не призводить до аварійного завершення ядра: парсеркоректно повертає опис проблеми, який виводиться у термінал, а середовище виконання навіть не запускається. Це є прямим наслідком використання bubble-upпатернів роботи з помилками у мові програмування Rust, що змушує обробляти всі гілки помилок на етапі компіляції самого ядра.

3.7. Модель ізольованого виконання VM-програм (VmProcess)

Для запуску байткодових програм у контрольованому середовищі реалізовано абстракцію VmProcess, що моделює мінімальний «процес» віртуальної машини. Кожен такий процес отримує власний ідентифікатор (PID), власну ділянку пам’яті (арену) та ізольований екземпляр середовища виконання Vm. Ідентифікатори процесів видаються атомарним лічильником NEXT_VM_PID, початкове значення якого встановлено на 1000, що дозволяє відрізняти VM-процеси від системних ідентифікаторів ядра.

Під час створення процесу методом VmProcess::create() виконується системний виклик mmap, який виділяє арену розміром у п’ять сторінок (20 КіБ) з правами читання та запису (PROT_READ | PROT_WRITE), але без права виконання. У межах цієї арени розміщується структура Vm разом зі стеком значень, масивом локальних змінних та вихідним буфером. Таким чином, уся

робоча пам’ять програми відокремлена від службових структур ядра й отримується через ту саму підсистему керування пам’яттю, що й інші динамічні алокації.

Ключовою властивістю цієї моделі є те, що байткодова програма принципово не має прямого доступу до пам’яті або апаратних ресурсів ядра. Усі дозволені дії обмежені набором інструкцій віртуальної машини, а будь-яка взаємодія із зовнішнім світом (наприклад, побудова зображення) відбувається лише через чітко визначені графічні інструкції SetPixel, FillRect, ClearScr та Present. Це фактично реалізує форму «пісочниці» (sandbox): навіть навмисно некоректна програма не здатна пошкодити стан ядра, оскільки помилки виконання повертаються як типізоване значення VmError (варіанти Parse, Runtime та RuntimeN із зазначенням позиції), а не призводять до неконтрольованого доступу до пам’яті.

Після завершення роботи арена процесу звільняється, а зайняті фізичні фрейми повертаються до системи. Така модель є природним підґрунтям для подальшого розвитку повноцінної підсистеми процесів user-space: механізм виділення ізольованої арени, призначення PID та контрольованого набору дозволених операцій безпосередньо узгоджується з майбутньою реалізацією системних викликів fork та exec.

3.8. Виконання машинного коду та політика W^X

Окремий дослідницький інтерес становить підсистема AsmExecutor, що демонструє виконання довільного нативного машинного коду x86_64 у середовищі ядра. На відміну від байткодової віртуальної машини, яка інтерпретує інструкції програмно, AsmExecutor виконує реальні машинні інструкції безпосередньо процесором, що наочно ілюструє взаємодію між підсистемою керування пам’яттю та політикою захисту виконання.

Процес виконання організовано в кілька кроків. Спочатку перевіряється коректність вхідного буфера: код не може бути порожнім і не повинен перевищувати обмеження MAX_CODE_SIZE (4096 байт). Далі через системний виклик mmap виділяється сторінкововирівняна ділянка пам’яті з правами читання, запису та виконання (PROT_READ | PROT_WRITE | PROT_EXEC). Машинний код копіюється у цю ділянку, після чого вказівник на її початок приводиться (transmute) до типу функції extern "C" fn() -> u64 і викликається. Повернене значення інтерпретується як 64-бітний результат, а ділянка пам’яті звільняється викликом munmap.

Принципово важливим є той факт, що спроба виконати код у звичайній динамічній пам’яті (heap) завідомо неможлива. Сторінки купи позначаються прапором NO_EXECUTE (NX-біт) під час побудови таблиць сторінок, тому передача керування на адресу в межах купи спричинила б виняток захисту сторінки. Саме тому виконуваний код обов’язково розміщується у спеціально виділеній через mmap ділянці з явно встановленим правом виконання. Це є практичною реалізацією принципу «запис або виконання, але не одночасно» (W^X, Write XOR Execute) - однієї з базових вимог безпеки, сформульованих у розділі нефункціональних вимог.

Для верифікації механізму використано набір мінімальних еталонних програм, поданих безпосередньо у машинному коді.

Очікуваний результат	Машинний код (x86_64)	Програма
Повертає значення 42	mov eax, 42; ret	simple_return_42
Повертає значення 3	mov eax,1; mov ecx,2; add eax,ecx; ret	simple_add_1_2
Повертає задане 64-бітне значення v	mov rax, v; ret (генерується)	return_argument(v)

Таблиця 3.7 - Програми для перевірки виконання нативного коду

Перевірка цих програм підтверджує коректність повного ланцюга «виділення виконуваної пам’яті- копіювання коду - передача керування - повернення результату - звільнення пам’яті» та демонструє узгоджену роботу підсистем mmap, munmap і сторінкової адресації з урахуванням прапора NX.

3.9. Система діагностики ядра

Для забезпечення спостережуваності роботи ядра реалізовано окрему систему структурованого логування- конвеєр діагностичних подій. На відміну від простого виведення тексту у серійний порт, ця підсистема накопичує події у ring-buffer фіксованої місткості (за замовчуванням 512 записів) та надає їх застосунку перегляду журналу LogsApp у реальному часі.

Кожна діагностична подія DebugEvent містить чотири складові: рівень важливості, категорію джерела, статичний рядок-ідентифікатор джерела та текст повідомлення. Додатково кожній події присвоюється порядковий номер, що дозволяє однозначно впорядкувати події навіть у разі їх надходження з обробників переривань. Передбачено чотири рівні діагностичних подій -Debug, Info, Warn та Error.Кожному з яких відповідає власний колір відображення у журналі, що покращує візуальне сприйняття діагностики.

Категоризація подій за джерелом дозволяє фільтрувати журнал за конкретною підсистемою під час налагодження.

Підсистема-джерело події	Категорія
Загальні події, що не належать до конкретної підсистеми	General
Ініціалізація та базова логіка ядра	Kernel
Керування пам’яттю: алокатори, сторінки, фрейми	Memory
Обробка переривань і процесорних винятків	Interrupts
Драйвери клавіатури та миші PS/2	Input
Фреймворк застосунків та маршрутизація подій	App
Графічна підсистема та конвеєр рендеру	Render

Байткодова віртуальна машина та її процеси	Vm
Диспетчер системних викликів	Syscall

Таблиця 3.8 - Категорії діагностичних подій ядра

Доступ до підсистеми надається через набір макросів log_debug!, log_info!, log_warn! та log_error!, що мають інтерфейс, подібний до стандартного форматованого виведення. Завдяки цьому інструментування коду новими діагностичними точками не вимагає звернення до внутрішньої реалізації буфера, а самі повідомлення формуються лінивим способом і потрапляють як у журнал LogsApp, так і у серійний порт середовища QEMU. Така архітектура дозволяє відокремити місце генерації події від місця її відображення та зберігання, що відповідає вимозі модульності, висунутій у розділі нефункціональних вимог.

3.10. Розширений набір алгоритмів розподілу пам’яті

У підрозділі 3.5 розглянуто три основні алгоритми heap-алокатора, що безпосередньо задіяні у роботі ядра. Проте кодова база містить ширший набір реалізацій, які чітко поділяються на дві групи: активні алокатори, що використовуються під час роботи системи, та резервні алокатори, підготовлені для майбутніх сценаріїв і дослідницьких порівнянь. Такий поділ є практичною реалізацією вимоги розширюваності: механізм розподілу пам’яті дозволяє заміну реалізації без зміни логіки роботи ядра, оскільки всі алокатори реалізують спільний інтерфейс GlobalAlloc.

Призначення та особливості	Статус	Алгоритм
Основний алокатор ядра; розподіл за класами розмірів O(1)	Активний	FixedSizeBlockAllocator
Резерв для великих блоків; злиття суміжних вільних ділянок	Активний	LinkedListAllocator
Найшвидший розподіл O(1) без звільнення; ініціалізація та тимчасові буфери	Резервний	BumpAllocator
LIFO-розподіл; звільнення у зворотному	Резервний	StackAllocator

порядку; атомарна вершина		
Кеш-вирівняний розподіл об’єктів фіксованого типу; параметризація розміром і вирівнюванням через узагальнені константи	Резервний	SlabAllocator
Розділення регіонів стеку та купи у межах однієї арени	Резервний	StackHeapAllocator

Таблиця 3.9 - Перелік реалізованих алгоритмів розподілу пам’яті

Резервні алокатори демонструють різні стратегії компромісу між швидкістю розподілу, можливістю звільнення та рівнем фрагментації. Зокрема, SlabAllocator реалізовано з використанням узагальнених констант (constgenerics) мови Rust, що дозволяє на етапі компіляції спеціалізувати алокатор під конкретний розмір і вирівнювання об’єкта(layout), повністю усуваючи накладні витрати на обчислення розміру під час виконання. Усі реалізації є безпечними для використання в потоках завдяки застосуванню атомарних операцій або спінлоків, що закладає основу для майбутньої багатоядерної конфігурації.

3.11. Система просторової навігації фокусом

Графічне середовище DuxOS підтримує навігацію елементами інтерфейсу не лише за допомогою мишки, а й з клавіатури. Для цього реалізовано систему навігації фокусом, що дозволяє переміщувати активний фокус між інтерактивними блоками (FocusBlock) клавішами-стрілками відповідно до їх фактичного розташування на екрані.

Алгоритм зміни фокусу, реалізований у функції move_focus(). Для поточного активного блоку обчислюється його геометричний центр, після чого для кожного елемента визначаються зміщення dx та dy відносно цього центру. Блокважається припустимою ціллю лише тоді, коли його розташування узгоджується з зворотнім напрямком: для руху вгору вимагається dy< 0 за умови, що горизонтальне відхилення не перевищує вертикального; аналогічні умови симетрично застосовуються для решти напрямків. Серед усіх припустимих

елементів обирається той, що має найменшу сумарну відстаньтаким чином фокус переходить до найближчого елемента у заданому напрямку.

Поточний активний елемент візуально виділяється рамкою фокуса завширшки в один піксель, яку малює функція draw_focus_ring(). Рамка дозволяє користувачевібистровизначити поточну позицію фокуса під час навігації без використання мишки. Описана підсистема є важливою з точки зору ергономіки та доступності інтерфейсу, оскільки забезпечує повноцінне керування системою у середовищі, де вказівний пристрій може бути недоступним або небажаним.

Реалізовані системні виклики

Номер	Назва	Категорія	Статус	Опис
0	read	I/O	Заглушка	Читання з stdin; повертає ENOSYS
1	write	I/O	Реалізовано	Запис у серійний порт/термінал (FD 1, 2)
2	open	I/O	Заглушка	Відкриття файлу; повертає ENOSYS
3	close	I/O	Заглушка	Закриття дескриптора; повертає ENOSYS
20	exit	Процес	Заглушка	Нескінченний spin-loop; справжнє завершення не реалізоване
21	fork	Процес	Частково	Виділяє таблицю сторінок дочірнього процесу, PID повертається
22	exec	Процес	Частково	Передає керування новому образу через sys_pstart
23	wait	Процес	Заглушка	Нескінченний spin-loop; очікування не реалізоване
24	getpid	Процес	Реалізовано	Повертає CURRENT_PID (або 1 для ядра)
40	mmap	Пам'ять	Реалізовано	Відображення сторінок через GlobalFrameAllocator
41	munmap	Пам'ять	Заглушка	Аргументи перевіряються, але сторінки не звільняються
42	brk	Пам'ять	Реалізовано	Розширення heap через sys_brk

Номер	Назва	Категорія	Статус	Опис
60	sleep	Час	Реалізовано	Активне очікування на основі TIMER_TICKS
61	gettime	Час	Реалізовано	Повертає ticks * 55 мс (PIT ~18.2 Гц)
80	kill	Сигнал	Не реалізовано	Немає обробника; повертає ENOSYS
81	signal	Сигнал	Не реалізовано	Немає обробника; повертає ENOSYS
100	chdir	Файлова с.	Не реалізовано	Присутній лише як значення enumSyscallNumber
101	mkdir	Файлова с.	Не реалізовано	Присутній лише як значення enumSyscallNumber

Диспетчер системних викликів отримує контекст через SyscallContext, зчитуючи аргументи з регістрів rdi, rsi, rdx, r10, r8, r9. Помилки повертаються як від'ємні значення у стилі POSIX: EINVAL (-22), EACCES (-13), ENOSYS (-38), EBADF (-9), ENOMEM (-12), EIO (-5). Виклики, для яких обробник ще не написано, повертають ENOSYS автоматично через гілку за замовчуванням у dispatch_syscall().

ВИСНОВКИ

У процесі розробки операційної системи було проведено комплексне дослідження архітектури низькорівневих компонентів, механізмів апаратної взаємодії та принципів організації базових підсистем, що формують основу функціонування будь-якого сучасного ядра. Робота охоплювала повний цикл створення системи: від етапу завантаження та переходу в режим x86_64 longmode до побудови власних механізмів керування пам’яттю, обробки винятків, переривань і виведення графічної інформації.

У рамках дослідження було визначено та реалізовано ключові підсистеми, необхідні для працездатності ОС на ранньому етапі розвитку. Зокрема, було розроблено підсистему ініціалізації, яка забезпечує коректне отримання керування від завантажувача, налаштування GDT та IDT, а також передачу структур framebuffer і карти пам’яті. Значна увага приділялась організації графічного виводу, що дозволило реалізувати framebuffer-консоль, базові графічні примітиви та механізм відображення тексту, необхідний для діагностики та налагодження.

Особливе місце у проєкті займала підтримка механізмів обробки критичних помилок та апаратних винятків. Реалізація panic-handler та окремих обробників для стандартних винятків x86_64 забезпечила можливість стабільного функціонування системи навіть у разі фатальних ситуацій. Створення коректної таблиці IDT, а також впровадження DoubleFaulthandler на окремому стеку надало системі базову стійкість та убезпечило від переходу у некерований стан (triplefault).

У ході роботи було сформовано підсистему обробки переривань, включаючи підтримку PIC 8259A, таймера PIT (IRQ0), клавіатури PS/2 (IRQ1) та миші PS/2 (IRQ12). Це створило умови для повноцінної взаємодії з апаратними пристроями та реалізації подієвої моделі вводу. Додатково було реалізовано диспетчер системних викликів з 18 обробниками, що закладає основу для майбутнього user-space.

Паралельно було організовано підсистему тестування, що включає юніт-тести в оточенні no_std, інтеграційні тести в QEMU та механізми виведення результатів. Значну увагу отримала автоматизація збірки: створено скрипти для генерації target, побудови ядра, формування ISO/IMG-образів та автоматичного запуску в QEMU.

Таким чином, у процесі роботи було сформовано фундаментальне ядро операційної системи з повним набором базових компонентів, необхідних для подальшої еволюції платформи. Розроблені підсистеми створюють основу для подальшого розширення: додавання файлової системи, драйверів пристроїв, планувальника задач, мережевих стеків та графічного інтерфейсу вищого рівня.

Проведене дослідження засвідчило, що побудова власної ОС - це складний, багатогранний процес, який вимагає глибокого розуміння як апаратних особливостей, так і загальних принципів системного програмування. Одержані результати підтверджують можливість створення повноцінної модульної операційної системи, що може слугувати основою для подальших експериментів, розширень та практичного застосування.

Окремою вагомою складовою реалізації стала байткодова віртуальна машина, що демонструє можливість виконання ізольованих програм безпосередньо у просторі ядра. ВМ підтримує 22 типи інструкцій, включно з графічними операціями через framebuffer, що дозволяє виконувати прості анімації та обчислення без прямого доступу до ресурсів ядра.

Реалізований фреймворк застосунків з тіньовим буфером, тайловим відстеженням змін та хешуванням рядків суттєво знижує кількість операцій запису у відеопам'ять порівняно з прямим рендером. Три вбудовані застосунки (термінал, журнал логів, текстовий редактор) демонструють роботу фреймворку і слугують основою для подальшого розширення набору програм системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Klabnik S., Nichols C. The Rust Programming Language. URL: https://doc.rust-lang.org/book/ (дата звернення: 17.06.2026).
2. Rust Embedded Working Group. The Embedded Rust Book. URL: https://docs.rust-embedded.org/book/ (дата звернення: 17.06.2026).
3. Bare Metal Rust. URL: https://docs.rust-embedded.org/ (дата звернення: 17.06.2026).
4. QEMU Emulator Documentation. URL: https://www.qemu.org/docs/ (дата звернення: 17.06.2026).
5. Bos M. Rust Atomics and Locks: Low-Level Concurrency in Practice. Sebastopol : O'Reilly Media, 2023. URL: https://www.oreilly.com/library/view/rust-atomics-and/9781098119430/ (дата звернення: 17.06.2026).
6. Blandy J., Orendorff J., Tindall L. Programming Rust: Fast, Safe Systems Development. 2nd ed. Sebastopol : O'Reilly Media, 2021. URL: https://www.oreilly.com/library/view/programming-rust-2nd/9781492052586/ (дата звернення: 17.06.2026).
7. Gjengset J. Rust for Rustaceans: Idiomatic Programming for Experienced Developers. San Francisco : No Starch Press, 2021. URL: https://rust-for-rustaceans.com/ (дата звернення: 17.06.2026).
8. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Boston : Pearson, 2014. 1136 p.
9. Карпин Д., Карпин А., Гарбич-Мошора О., Вежделл А. Методи проєктування та реалізації безпечного ядра операційної системи на Rust у середовищі no_std для архітектури x86_64. Наука і техніка сьогодні. 2026. № 5(59). С. 4572–4585.
10. Вежделл А., Карпин Д. Розробка ядра операційної системи. Актуальні проблеми сучасної науки : матеріали XIII Міжнародної науково-практичної

конференції студентів та викладачів факультету фізики, математики, економіки та інноваційних технологій, м. Дрогобич, 27–30 квітня 2026 р. Дрогобич : Редакційно-видавничий відділ ДДПУ імені Івана Франка, 2026. С. 60–61.