

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

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка ядра Unix-подібної ОС з можливістю створення драйверів на WebAssembly

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

Студент гр. ППЗ-22-1

_____ /Н. Б. Нікулін /

Керівник

кваліфікаційної роботи

/ Д. В. Швець /

Завідувач кафедри

/ А. М. Стрюк /

Кривий Ріг
2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

_____ А. М. Стрюк

«___» _____ 20__ р.

ЗАВДАННЯ на кваліфікаційну роботу

студенту групи ІПЗ-22-1 Нікуліну Назару Богдановичу

1. Тема: Розробка ядра Unix-подібної ОС з можливістю створення драйверів на WebAssembly, затверджено наказом КНУ № 284 від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «2» червня 2026р.
3. Вихідні дані по роботі: розроблена система повинна забезпечувати витісняючу багатозадачність з ізольованими процесами, підтримувати завантаження та виконання драйверів пристроїв у вигляді WebAssembly-модулів з JIT-компіляцією
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): проаналізувати існуючі архітектури операційних систем та підходи до ізоляції драйверів; обґрунтувати вибір WebAssembly як середовища виконання драйверів; спроектувати архітектуру ядра з підсистемами управління пам'яттю, процесами, VFS та Wasm-рантайму

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	05.01.26 – 20.01.26
2	Аналіз існуючих методів вирішення проблеми	21.01.26 – 04.02.26
3	Формулювання актуальності роботи і постановка завдань	05.02.26 – 15.02.26
4	Оформлення матеріалів першого розділу роботи	16.02.26 – 01.03.26
5	Створення алгоритму додатку	02.03.26 – 20.03.26
6	Оформлення матеріалів другого розділу роботи	21.04.26 – 11.04.26
7	Розробка програмних модулів	12.04.26 – 01.05.26
8	Оформлення додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 30.05.26

Дата видачі завдання: «04» січня 2026 р.

Студент _____ / Н. Б. Нікулін /

Керівник роботи _____ / Д. В. Швець /

РЕФЕРАТ

Розробка ядра Unix-подібної ОС з можливістю створення драйверів на WebAssembly

ОПЕРАЦІЙНА СИСТЕМА, ЯДРО ОС, WEBASSEMBLY, ДРАЙВЕР ПРИСТРОЮ, ІЗОЛЯЦІЯ КОМПОНЕНТІВ, ЛІТ-КОМПІЛЯЦІЯ, ФАЙЛОВА СИСТЕМА EXT2, МОВА ПРОГРАМУВАННЯ ODIN.

Пояснювальна записка: 98 ст., 3 рис., 7 табл., 17 джерел.

У кваліфікаційній роботі розроблено ядро Unix-подібної ОС з підтримкою драйверів пристроїв у вигляді WebAssembly-модулів.

Проведено огляд архітектур ОС: монолітного ядра, мікроядра та підходів на основі SFI й WebAssembly. Обґрунтовано доцільність застосування WebAssembly як ізольованого середовища виконання для драйверів у монолітному ядрі.

Реалізовано підсистеми керування процесами (Round-Robin), пам'яттю, VFS, сигналів та системних викликів. Підсистема WebAssembly виконує ЛІТ-компіляцію модулів у нативний x86-64 код і надає host-функції для взаємодії з ядром. Реалізацію виконано мовою Odin.

Портовано mlibc, скомпільовано sbase та oksh. Порівняльне тестування драйвера ext2 показало накладні витрати Wasm-рантайму на рівні 27% для RAM-блокового пристрою.

ABSTRACT

Development of a Unix-like Operating System Kernel with Support for Creating Drivers in WebAssembly

OPERATING SYSTEM, OS KERNEL, WEBASSEMBLY, DEVICE DRIVER, COMPONENT ISOLATION, JIT COMPILATION, EXT2 FILESYSTEM, ODIN PROGRAMMING LANGUAGE.

Explanatory note: 98 pages, 3 figures, 7 tables, 17 sources.

This work presents the design and implementation of a Unix-like OS kernel with support for device drivers as WebAssembly modules.

Existing OS architectures were reviewed: monolithic kernel, microkernel, and SFI/WebAssembly-based isolation. The use of WebAssembly as an isolated execution environment for drivers inside a monolithic kernel is justified.

Process management (Round-Robin), memory, VFS, signals, and syscall subsystems were implemented. The WebAssembly subsystem JIT-compiles modules to native x86-64 and exposes host functions for kernel interaction. The implementation is written in Odin.

The mlibc C library was ported, sbase utilities and oksh shell were compiled and run. Comparative benchmarking of the ext2 driver showed a 27% WebAssembly runtime overhead on a RAM block device.

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	9
1.1 Огляд існуючих Unix-подібних операційних систем	9
1.2 Архітектури ядер операційних систем.....	10
1.3 Проблеми класичних підходів до розробки драйверів.....	12
1.4 Огляд WebAssembly як технологічної основи	13
1.5 Висновки до розділу.....	13
РОЗДІЛ 2 ПРОЄКТУВАННЯ СИСТЕМИ.....	15
2.1 Вимоги до системи	15
2.2 Підсистема управління пам'яттю.....	17
2.3 Початковий образ файлової системи (initrd).....	17
2.4 Підсистема процесів.....	18
2.5 Підсистема Wasm-драйверів.....	18
2.6 VFS та системні виклики	19
2.7 Проєктування Wasm-драйвера файлової системи ext2	20
2.8 Висновки до розділу.....	21
РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ.....	22
3.1 Технологічний стек	22
3.2 Послідовність завантаження системи	23
3.3 Реалізація підсистеми управління пам'яттю	24
3.4 Реалізація підсистеми процесів	25
3.5 Реалізація підсистеми Wasm-драйверів	34
3.6 Реалізація віртуальної файлової системи.....	38
3.7 Реалізовані Wasm-драйвери.....	40
РОЗДІЛ 4 ТЕСТУВАННЯ СИСТЕМИ ТА ОЦІНКА РЕЗУЛЬТАТІВ.....	47
4.1 Формування мінімального користувацького оточення.....	47
4.2 Розробка нативного драйвера файлової системи ext2.....	49
4.3 Програма порівняльного тестування.....	50

	7
4.4 Результати та аналіз	51
4.5 Висновки до розділу.....	52
ВИСНОВКИ	54
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	56
Додаток А	58
Додаток Б.....	65
Додаток В.....	76
Додаток Г.....	89
Додаток Д	90

ВСТУП

У монолітних ядрах драйвери пристроїв виконуються у привілейованому режимі поряд із ядром, тому будь-яка помилка в їхньому коді може призвести до збою всієї системи. Мікроядра вирішують цю проблему ціною суттєвих накладних витрат на міжпроцесну комунікацію. Таким чином, існує потреба у підході, що поєднує швидкодію монолітної архітектури з ізоляцією компонентів.

У рамках даної кваліфікаційної роботи буде розроблено ядро Unix-подібної операційної системи, в якому драйвери пристроїв виконуватимуться у вигляді WebAssembly-модулів. Буде реалізовано базові підсистеми ядра - управління пам'яттю, планування процесів, VFS, сигнали та системні виклики - а також однопрохідний JIT-компілятор WebAssembly у x86-64 машинний код. На основі розробленої підсистеми буде написано драйвери клавіатури, терміналу та файлової системи ext2.

Для підтвердження практичної придатності системи буде портовано стандартну бібліотеку C та скомпільовано набір базових Unix-утиліт. Порівняльне тестування Wasm-драйвера ext2 відносно нативної реалізації дозволить кількісно оцінити накладні витрати запропонованого підходу.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд існуючих Unix-подібних операційних систем

Unix-подібні операційні системи міцно закріпилися в сучасній інфраструктурі. Більшість серверів у світі - понад 96% - працюють на Linux або його родичах [1].

1.1.1 Linux

Linux - монолітне відкрите ядро, яке Лінус Торвальдс опублікував у 1991 році. Зараз його можна зустріти всюди: сервери, смартфони на Android, вбудовані пристрої, суперкомп'ютери.

Вся суть архітектури Linux у тому, що все ядро - планувальник, драйвери, файлові системи, мережі - працює разом, у єдиному привілейованому просторі (kernel space). Підсистеми безпосередньо викликають функції одна одної. Ніяких бар'єрів, ніякої справжньої ізоляції.

Плюс такого підходу очевидний: мінімальні затримки, максимальна продуктивність.

Але оскільки драйвери мають такі ж права, як і саме ядро, кожна помилка чи вразливість у драйвері - це вже потенційна паніка всієї системи або й шлях до захоплення контролю. Статистика: дві третини критичних багів ядра Linux пов'язані саме із завантажуваними модулями [2], а драйвери - на першому місці [3].

1.1.2 MINIX

MINIX - ще одна Unix-подібна система, але з абсолютно протилежним підходом. Третя версія (MINIX 3, 2005 рік) перетворилася на повноцінну дослідницьку ОС з акцентом на стабільності [4].

Архітектура MINIX - мікроядро. Це означає: саме ядро робить мінімум роботи. Решта - файлові системи, мережі, драйвери пристроїв - віддається у

простір звичайного користувача. Вони взаємодіють лише через інтерфейс повідомлень.

Якщо драйвер дає збій, завершується лише цей процес. Ядро бачить проблему, перезапускає відмовника, решта системи при цьому спокійно працює далі.

Недолік очевидний - вартість цієї розділеності. Будь-який доступ до диска - це вже кілька пересилань повідомлень, кожне супроводжується перемиканням контексту, копіюванням даних, додатковими перевірками.

1.1.3 Порівняння

Таблиця 1.1 – Порівняння монолітного ядра та мікроядра

Характеристика	Linux (моноліт)	MINIX 3 (мікроядро)
Адресний простір	Один (kernel space)	Ядро + ізольовані сервіси
Продуктивність I/O	Висока	Відчутно нижча (через IPC)
Ізоляція драйверів	Немає	Повна (user space процеси)
Збій драйвера	Kernel panic	Перезапуск процесу
Складність розробки	Вища, без захисту	Трохи менша
Застосування	Сервери, хмари	Дослідження, контролери

1.2 Архітектури ядер операційних систем

Від архітектури ядра залежить, що виконуватиметься з максимальними правами, як підсистеми обмінюються даними та які механізми обмеження вони отримують.

1.2.1 Монолітна архітектура

У класичному монолітному ядрі все збирається в єдиний виконуваний файл, підсистеми працюють разом, викликають функції напряму з мінімальними затримками. Мінус - жодної ізоляції: якщо щось зламалося в

одному місці, наслідки можуть бути фатальними для всього ядра. Так працюють Linux, FreeBSD, OpenBSD.

1.2.2 Мікроядро

Ядро виконує лише мінімум необхідного (планування, контроль адрес, обмін повідомленнями), решта підсистем винесена у простір звичайного користувача й бачить світ лише через контрольований IPC. Головна сила тут - надійність і формальна перевіряємість, як у seL4 [5]. Разом з тим такий підхід дає відчутні втрати на швидкодії через накладні витрати IPC.

1.2.3 Гібридна архітектура

Гібридні ядра поєднують обидва підходи: частина підсистем всередині ядра заради швидкодії, інші - ізольовано у user space. Яскраві приклади - Windows NT та macOS (XNU).

1.2.4 Sandbox-моделі у ядрі

Окремо варто згадати про пісочницю всередині самого ядра. Прикладом є Software Fault Isolation (SFI) [6], де неперевірений або сторонній код виконується з обмеженим доступом до ресурсів ядра. Така концепція пісочниці є базою для ідеї запуску драйверів через WebAssembly.

Таблиця 1.2 – Порівняння архітектур ядер

Архітектура	Продуктивність	Ізоляція	Приклади
Монолітна	Висока	Немає	Linux, FreeBSD
Мікроядро	Нижча	Повна	MINIX 3, seL4
Гібридна	Помірна	Часткова	Windows NT, XNU
Sandbox у ядрі	Висока	Часткова	eBPF, Wasm-ядро

1.3 Проблеми класичних підходів до розробки драйверів

Драйвери пристроїв - болюча точка системного програмування. Основних проблем тут три: безпека, складність розробки та платформна залежність.

1.3.1 Безпека

У Linux та подібних системах драйвер завжди працює у ring 0. Будь-яка помилка може не лише спричинити збій системи, а й відкрити зловмиснику повний доступ до неї.

1.3.2 Складність розробки

Для написання драйвера під Linux чи FreeBSD потрібно розбиратися у управлінні пам'яттю, особливостях апаратної платформи, нестабільних API. Ядро не гарантує стабільного ABI: з виходом нової версії навіть ідеально написаний драйвер раптом перестас збиратись або працювати нормально.

1.3.3 Платформна залежність

Драйвери завжди прив'язані до апаратної архітектури: специфічні інструкції, DMA-відображення, бар'єри пам'яті. Якщо треба перенести модуль з x86 на ARM чи RISC-V, доведеться переписувати і окремо збирати під кожную платформу.

Таблиця 1.3 – Проблеми класичних підходів до розробки драйверів

Проблема	Прояв	Наслідок
Відсутність ізоляції	Драйвер у ring 0	Kernel panic, ескалація прав
Складний ABI	Нестабільний API ядра	Перекомпіляція при оновленні
Платформна залежність	Arch-specific код	Потрібне портування для кожної ISA
Складність налагодження	Немає user-space інструм.	Довгий цикл дебагу і тестування
Нема sandboxing	Повний доступ до пам'яті	Будь-який збій компрометує систему

1.4 Огляд WebAssembly як технологічної основи

1.4.1 Модель виконання та особливості

WebAssembly (Wasm) - відкритий стандарт бінарних інструкцій, затверджений W3C у 2019 році [7]. Хоча в назві є "Web", по суті це універсальна цільова платформа для мов на кшталт C, C++, Rust. Wasm-модуль - всього лише байткод у бінарному .wasm-файлі, розроблений так, щоб виконання коду було ідентичним на будь-якій платформі, де є відповідний рантайм.

1.4.2 Безпека

Wasm працює всередині визначеного буфера "лінійна пам'ять". Якщо код намагається вилізти за ці межі - він миттєво зупиняється із trap-помилкою. Уся взаємодія із зовнішнім світом - тільки через явно заданий інтерфейс імпорту/експорту. Для стандартизованого системного інтерфейсу існує специфікація WASI [8], яка описує набір дозволених операцій ОС.

1.4.3 Переносимість

Wasm-байткод не залежить від апаратної архітектури. Один і той же .wasm-файл однаково працює хоч на x86-64, хоч на ARM64, хоч на RISC-V. Дослідження з виконання Wasm-модулів безпосередньо у ядрі демонструють практичну застосовність цього підходу для ізоляції системного коду [9].

1.5 Висновки до розділу

Усі основні архітектури ядер розриваються між двома крайнощами: або висока швидкість на шкоду ізоляції (Linux), або жорстка ізоляція, але ціною втрати на комунікаціях (MINIX 3). Класична розробка драйверів - джерело критичних багів і головна поверхня атаки.

WebAssembly дає структурні засоби вирішення цих проблем: його модель пам'яті гарантує ізоляцію, байткод універсальний, імпорти та експорти явно описують, що можна і що не можна.

РОЗДІЛ 2

ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Вимоги до системи

Після аналізу предметної області у першому розділі сформульовано вимоги до операційної системи. Вони діляться на функціональні й нефункціональні.

2.1.1 Функціональні вимоги

Система повинна вміти:

- а) підтримувати витісняючу багатозадачність із ізольованими процесами;
- б) запускати user-space програми у форматі ELF та обслуговувати базові POSIX-сумісні системні виклики;
- в) мати ієрархічну файлову систему та можливість монтувати кілька файлових систем;
- г) завантажувати й виконувати драйвери пристроїв у вигляді WebAssembly-модулів;
- д) забезпечити механізм доставки сигналів між ядром і user-space процесами;
- е) працювати з типовими пристроями вводу-виводу: PS/2 клавіатура, текстовий термінал.

2.1.2 Нефункціональні вимоги

- а) ізоляція: навіть якщо драйвер зазнає збою, це не повинно зупиняти все ядро чи зачіпати інші підсистеми;
- б) простота написання драйверів: розробнику достатньо знати лише документований набір host-функцій, а не внутрішнє влаштування ядра.

Алгоритм ініціалізації ядра операційної системи

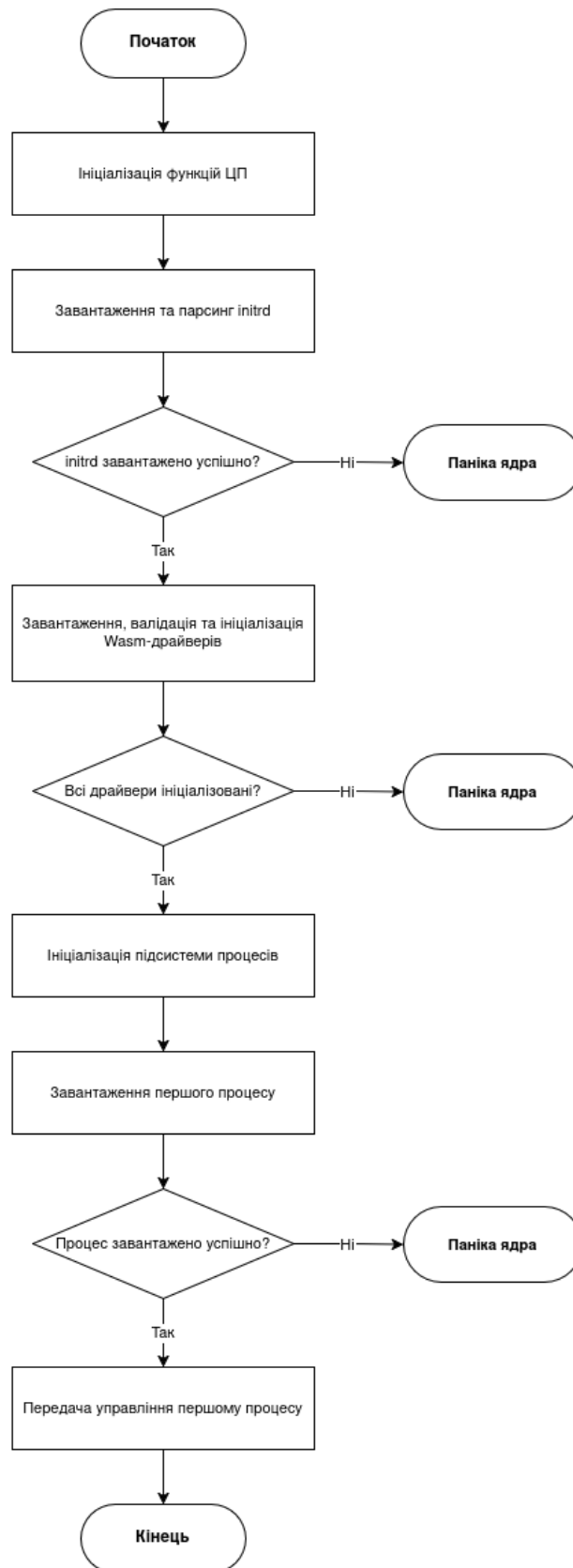


Рисунок 2.1 – Функціональна схема

2.2 Підсистема управління пам'яттю

2.2.1 Алгоритм управління фізичною пам'яттю

Для управління фізичною пам'яттю обрано бітову карту з кешованим курсором. Вона проста, економна до пам'яті (1 біт на 4 KiB) і цілком швидка для типових завдань - особливо коли алокації йдуть послідовно. У типовому випадку виділення відбувається за константний час, а у гіршому - лінійно.

2.2.2 Організація віртуального адресного простору

Ядро знаходиться у верхній половині віртуального простору, решта - під user space. Це класика: додатки розміщуються в нижній області, а ядро присутнє у сторінкових таблицях всіх процесів, що дозволяє уникнути частих TLB-збросів. Для зручного доступу до фізичної пам'яті організовується Higher Half Direct Map (HHDM). Це сильно спрощує алокатор пам'яті та виключає необхідність створювати тимчасові маппінги.

2.2.3 Вибір алокатора пам'яті ядра

Як динамічний алокатор ядра використовується freelist-алокатор із стратегією first-fit та автоматичним злиттям сусідніх вільних блоків при звільненні (coalescing). Він простий і не створює зайвої фрагментації.

2.3 Початковий образ файлової системи (initrd)

На етапі запуску системи ядру потрібен initrd - початковий образ у форматі TAR (USTAR) [10]. Його просто парсити і легко зібрати стандартними засобами.

2.4 Підсистема процесів

2.4.1 Модель процесу

Процес описується структурою, де є все мінімально необхідне: збережені регістри, PID/PPID/PGID, список файлових дескрипторів, дані для керування пам'яттю, масив обробників сигналів та стан процесу.

2.4.2 Алгоритм планування

Планувальник: витісняючий Round-Robin [11, с. 158].

2.4.3 Механізм блокування

Для очікування подій (I/O, sleep, wait) використовується механізм черг очікування - задачі підписуються на подію, і як тільки вона трапляється, ядро переводить задачі зі стану очікування у виконання.

2.5 Підсистема Wasm-драйверів

2.5.1 Підхід до виконання Wasm-модулів

Розглядалися інтерпретатор, JIT та AOT. Вибрано JIT: драйверам треба реагувати швидко. AOT вимагав би компілювати під кожну архітектуру - це псує портативність WebAssembly. JIT дає пристойну продуктивність і залишається універсальним. Він транслює байткод у x86-64 прямо, без складного проміжного представлення, що скорочує час завантаження і спрощує реалізацію.

2.5.2 API між ядром і Wasm-модулем

Усі операції драйвери виконують тільки через явно дозволені host-функції. Драйвер не бачить нічого зайвого, немає доступу до сирих пристроїв, портів чи сторінок - тільки те, що ядро дає. Кожна host-функція проста, всі звернення у пам'ять та дескриптори перевіряються на валідність.

Host API охоплює такі категорії: файлову систему, реєстрацію пристроїв, обробку переривань, роботу з відеобуфером, копіювання пам'яті у/з user space, роботу із задачами й сигналами, доступ до I/O портів та синхронізацію між задачами.

2.5.3 Модель ізоляції

Ізоляція тут залізна: пам'ять кожного драйвера окрема, доступ до ресурсів тільки через API, довільний код виконати неможливо - все пропускається через власний JIT-компілятор, який жорстко контролює генерований код.

2.5.4 Життєвий цикл драйвера

Wasm-файл читається із initrd, парситься, валідується, лінкується з host-функціями ядра, компілюється JIT-ом у машинний код і викликається точка входу. Якщо щось не так на етапі лінкування - наприклад, модуль намагається імпортувати невідому функцію - драйвер не завантажується.

2.5.5 Інтеграція з VFS

Взаємодія реалізується через адаптер: драйвер експонує функції типу fs_mount, fs_open та інші, адаптер підганяє виклики VFS під необхідний wasm-інтерфейс. Нові файлові системи можна додавати просто як нові Wasm-модулі.

2.6 VFS та системні виклики

2.6.1 Принципи VFS

Віртуальна файлова система побудована як розширювана структура з двома рівнями [11, с. 296]: FileSystem (реєструється окремо, прив'язана до mountpoint) і VfsFile (відкритий файл із операціями). При відкритті файлу VFS розбирає шлях з урахуванням симлінків, визначає файлову систему та делегує виклик.

2.6.2 Підмножина системних викликів

З системних викликів вибрано POSIX-мінімум: робота з файлами (open, read, write та інші), процеси (fork, execve та інші), управління пам'яттю (brk, mmap), сигнали, прості дії з каталогами, базовий IPC через pipe. Цього вистачає, щоб запускати прості Unix-програми.

2.7 Проєктування Wasm-драйвера файлової системи ext2

Як практичне підтвердження підсистеми Wasm-драйверів спроектовано драйвер файлової системи ext2. Вона обрана через простоту специфікації, широке використання у вбудованих системах і зручність для тестування: образ легко створити стандартними утилітами.

2.7.1 Структура ext2

Файлова система ext2 організована у групи блоків. Кожна група містить бітову карту блоків, бітову карту інодів, таблицю інодів і блоки даних. Суперблок (зміщення 1024 байт від початку) зберігає глобальні параметри: розмір блоку, кількість груп, розмір inode. Дескриптори груп (BGDT) розташовані одразу після суперблоку.

Inode описує файл: режим, розмір, часові мітки та масив з 15 номерів блоків - 12 прямих, один одинарно непрямої, один подвійно непрямої і один потрійно непрямої. Каталог - це звичайний файл, що містить послідовність записів фіксованого формату: номер inode, довжина запису, довжина імені, тип файлу та ім'я.

2.7.2 Інтерфейс з VFS

Драйвер реалізує стандартний набір експортованих функцій: fs_mount, fs_open, fs_read, fs_write, fs_stat, fs_readdir, fs_ref, fs_unref. VFS-адаптер ядра відображає ці функції на внутрішні операції VfsFile і FileSystem. Весь дисковий I/O йде виключно через host-функції vfs_read/vfs_write/vfs_lseek - прямого доступу до пам'яті ядра чи портів драйвер не має.

2.7.3 Робота з блоками та алокація

Читання і запис файлів виконуються поблочно через read-modify-write: для кожного блоку файлу обчислюється його номер на диску (прямий або через таблицю непрямої адресації), блок зчитується, модифікується і записується назад. Алокація нових блоків та інодів реалізована через сканування бітових карт BGDТ із оновленням лічильників вільних блоків і вільних інодів у відповідному дескрипторі групи.

2.8 Висновки до розділу

У цьому розділі запропонована архітектура мінімалістичної Unix-подібної операційної системи з ізолятором для Wasm-драйверів прямо у ядрі. Основні рішення:

- а) монолітне ядро плюс sandbox для драйверів (без класичних накладних витрат на IPC);
- б) бітова карта з курсором для фізичної пам'яті як золота середина між складністю, швидкістю і економією пам'яті;
- в) простий, але ефективний freelist для динаміки пам'яті ядра;
- г) планувальник Round-Robin;
- д) однопрохідна JIT-компіляція - баланс між реальною швидкістю та простотою;
- е) інтерфейс Kernel API спроектований максимально безпечним, із жорсткою валідацією;
- є) VFS побудований як адаптерний розширювальний фреймворк - Wasm-драйвери легко інтегруються;
- ж) ext2 обрана як демонстраційна файлова система для Wasm-драйвера: проста специфікація, стандартний набір операцій, зручність створення образів.

Завдяки цьому система повністю відповідає вимогам з початку розділу: функціональні покриваються за рахунок POSIX-сумісності й підтримки Wasm-драйверів, нефункціональні - через JIT, лінійну пам'ять, sandbox та простий і вузький Kernel API.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Технологічний стек

Основу ядра написано на мові Odin [12]. Вона дає прямий контроль над пам'яттю, немає прихованих алокацій і обов'язкового runtime. Компілятор Odin використовує LLVM як бекенд, що у тому числі підтримує wasm як цільову архітектуру.

Завантажувач - Limine v9 [13]. Він підтримує UEFI і Legacy BIOS, передає ядру інформацію про пам'ять, відеобуфер і initrd через Limine Boot Protocol.

3.1.1 Limine Boot Protocol

Limine Boot Protocol [13] - механізм запит/відповідь між ядром і завантажувачем. Ядро декларує у своєму ELF-файлі набір глобальних структур-запитів з фіксованими магічними числами (два 64-бітних значення). До передачі керування Limine сканує ELF-секції ядра, знаходить ці структури й заповнює у кожній поле response вказівником на відповідь у пам'яті.

Ядро оголошує такі запити:

- а) hhdmm - зсув Higher Half Direct Map (базова адреса, де фізична пам'ять відображена у віртуальний простір);
- б) memmap - карта фізичної пам'яті з типами регіонів (USABLE, RESERVED, BOOTLOADER_RECLAIMABLE тощо);
- в) framebuffer - адреса та параметри відеобуфера UEFI GOP (ширина, висота, формат пікселя);
- г) module - список файлів-модулів, переданих завантажувачем (initrd.tar та інші);

Після того як Limine заповнив усі відповіді, він переводить ядро у 64-бітний режим (якщо ще не в ньому), вмикає сторінкову трансляцію з identity

map і передає керування на точку входу ядра у вигляді виклику C-функції без аргументів. Усі дані доступні через поле response відповідного запиту.

3.2 Послідовність завантаження системи

Як тільки Limine передає керування точці входу ядра (kernel_start), починається ланцюг ініціалізації:

```
kernel_start :: proc "c" () {
    enable_sse()
    mem_init()
    kmem_init()
    context allocator = fl_allocator()
    cpu_init(mp_request.response.cpus[0])
    vfs_init()
    irq_stubs_init()
    ramblk_init()
    wasm_init()

    // пошук initrd.tar серед модулів Limine
    initrd_file: ^LimineFile = nil
    for i in 0..

```

```

    apic_timer_enable()
}

```

3.3 Реалізація підсистеми управління пам'яттю

3.3.1 Фізична пам'ять

Менеджер фізичної пам'яті реалізує бітову карту: кожен біт означає сторінку розміром 4 КіБ. Типи USABLE сторінки від Limine одразу позначаються як вільні, інші - зайняті.

Алокація послідовних сторінок (`reserve_pages`) використовує кешований курсор (`last_idx`), що забезпечує $O(1)$ у типовому випадку послідовних алокацій:

```

reserve_pages :: proc(count: u64) -> u64 {
    @(static) last_idx: u64 = 0
    i := last_idx
    for iter in 0..<limit {
        if phys_map_get(i) {
            // перевірка count послідовних вільних сторінок
            for j in 0..<count {
                phys_map_clear(i + j)
            }
            last_idx = i + count
            return i * PAGE_SIZE
        }
        i = (i + 1) % limit
    }
}

```

3.3.2 Віртуальна пам'ять

Менеджер віртуальної пам'яті реалізує стандартну 4-рівневу сторінкову таблицю x86-64. Структура `PageTableEntry` описана як `bit_field`, що дає прямий доступ до прапорців типу `present`, `rw`, `us`, `xd`:

```

PageTableEntry :: bit_field u64 {
    present: bool | 1,
    rw:      bool | 1,
    us:      bool | 1,
    // ...
    address: u64   | 40,
    xd:      bool | 1,
}

```

HHDM-зсув отримується від Limine (hhdm_request.response.offset) і використовується для трансляції фізичних адрес у віртуальні. Функція `vm_mmap` виділяє сторінки, відображає їх у вказану таблицю та повертає віртуальну адресу.

При створенні нового процесу алокується окрема PML4-таблиця, де верхня половина (записи 256–511) копіюється з ядра, нижня (0–255) очищується для user-space:

```
new_pml4 := transmute(^PageTable)(reserve_page() + hhdm_offset)
intrinsic.mem_zero(new_pml4, PAGE_SIZE)
for i in 256..<512 {
    new_pml4.entries[i] = pml4.entries[i] // спільне ядро
}
```

3.3.3 Алокатор пам'яті ядра

Freelist-алокатор реалізує стратегію first-fit із автоматичним злиттям сусідніх блоків при звільненні (coalescing). Він встановлюється як `context allocator` Odin, тому стандартні бібліотечні можливості - динамічні масиви, `map`, `strings` - працюють прозоро.

3.4 Реалізація підсистеми процесів

3.4.1 Структура задачі

Всі задачі описані структурою `Task`. Основне - `TaskRegs`: повний контекст процесора при переключенні, всі регістри, CR3 (таблиця сторінок), RSP, RIP, RFLAGS, сегментні регістри CS/SS:

```
Task :: struct {
    ctx:          TaskContext, // збережені регістри + xstate
    pid, ppid, pgid: int,
    kernel_stack: rawptr,
    open_files:   ^TaskFiles,  // MAX_FD = 16
    mm:           ^TaskMM,     // VMA-список, brk
    state:        TaskState,   // Running | Blocked | Zombie
    signal_actions: [64]KSigAction,
    pending_signals: u64,
    child_wait:   WaitQueue,
    // ...
}
```

TaskRegs зберігає повний контекст процесора при перемиканні: усі регістри загального призначення, CR3, RSP, RIP, RFLAGS та сегментні регістри CS/SS.

3.4.2 Перемикання контексту і планування

Планувальник живе у `switch_task`, її викликає обробник таймера APIC. Алгоритм обходить масив задач, пропускає `Zombie` й перевіряє умови пробудження для `BlockedInterruptible` (сигнали, `sleep timeout`):

```
switch_task :: proc(regs: ^TaskRegs) {
  if cur := get_current_task(); cur != nil {
    cur.ctx.regs = regs^
  }
  for _ in 0..<len(tasks) {
    current_task_i = (current_task_i + 1) % len(tasks)
    t := tasks[current_task_i]
    if t.state == .Running do break
    if t.state == .BlockedInterruptible &&
      t.pending_signals & ~t.signal_mask != 0 {
      t.state = .Running
      break
    }
  }
  next := tasks[current_task_i]
  regs^ = next.ctx.regs
  deliver_pending_signals_irq(regs)
}
```

Після вибору наступної задачі контекст відновлюється записом у структуру регістрів обробника переривання, що при поверненні з переривання (`iretq`) автоматично перемикає стек, таблицю сторінок та покажчик інструкцій.

3.4.3 Системні виклики

В ядрі все стартує через інструкцію `SYSCALL`. Для цього спочатку налаштовуються `Model-Specific Registers (MSR)`:

```
enable_syscalls :: proc() {
  writemsr(0xC0000081, 0x0013000800000000) // STAR
  writemsr(0xC0000082, transmute(u64)syscall_handler_asm)
  writemsr(0xC0000084, 0xFFFFFFFFFFFFFFFFD) // SFMASK
  writemsr(0xC0000080, readmsr(0xC0000080) | 1) // EFER.SCE
}
```

Таблиця 3.1 – Реалізовані системні виклики за категоріями

Категорія	Системні виклики
Файлові операції	open, read, write, close, lseek, dup, dup2, fcntl, stat, fstat, ioctl, getdents64
Процеси	fork, execve, exit, waitid, getpid, getppid, setpgid, getpgrp
Пам'ять	brk, mmap
Сигнали	sigaction, sigreturn, kill, sigsuspend
Каталоги	mkdir, unlink, rmdir, chdir, getcwd, access, faccessat
IPC	pipe
Система	nanosleep, getrusage, mount, getticks

3.4.4 Файлові системні виклики

open - типовий для файлових систем: спочатку перевірка рядка з user-space, далі резолюція шляху щодо поточного каталогу процесу, делегування в VFS і видача дескриптора:

```
sys_open :: proc "c" (filename: cstring, flags: int, mode: int)
-> int {
    filenamestr, ok := validate_user_string(filename)
    if !ok do return -int(Error.EFAULT)

    task := get_current_task()
    resolved := vfs_resolve_path(task.cwd, filenamestr)
    defer delete(resolved)

    file, err := vfs_open(resolved, flags, mode)
    if err != .ENONE do return -int(err)

    fd, err_fd := task_get_free_fd(task)
    if err_fd != .ENONE {
        vfs_close(file)
        return -int(err_fd)
    }
    task.open_files.fds[fd] = file
    if flags & O_CLOEXEC != 0 {
        task.open_files.fd_flags[fd] = FD_CLOEXEC
    }
    return fd
}
```

Інші типові системні виклики - read, write, close, lseek, stat, dup, dup2, ioctl, getdents64 - перевіряють дескриптор через task_get_file, буфер user-space і делегують відповідній операції FileOps.

3.4.5 Надсилання сигналів (kill)

kill - надсилає сигнал конкретному процесу або групі. Якщо pid > 0 - шукаю по PID, якщо pid < -1 - по PGID. Якщо процес BlockedInterruptible, він прокидається одразу:

```
sys_kill :: proc "c" (pid: int, sig: int) -> int {
  if sig < 0 || sig >= MAX_SIGNALS do return -
int(Error.EINVAL)
  if pid > 0 {
    for t in tasks {
      if t.pid == pid {
        if sig == 0 do return 0 // перевірка існування
t.pending_signals |= u64(1) << u64(sig)
        if t.state == .BlockedInterruptible {
          t.state = .Running
        }
        return 0
      }
    }
    return -int(Error.ESRCH)
  } else if pid < -1 {
    pgid := -pid
    sent := 0
    for t in tasks {
      if t.pgid == pgid {
        t.pending_signals |= u64(1) << u64(sig)
        if t.state == .BlockedInterruptible {
          t.state = .Running
        }
        sent += 1
      }
    }
    if sent == 0 do return -int(Error.ESRCH)
    return 0
  }
  return -int(Error.ESRCH)
}
```

3.4.6 Затримка виконання (nanosleep)

nanosleep переводить процес у BlockedInterruptible на потрібну кількість тіків таймера. Якщо під час сну приходить сигнал - повертається EINTR:

```
sys_nanosleep :: proc "c" (req: ^TimeSpec, rem: ^TimeSpec) ->
int {
  if !validate_user_buffer(req, size_of(TimeSpec), false) {
    return -int(Error.EFAULT)
  }
  ticks_to_sleep := u64(req.tv_sec * 50)
  ticks_to_sleep += u64(req.tv_nsec / 20_000_000)
```

```

    if ticks_to_sleep > 0 {
        if task_sleep_until(ticks + ticks_to_sleep) {
            return -int(Error.EINTR)
        }
    }
    return 0
}

```

3.4.7 Монтування файлових систем (mount)

mount працює і з вбудованими, і з Wasm-файловими системами. Спочатку шукається вбудована реалізація за типом, а за її відсутності - Wasm-драйвер, зареєстрований через fs_register:

```

sys_mount :: proc "c" (source: cstring, target: cstring,
                      fstype: cstring, flags: int, data:
rawptr) -> int {
    source_str := ""
    if source != nil {
        s, ok := validate_user_string(source)
        if !ok do return -int(Error.EFAULT)
        source_str = s
    }
    target_str, ok2 := validate_user_string(target)
    if !ok2 do return -int(Error.EFAULT)
    fstype_str, ok3 := validate_user_string(fstype)
    if !ok3 do return -int(Error.EFAULT)

    task := get_current_task()
    resolved := vfs_resolve_path(task.cwd, target_str)
    defer delete(resolved)

    if native_fn, found := native_fs_types[fstype_str]; found {
        err := native_fs_do_mount(native_fn, source_str,
resolved)
        if err != .ENONE do return -int(err)
        return 0
    }
    typ, found := wasm_fs_types[fstype_str]
    if !found do return -int(Error.ENODEV)
    err := wasm_fs_do_mount(typ, source_str, resolved)
    if err != .ENONE do return -int(err)
    return 0
}

```

3.4.8 Реалізація fork

fork робить глибоке копіювання 4-рівневої таблиці сторінок. Нижня половина (user-space) рекурсивно клонується до рівня PTE, де сторінки з прапорцями `us` та `rw` копіюються фізично:

```
sys_fork :: proc "c" () -> int {
    cur_task := get_current_task()
    pml4 := transmute(^PageTable)(cur_task.ctx.regs.cr3 +
    hhdm_offset)
    new_pml4 := transmute(^PageTable)(reserve_page() +
    hhdm_offset)
    mem.zero(transmute(^u8)(new_pml4), PAGE_SIZE)

    for i in 0..<256 { // клонування user-space
        if pml4.entries[i].present {
            // ... рекурсивний обхід PDPT → PD → PT
            for l in 0..<512 { // рівень PTE
                if t3Table.entries[l].us &&
t3Table.entries[l].rw {
                    addr := reserve_page()
                    mem.copy(virt_new, virt_old, PAGE_SIZE)
                    newt3Table.entries[l].address = addr >> 12
                } else {
                    newt3Table.entries[l].address =
                        t3Table.entries[l].address // спільна
                }
            }
        }
    }
    for i in 256..<512 { // спільне ядро
        new_pml4.entries[i] = pml4.entries[i]
    }

    new_task := new(Task)
    new_task^ = cur_task^
    new_task.pid = next_pid; next_pid += 1
    new_task.ppid = cur_task.pid
    new_task.ctx.regs.cr3 = phys_addr(new_pml4)
    new_task.ctx.regs.rah = 0 // дочірній бачить 0
    new_task.state = .Running
    append(tasks, new_task)
    return new_task.pid // батьківський бачить PID
}
```

3.4.9 Реалізація execve

execve - завантажує ELF64-бінарник у процес. Перше - валідація заголовків, копіювання `argv/envp` у ядро (ще до знищення старої пам'яті), потім завантаження `PT_LOAD` сегментів:

```

sys_execve :: proc "c" (pathname: cstring,
                        argv: [^]cstring, envp: [^]cstring) ->
int {
    // ... відкриття файлу, зчитування у elf_buf

    ehdr := transmute(^Elf64_Ehdr)&elf_buf[0]
    if ehdr.e_ident[0] != 0x7F || ehdr.e_ident[1] != 'E' ||
        ehdr.e_ident[2] != 'L' || ehdr.e_ident[3] != 'F' {
        return -int(Error.ENOEXEC)
    }
    if ehdr.e_type != .ET_EXEC      do return -
int(Error.ENOEXEC)
    if ehdr.e_machine != .EM_X86_64 do return -
int(Error.ENOEXEC)

    // копіювання argv/envp у ядро до знищення адресного
простору
    args := make([dynamic]string)
    if argv != nil {
        for i := 0; argv[i] != nil; i += 1 {
            s, s_ok := validate_user_string(argv[i])
            if !s_ok do return -int(Error.EFAULT)
            append(&args, strings.clone(s))
        }
    }

    // — point of no return —
    vm_free_user_half(task.ctx.regs.cr3)

    for i in 0..<int(ehdr.e_phnum) {
        ph := phdrs[i]
        if ph.p_type == .PT_LOAD {
            page_count := aligned_size(ph.p_memsz) / PAGE_SIZE
            vm_mmap(pml4, u64(ph.p_vaddr), page_count,
{.R,.W,.X}, true)
            mem.zero(rawptr(uintptr(ph.p_vaddr)),
int(ph.p_memsz))
            mem.copy(rawptr(uintptr(ph.p_vaddr)),
                    rawptr(uintptr(&elf_buf[0]) +
uintptr(ph.p_offset)),
                    int(ph.p_filesz))
        }
    }
    // побудова стеку: [argc, argv[], NULL, envp[], NULL]
}

```

3.4.10 Реалізація waitid

waitid - батьківський процес блокується в черзі WaitQueue, доки дитина не стане Zombie. Як тільки знаходиться zombie-процес - заповнюється структура SigInfo, вивільняються ресурси:

```
sys_waitid :: proc "c" (which: int, id: int,
                        infop: ^SigInfo, options: int) -> int {
  task := get_current_task()
  for {
    has_child := false
    for i in 0..<len(tasks) {
      child := tasks[i]
      if child.ppid != task.pid do continue
      if which == P_PID && child.pid != id do continue
      has_child = true
      if child.state == .Zombie {
        if infop != nil {
          infop.si_signo = SIGCHLD
          infop.si_pid    = i32(child.pid)
          infop.si_status = i32(child.exit_code)
        }
        vm_free_page_tables(child.ctx.regs.cr3)
        ordered_remove(tasks, i)
        free(child)
        return 0
      }
    }
    if !has_child do return -int(Error.ECHILD)
    if options & WNOHANG != 0 { return 0 }
    if task_sleep(&task.child_wait, .BlockedInterruptible) {
      return -int(Error.EINTR)
    }
  }
}
```

3.4.11 Механізм сигналів

Механізм сигналів - POSIX-сумісний. sigaction задає обробники, а при доставці сигналу ядро формує на стеку SigFrame з trampoline-кодом, контекст реєстрів і маску сигналів:

```
deliver_pending_signals :: proc "c" () {
  task := get_current_task()
  pending := task.pending_signals & ~task.signal_mask
  if pending == 0 do return

  signo := 0
  for signo < MAX_SIGNALS && (pending >> u64(signo)) & 1 == 0
  {
```

```

        signo += 1
    }
    task.pending_signals &= ~(u64(1) << u64(signo))

    action := &task.signal_actions[signo]
    handler := action.handler

    if handler == SIG_IGN do return
    if handler == SIG_DFL {
        if signo == 17 do return // SIGCHLD: ignore
        task.exit_code = -signo
        task_do_exit(task)
        for { int41() }
    }

    // побудова signal frame на стеку user-space
    sig_sp := orig_rsp - 112 - uintptr(size_of(SigFrame))
    sig_sp &= ~uintptr(15) // вирівнювання 16 байт
    sigframe := transmute(^SigFrame)sig_sp

    // trampoline: mov rax, 21; syscall
    sigframe.trampoline = {
        0x48, 0xC7, 0xC0, 21, 0, 0, 0, 0x0F, 0x05,
        0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
    }
    sigframe.orig_rsp = u64(orig_rsp)
    sigframe.saved_mask = task.signal_mask
    mem.copy(&sigframe.frame[0], rawptr(frame_ptr), 112)

    task.signal_mask |= action.mask[0]
    if action.flags & SA_NODEFER == 0 {
        task.signal_mask |= u64(1) << u64(signo)
    }

    // перенаправлення виконання на обробник
    (transmute(^u64)(new_frame + 64))^ = u64(signo) // rdi
    (transmute(^u64)(new_frame + 96))^ = handler // rcx =
RIP
}

```

Після обробника trampoline-код викликає `sys_sigreturn`, який повертає збережений контекст і маску:

```

sys_sigreturn :: proc "c" () -> int {
    sigframe := transmute(^SigFrame)(uintptr(user_ctx.rsp))
    task := get_current_task()
    task.signal_mask = sigframe.saved_mask
    user_ctx.rax = sigframe.rax
    new_frame := uintptr(sigframe.orig_rsp) - 112
    mem.copy(rawptr(new_frame), &sigframe.frame[0], 112)
    (transmute(^u64)(uintptr(task.kernel_stack) - 8))^ =
u64(new_frame)
}

```

```

    return int(sigframe.rax)
}

```

3.5 Реалізація підсистеми Wasm-драйверів

3.5.1 Парсер Wasm-модулів

Парсер розбирає бінарний формат WebAssembly, спочатку перевіряє магічне число (0x6D736100) та версію, далі йде секціями. Результат - структура WasmModuleMetadata:

```

parse_metadata :: proc(wasm_bytes: []byte) ->
(WasmModuleMetadata, bool) {
    bytes := slice.as_ptr(wasm_bytes)
    magic := (cast(^u32)(&bytes[0]))^
    if magic != WASM_MAGIC do return {}, false
    version := ((cast(^u32)&bytes[4]))^
    if version != WASM_VERSION do return {}, false
    wasm_bytes := wasm_bytes[8:]
    mod := WasmModuleMetadata { start_function = -1 }
    for len(wasm_bytes) != 0 {
        wasm_bytes = parse_section(&mod, wasm_bytes) or_return
    }
    return mod, true
}

```

parse_section обробляє секцію за її типом (Type, Import, Function, Memory, Export, Code і т.д.). Наприклад, Import - парсить ім'я модуля, ім'я символу, тип імпорту (функція, пам'ять, глобал, таблиця):

```

parse_section :: proc(mod: ^WasmModuleMetadata, bytes: []u8)
-> (result: []u8, ok: bool) {
    type := SectionType(bytes[0])
    bytes = bytes[1:]
    ln: u32 = 0
    ln, bytes = parse_uleb_32(bytes)
    #partial switch type {
    case .Type:
        types: u32 = 0
        types, bytes = parse_uleb_32(bytes)
        mod.types = make([]Type, types)
        for i in 0..<types {
            mod.types[i], bytes = parse_type(bytes)
        }
    case .Import:
        imports: u32 = 0
        imports, bytes = parse_uleb_32(bytes)
        mod.imports = make([]Import, imports)
        for i in 0..<imports {

```



```

        // парсинг module_name, name, type
(Func/Mem/Global/Table)
        // ...
    }
    // аналогічно для Function, Memory, Export, Code, Data, ...
}
return bytes, true
}

```

3.5.2 Завантаження та лінування модулів

Модулі збираються в ExecutionContext - по суті реєстр завантажених модулів. Коли завантажується модуль (exectx_load_module) - виконується лінування: imports зв'язуються з exports у контексті:

```

exectx_load_module :: proc(ctx: ^ExecutionContext,
    metadata: ^WasmModuleMetadata, name: string) ->
ModuleLoadError {
    mod := new(WasmModule)
    mod.metadata = metadata
    mod.functions = make([dynamic]^WasmFunction)

    // лінування імпортів
    for &imp in metadata.imports {
        module, found := ctx.modules[imp.module_name]
        if !found do return .ModuleNotFound
        #partial switch i in imp.type {
        case FunctionType:
            fn := ImportedFunction { type = i, imp = &imp }
            fnidx, ok := module.exported_funcs[imp.name]
            if ok do fn.resolved = module.functions[fnidx]
            append(&mod.functions, new_clone(WasmFunction(fn)))
        case Memory:
            mem_idx := module.exported_memories[imp.name]
            mem := ImportedMemory { mem =
module.memory[mem_idx] }
            append(&mod.memory, new_clone(WasmMemory(mem)))
        // аналогічно для Global, Table
        }
    }

    // реєстрація власних функцій, глобалів, пам'яті, таблиць
    for fn, i in metadata.functions {
        func := OwnFunction {
            code = &metadata.code[i],
            type = fn.(FunctionType),
            mod = mod,
        }
        append(&mod.functions, new_clone(WasmFunction(func)))
    }

    // реєстрація exports
    for export in metadata.export {

```

```

        if export.idx_type == .Func {
            mod.exported_funcs[export.name] = int(export.idx)
        }
        // ...
    }
    ctx.modules[name] = mod
    return nil
}

```

3.5.3 Валідація функцій

Перед JIT-компіляцією кожна функція проходить валідацію (`validate_function`): стек типів моделюється, перевіряються операнди кожної інструкції. Валідатор тримає стек блоків управління - перевіряє `br/br_if/br_table`:

```

validate_function :: proc(fn: ^OwnFunction, main_alloc:
mem.Allocator)
                                -> ModuleLoadError {
    if fn.instructions == nil {
        fn.instructions, _ = parse_instructions(fn.mod.metadata,
fn.code)
    }
    // побудова масиву типів локальних змінних
    localss := make([]Type, len(fn.type.args) + local_count)
    for arg, i in fn.type.args { localss[i] = arg }

    val_ctx := ValCtx {
        fn, make([dynamic]Type), // type_stack
        fn.instructions, localss,
        make([dynamic]BlockStackEntry), // block_stack
    }
    append(&val_ctx.block_stack, BlockStackEntry {
        output_types = fn.type.ret,
        end = len(fn.instructions),
    })
    // послідовна перевірка кожної інструкції:
    // expect() - перевірка типів на стеку
    // push() - додавання результату
    // ...
}

```

3.5.4 JIT-компілятор

JIT-компілятор трансліює валідовані Wasm-функції у x86-64 машинний код в один прохід. Використовується бібліотека `x86asm` [14] - бібліотека з типобезпечним інтерфейсом для емітування інструкцій.

Для стеку значень Wasm - змішані підходи:

- а) 4 GP-реєстри (R12–R15) і 4 XMM-реєстри (XMM12–XMM15) для верхніх елементів;
- б) якщо не вистачає - значення падають у x86-стек.

`jit_function` ітерує інструкції Wasm і емітує x86-64 код:

```
jit_function :: proc(exectx: ^ExecutionContext, fn:
^OwnFunction) {
    ctx := JittingContext {
        fn = fn, exectx = exectx,
        block_stack = make([dynamic]JitBlockEntry),
        type_stack = make([]bool, fn.max_stack),
    }
    x86.init_asm(&ctx.xasm, false)
    using ctx; using x86
    epillog = create_label(&xasm)
    trapped = create_label(&xasm)
    jit_prolog()

    for instr in fn.instructions {
        #partial switch instr.opcode {
        case .i32_const:
            mov(&xasm, eax, i32(instr.args[0].(i32)))
            jit_push_gp(rax)
        case .i64_add:
            n2 := jit_pop_gp()
            n1 := jit_pop_gp()
            add(&xasm, n1, n2)
            jit_push_gp(n1)
        case .i64_div_u:
            n2 := jit_pop_gp()
            n1 := jit_pop_gp()
            mov(&xasm, rax, n1)
            mov(&xasm, rdx, 0)
            cmp(&xasm, n2, rdx)
            je(&xasm, trapped) // ділення на нуль → trap
            div(&xasm, n2)
            jit_push_gp(rax)
        // ... ще ~150 опкодів: f64_mul, local.get/set,
        //      block, loop, if, br, br_table, call, ...
        }
    }
    jit_epilog()
    fn.jitted = x86.assemble(&xasm)
}
```

Управління Wasm-блоками - `block`, `loop`, `if` - транслюються у мітки x86 із переходами. `br` - це `jmp` на кінець блоку (`block`) або початок (`loop`). `br_table` виходить з таблицею непрямих переходів.

3.5.5 Kernel API

Ядро експортує модуль "kernel" із набором host-функцій, які реєструються через `hexctx_define_func`. Host-функції отримують `FunctionContext` - через нього доступ до лінійної пам'яті модуля.

Доступ до пам'яті перевіряє `memory_get_slice`:

```
memory_get_slice :: proc(mem: ^WasmMemory, ptr: u32, count: i32)
-> []u8 {
    memory := memory_get_own(mem)
    size := u64(memory.allocated_pages << 16)
    if u64(ptr) + u64(count) > size do return nil
    return memory.data[ptr:][:count]
}
```

Крім базових операцій з файловою системою та пристроями, Kernel API включає доступ до портів вводу-виводу (`_inb`, `_outb`), управління IRQ (`_irq_open`, `_irq_read`, `_irq_set_callback`), копіювання між user-space та пам'яттю модуля (`_copy_from_user`, `_copy_to_user`), створення задач (`_spawn_task`) та блокуючих буферів (`_waitbuf_create`, `_waitbuf_read`, `_waitbuf_write`).

3.6 Реалізація віртуальної файлової системи

VFS - єдиний інтерфейс для різних файлових систем. Архітектура: `FsOps` - на рівні файлової системи, `FileOps` - на рівні відкритого файлу.

```
FileOps :: struct {
    read:      FileRead,
    write:     FileWrite,
    close:     FileClose,
    seek:      FileSeek,
    ioctl:     FileIoctl,
    getdents:  FileGetDents,
    stat:      FileStat,
}
FsOps :: struct {
    open:      FsOpen,
    mkdir:     FsMkdir,
    symlink:   FsSymlink,
    unlink:    FsUnlink,
    rmdir:     FsUnlink,
    getdents:  FsGetDents,
}
```

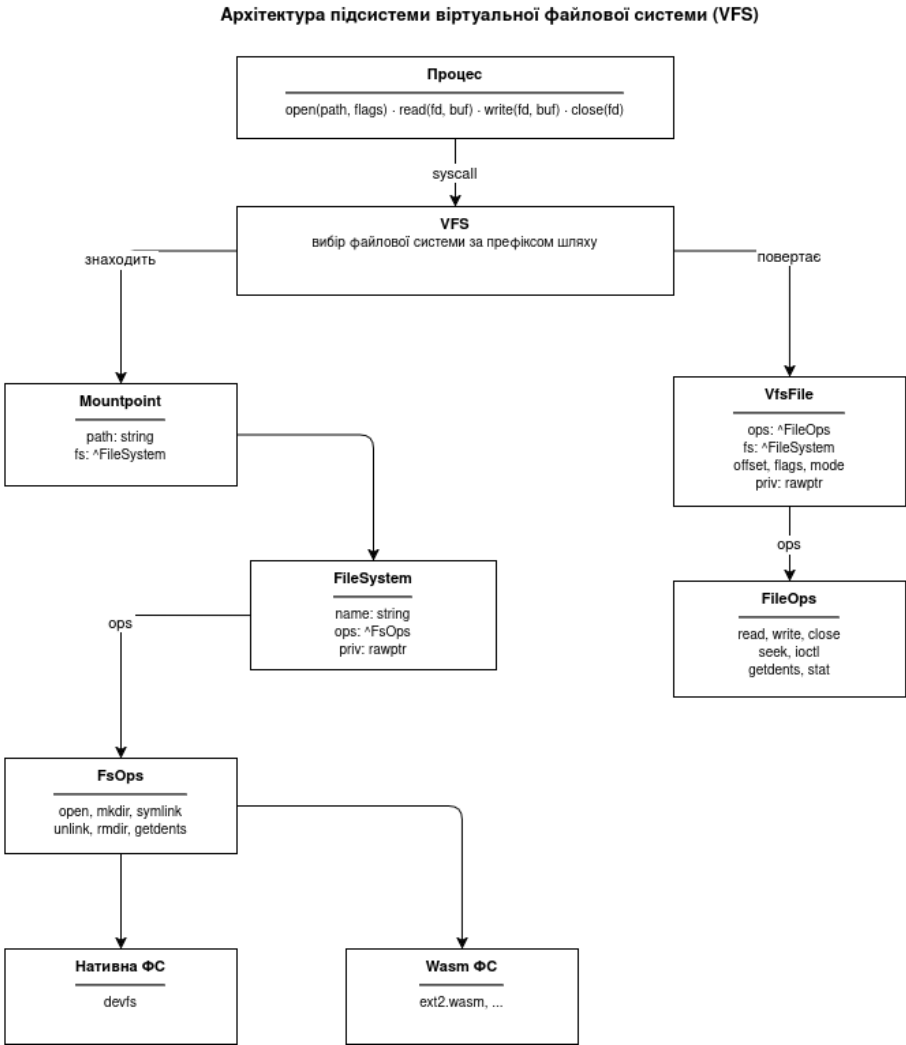


Рисунок 3.1 – Структура підсистеми віртуальної файлової системи

Відкритий файл описує VfsFile: тут знаходяться операції, файлова система, позиція й лічильник посилань. Файлова система реєструється через Mountpoint.

Є п'ять вбудованих файлових систем:

Таблиця 3.2 – Вбудовані файлові системи

Файлова система	Призначення
devfs	Реєстрація символьних пристроїв (/dev/tty0, /dev/kbd)
ramfs	Tmpfs-подібна ФС в оперативній пам'яті (symlink, mkdir, файли)

Продовження таблиці 3.2

Файлова система	Призначення
tarfs	Доступ до вмісту initrd.tar (read-only)
pipe	Однонаправлений канал з ring-буфером (4 KiB)
ramblk	Блоковий пристрій в RAM для монтування ext2

Адаптер для `wasm_fs` інтегрує Wasm-драйвери з VFS. `WasmFsType` містить вказівники на JIT-скомпільовані функції драйвера. Коли викликається VFS-операція, адаптер копіює дані у `scratch`-область лінійної пам'яті модуля, кличе Wasm-функцію, потім результат повертає назад.

3.7 Реалізовані Wasm-драйвери

Всі драйвери написані мовою Odin та скомпільовані у WebAssembly. Вони взаємодіють з ядром через імпортовані `host`-функції з модуля "kernel".

3.7.1 Драйвер терміналу (TTY)

Драйвер терміналу - графічний емулятор з підтримкою ANSI escape-послідовностей, канонічного режиму (редагування рядка) й сигналів (Ctrl-C).

Виведення символу: `tty_putchar` обробляє escape-послідовності та рендерить у тінювий фреймбufer:

```
tty_putchar :: proc "contextless" (ch: u8) {
    switch esc_state {
    case .Normal:
        switch ch {
        case 0x1B: esc_state = .EscSeen
        case '\n':
            cursor_x = 0
            cursor_y += 1
            if cursor_y >= rows { tty_scroll_up() }
        case '\r': cursor_x = 0
        case 0x08: // backspace
            if cursor_x > 0 { cursor_x -= 1 }
            screen[int(cursor_y)*int(cols)+int(cursor_x)] = ' '
            tty_render_char(cursor_x, cursor_y, ' ')
        case 0x20..=0x7E:
            screen[int(cursor_y)*int(cols)+int(cursor_x)] = ch
            tty_render_char(cursor_x, cursor_y, ch)
            cursor_x += 1
            if cursor_x >= cols {
```

```

        cursor_x = 0; cursor_y += 1
        if cursor_y >= rows { tty_scroll_up() }
    }
}
case .EscSeen:
    if ch == '[' { esc_state = .CSI; /* ... */ }
case .CSI:
    _tty_csi_feed(ch) // обробка CSI-послідовностей
}
}

```

Введення з клавіатури - окрема задача (tty0_input_loop), вона читає скан-коди з /dev/kbd, враховує Shift/Ctrl, обробляє сигнальні символи (Ctrl-C - SIGINT), буферизує рядки в канонічному режимі.

3.7.2 Драйвер клавіатури PS/2

Драйвер реєструє IRQ1 (порт 0x60), створює пристрій /dev/kbd, зчитує сирі скан-коди через irq_read:

```

package kbd
import "../..kapi"

scratch: [4]u8
irq_slot: i32 = -1

@export
kbd_read :: proc "c" (buf_ptr: i32, count: i32) -> i32 {
    if irq_slot < 0 || count <= 0 do return 0
    buf := (cast([^]u8)uintptr(buf_ptr))[:count]
    return kapi.irq_read(irq_slot, buf)
}

@export
driver_main :: proc "c" () {
    irq_slot = kapi.irq_open(1, 0x60)
    if irq_slot < 0 {
        kapi.printf("kbd: irq_open failed: %i\n", irq_slot)
        return
    }
    kapi.devfs_register("kbd", scratch[:])
    kapi.print("kbd: ready\n")
}

```

3.7.3 Драйвер файлової системи ext2

ext2 - підтримка читання і запису. Структури суперблоку та inode повністю відповідають специфікації:

```

Superblock :: struct #packed {

```

```

s_inodes_count:    u32,
s_blocks_count:    u32,
// ...
s_log_block_size:  u32,
s_blocks_per_group: u32,
s_inodes_per_group: u32,
s_magic:           u16, // 0xEF53
s_rev_level:       u32,
s_first_ino:       u32,
s_inode_size:      u16,
}

```

```

Inode :: struct #packed {
    i_mode:      u16,
    i_size:      u32,
    i_blocks_512: u32,
    i_block:     [15]u32,
    i_size_high: u32,
}

```

`fs_mount` зчитує суперблок зі зміщення 1024, перевіряє `magic` і ініціалізує глобальні параметри. BGDТ розташовується на блоці 2 для блоків 1024 байт або на блоці 1 для більших:

```

@export
fs_mount :: proc "c" (src_ptr: i32, src_len: i32) -> i64 {
    src := scratch[:src_len]
    fd  := kapi.vfs_open(string(src), kapi.O_RDWR)
    if fd < 0 do fd = kapi.vfs_open(string(src), kapi.O_RDONLY)
    if fd < 0 do return -5

    sb_buf: [90]u8
    if kapi.vfs_lseek(fd, 1024, SEEK_SET) != 1024
{ kapi.vfs_close(fd); return -5 }
    if kapi.vfs_read(fd, sb_buf[:]) != 90
{ kapi.vfs_close(fd); return -5 }
    sb := (^Superblock)(&sb_buf[0])
    if sb.s_magic != 0xEF53
{ kapi.vfs_close(fd); return -5 }

    blkfd      = fd
    block_size  = u32(1024) << sb.s_log_block_size
    inodes_per_group = sb.s_inodes_per_group
    blocks_per_group = sb.s_blocks_per_group
    inode_size   = sb.s_rev_level >= 1 ?
u32(sb.s_inode_size) : 128
    bgdt_block   = block_size > 1024 ? 1 : 2
    return 2 // кореневий inode ext2 завжди 2
}

```

Розрахунок номера диска для блоку файлу (`file_block`) підтримує 12 прямих і один рівень непрямої адресації:


```

file_block :: proc "contextless" (inode: ^Inode, blk_idx: u32)
-> u32 {
    ptrs := block_size / 4
    if blk_idx < 12 do return inode.i_block[blk_idx]
    idx := blk_idx - 12
    if idx < ptrs {
        indir := inode.i_block[12]
        if indir == 0 do return 0
        if !read_bytes(i64(indir) * i64(block_size),
indir_buf[:block_size]) do return 0
        return ([^]u32)(&indir_buf[0])[idx]
    }
    return 0
}

```

Пошук файлу (`dir_lookup`) перебирає блоки каталогу і порівнює записи `ext2_dir_entry` за іменем:

```

dir_lookup :: proc "contextless" (dir_ino: i64, name: []u8) ->
i64 {
    if !load_inode(dir_ino) do return 0
    inode := (^Inode)(&inode_buf[0])
    nblocks := (u32(inode.i_size) + block_size - 1) /
block_size
    for bi in u32(0)..<nblocks {
        blkno := file_block(inode, bi)
        if blkno == 0 || !read_bytes(i64(blkno) *
i64(block_size), blk_buf[:block_size]) do continue
        off: u32 = 0
        for off + 8 <= block_size {
            d_ino := (^u32)(&blk_buf[off])^
            rec_len := (^u16)(&blk_buf[off+4])^
            nlen := blk_buf[off+6]
            if rec_len == 0 do break
            if d_ino != 0 && int(nlen) == len(name) {
                match := true
                for i in 0..<int(nlen) {
                    if blk_buf[off+8+u32(i)] != name[i] { match
= false; break }
                }
                if match do return i64(d_ino)
            }
            off += u32(rec_len)
        }
    }
    return 0
}

```

`load_inode` знаходить `inode` через BGDТ: обчислює групу і локальний індекс, зчитує відповідний рядок BGDТ, потім читає сам `inode` з таблиці інодів:

```

load_inode :: proc "contextless" (ino: i64) -> bool {
    group := u32(ino - 1) / inodes_per_group
    local := u32(ino - 1) % inodes_per_group

```

```

    bgdt_off := i64(bgdt_block) * i64(block_size) + i64(group) *
32    if !read_bytes(bgdt_off, bgdt_buf[:32]) do return false
    inode_table := (^u32)(&bgdt_buf[8])^
    disk_off    := i64(inode_table) * i64(block_size) +
i64(local) * i64(inode_size)
    return read_bytes(disk_off, inode_buf[:inode_size])
}

```

fs_read ітерує по блоках файлу через **file_block** і копіює дані у буфер користувача; спарсені блоки заповнюються нулями:

```

@export
fs_read :: proc "c" (ino: i64, offset: i64, buf_ptr: i32, count:
i32) -> i32 {
    if blkfd < 0 || !load_inode(ino) do return -5
    inode      := (^Inode)(&inode_buf[0])
    file_size := i64(inode.i_size) | (i64(inode.i_size_high) <<
32)
    if offset >= file_size do return 0
    rem    := int(min(i64(count), file_size - offset))
    out    := ([^]u8) (uintptr(buf_ptr))
    total := 0
    for total < rem {
        pos      := offset + i64(total)
        blk_idx  := u32(pos / i64(block_size))
        blk_off  := int(pos % i64(block_size))
        to_copy  := min(rem - total, int(block_size) - blk_off)
        blkno    := file_block(inode, blk_idx)
        if blkno == 0 {
            for i in 0..<to_copy { out[total+i] = 0 }
        } else {
            if !read_bytes(i64(blkno) * i64(block_size),
blk_buf[:block_size]) do return -5
            for i in 0..<to_copy { out[total+i] =
blk_buf[blk_off+i] }
        }
        total += to_copy
    }
    return i32(total)
}

```

fs_write виконує read-modify-write: якщо блок не виділений - **alloc_block** виділяє новий через бітову карту BGDТ; після запису оновлює розмір **inode**:

```

@export
fs_write :: proc "c" (ino: i64, offset: i64, buf_ptr: i32,
count: i32) -> i32 {
    if blkfd < 0 || !load_inode(ino) do return -5
    inode_copy: [256]u8
    for i in 0..<int(inode_size) { inode_copy[i] =
inode_buf[i] }
    inode := (^Inode)(&inode_copy[0])
    in_ptr := ([^]u8) (uintptr(buf_ptr))

```

```

total := 0
for total < int(count) {
    pos := offset + i64(total)
    blk_idx := u32(pos / i64(block_size))
    blk_off := int(pos % i64(block_size))
    to_copy := min(int(count) - total, int(block_size) -
blk_off)
    if blk_idx >= 12 do break
    blkno := inode.i_block[blk_idx]
    if blkno == 0 {
        blkno = alloc_block()
        if blkno == 0 do break
        inode.i_block[blk_idx] = blkno
        inode.i_blocks_512 += block_size / 512
        for i in 0..

```

fs_open з прапорцем **O_CREATE** виділяє новий inode, ініціалізує його поля й додає запис до каталогу батька через **dir_add_entry**:

```

@export
fs_open :: proc "c" (path_ptr: i32, path_len: i32, flags: i32,
mode: i32) -> i64 {
    if blkfd < 0 do return -5
    path := scratch[:path_len]
    ino := path_lookup(path)
    if ino > 0 {
        if flags & kapi.O_CREATE != 0 && flags & 0x80 != 0 do
return -17 // EEXIST
        return ino
    }
    if flags & kapi.O_CREATE == 0 do return ino
}

```

```

parent_ino, name := path_lookup_parent(path)
if parent_ino <= 0 || len(name) == 0 do return -2

new_ino := alloc_inode()
if new_ino <= 0 do return -28 // ENOSPC

for i in 0..

```

РОЗДІЛ 4

ТЕСТУВАННЯ СИСТЕМИ ТА ОЦІНКА РЕЗУЛЬТАТІВ

4.1 Формування мінімального користувацького оточення

Щоб перевірити роботу системи і створити рівні умови для тестування, я зібрав мінімальне користувацьке оточення. В нього входять стандартна C-бібліотека (mlibc) [15], базові Unix-утиліти з набору sbase [16] і командний інтерпретатор oksh [17].

4.1.1 Портування стандартної бібліотеки C (mlibc)

Я використав mlibc [15] – переносиму стандартну бібліотеку C, яка була створена максимально портативною. Щоб вона запрацювала, треба було написати спеціальний прошарок sysdeps, який перетворює виклики бібліотеки у системні виклики ядра. Весь цей прошарок розміщено у двох файлах: entry.cpp і generic.cpp.

entry.cpp відповідає за старт програми: розбирає, що там у ехес-стеку (argc, argv, envp), задає локаль, запускає глобальні конструктори (.init_array) й передає керування main:

```
extern "C" void __mlibc_entry(uintptr_t *entry_sp) {
    __mlibc_initLocale();
    mlibc::parse_exec_stack(entry_sp, &__mlibc_stack_data);
    mlibc::set_startup_data(__mlibc_stack_data.argc,
                           __mlibc_stack_data.argv,
                           __mlibc_stack_data.envp);
    for (InitFunc *fn = __init_array_start; fn !=
         __init_array_end; fn++)
        (*fn)();
    auto result = main(__mlibc_stack_data.argc,
                      __mlibc_stack_data.argv, environ);
    exit(result);
}
```

generic.cpp містить реалізацію системних викликів ядра. Всі вони йдуть через статичну функцію syscall, яка прямо викликає інструкцію SYSCALL – з аргументами, як того вимагає System V AMD64 ABI:

```
static long syscall(long num, long a1=0, long a2=0, long a3=0,
                   long a4=0, long a5=0, long a6=0) {
    long ret;
```

```

register long r10 asm("r10") = a4;
register long r8  asm("r8")   = a5;
register long r9  asm("r9")   = a6;
asm volatile("syscall"
             : "=a"(ret)
             : "a"(num), "D"(a1), "S"(a2), "d"(a3),
               "r"(r10), "r"(r8), "r"(r9)
             : "rcx", "r11", "memory");
return ret;
}

int sys_read(int fd, void *buf, size_t count, ssize_t
*bytes_read) {
    long ret = syscall(SYS_READ, fd, (long)buf, count);
    if (ret < 0) return -ret;
    *bytes_read = ret;
    return 0;
}

int sys_write(int fd, const void *buf, size_t count, ssize_t
*bytes_written) {
    long ret = syscall(SYS_WRITE, fd, (long)buf, count);
    if (ret < 0) return -ret;
    *bytes_written = ret;
    return 0;
}

```

Якщо `syscall` повертає негативне значення – це код помилки POSIX.

4.1.2 Збірка базових Unix-утиліт (sbase)

На базі портованої `mlibc` я скомпілював 18 утиліт із `sbase` [16]: `ls`, `cat`, `echo`, `true`, `false`, `pwd`, `mkdir`, `rm`, `cp`, `touch`, `tail`, `head`, `sleep`, `mv`, `kill`, `find`, `ed`, `cut`. Кожна збирається як статичний ELF-файл:

- а) Явні опції `-nostdinc` та `-isystem` гарантують, що компілятор бачить тільки заголовки з `mlibc`, ігноруючи файли системи хоста.
- б) Готові утиліти я склав у каталог `/bin` в `initrd`-образі.

4.1.3 Збірка командного інтерпретатора (oksh)

В ролі командного інтерпретатора виступає `oksh` [17] – порт Korn Shell з OpenBSD. Він підтримує інтерактивний режим, пайпи, редиректи, змінні середовища, запуск скриптів. `oksh` стартує автоматично як процес із PID 1.

Всі програми функціонують "як є", без виправлення вихідного коду.

```

unix# ls
bin
child
dev
ext2.img
ext2.wasm
fstest
init
init.sh
kbd.wasm
mnt
test-native.sh
test-wasm.sh
test.txt
tty.wasm
unix#

```

Рисунок 4.1 – Робочий термінал з командним інтерпретатором oksh

4.2 Розробка нативного драйвера файлової системи ext2

Щоб знайти накладні витрати Wasm-рантайму, я написав нативний драйвер ext2 прямо в ядрі, який є майже ідентичним до wasm варіанту. Він реєструється під іменем ext2native.

Структура Ext2Fs утримує параметри файлової системи та посилання на блоковий пристрій, а Ext2File відстежує інформацію про відкритий файл: inode, розмір, зону блоків:

```

Ext2Fs :: struct {
    blkdev:      ^VfsFile,
    block_size:  int,
    inode_size:  int,
    inodes_per_group: u32,
    blocks_per_group: u32,
    bgdt_block:   u32,
    writable:    bool,
}

Ext2File :: struct {
    fs:      ^Ext2Fs,
    ino:     i64,
    size:    i64,
    mode:    u16,
    i_block: [15]u32, // 12 direct + 1 indirect + 2 reserved
}

```

Інтерфейс до файлової системи (VFS) зроблений через таблиці функцій ext2_ops (FsOps) і ext2_fileops (FileOps), де є вказівники на відкриття, читання,

запис, пошук, закриття, отримання статистики, отримання списку файлів, mkdir, unlink, rmdir.

4.3 Програма порівняльного тестування

Щоб поміряти реальну продуктивність файлових операцій, я розробив утиліту `fstest` (`userland/fstest.c`). Вона написана на С без стандартної бібліотеки – весь доступ лише через системні виклики ядра. Для замірів часу використовується `getticks`, який повертає поточний лічильник АРІС-таймера.

`fstest` приймає три аргументи: який блоковий пристрій використовувати, куди монтувати і який тип файлової системи (`ext2native` або `ext2`). Далі монтується образ і виконується такий цикл 5000 разів:

- а) створити файл `/mnt/fstest_tmp` з прапорцями `O_CREATE` і `O_TRUNC`;
- б) записати 4096 байт із шаблоном і `& 0xFF`;
- в) зробити `seek` на початок;
- г) прочитати ті ж 4096 байт;
- д) перевірити правильність даних побайтово;
- е) видалити файл (`unlink`).

Час на кожній операції фіксується окремо. Перевірка правильності даних не входить до часу запису і читання. Ключовий вимірювальний цикл:

```
for (int iter = 0; iter < ITERATIONS; iter++) {
    unsigned long long t0, t1;

    t0 = sys_getticks();
    int fd = sys_open(path, O_CREATE | O_TRUNC, 0644);
    t1 = sys_getticks();
    t_create += t1 - t0;

    t0 = sys_getticks();
    sys_write(fd, write_buf, BUF_SIZE);    // 4096 байт
    t1 = sys_getticks();
    t_write += t1 - t0;

    sys_lseek(fd, 0, 0);
    t0 = sys_getticks();
    sys_read(fd, read_buf, BUF_SIZE);
    t1 = sys_getticks();
    t_read += t1 - t0;
```



```

sys_close(fd);

t0 = sys_getticks();
sys_unlink(path);
t1 = sys_getticks();
t_delete += t1 - t0;
}

```

Тест запускається два рази: спершу з ext2native (нативний драйвер), потім з ext2 (Wasm-драйвер), причому умови ідентичні – образ /dev/ram0, та сама точка монтування /mnt, незмінні параметри QEMU.

4.4 Результати та аналіз

4.4.1 Умови проведення тестування

Все тестування відбувалось у QEMU з такими налаштуваннями:

Таблиця 4.1 – Умови тестування

Параметр	Значення
Платформа	QEMU 11.0.0, x86-64, Cascadelake-Server-v5
ОЗП	1 ГБ
Диск	ext2, 128 КБ, /dev/ram0
Ітерацій	5000 для кожного сценарію
Буфер	4096 байт
Метрика	тіки APIC-таймера

4.4.2 Порівняльна таблиця продуктивності

Таблиця нижче порівнює, скільки часу займає кожна операція у нативному та Wasm-драйвері ext2 (в тіках, сумарно за 5000 ітерацій):

Таблиця 4.2 – Порівняння продуктивності нативного та Wasm-драйвера ext2

Операція	Нативний	Wasm	Різниця (Wasm – Нативний)
Створення файлу	8	5	–3
Запис 4 КБ	33	43	+10
Читання 4 КБ	7	17	+10
Видалення файлу	7	5	–2

4.4.3 Аналіз результатів

Ці результати дають чітке уявлення про накладні витрати, які створює підсистема Wasm-драйверів у порівнянні з нативною реалізацією. Найтяжчі – це операції запису і читання: Wasm-драйвер на кожній з них витрачає на 10 тіків більше. У випадку читання це в основному йде на виклик функції через таблицю функцій JIT-компільованого модуля і на копіювання з scratch-буфера Wasm у буфер користувача. Для запису додається ще операція read-modify-write: будь-яка операція з блоком відбувається через host-функції (`_vfs_read` / `_vfs_write`) з копіюванням даних між лінійною пам'яттю модуля і ядром.

Операції створення і видалення файлу, навпаки, виявились навіть трохи швидшими у Wasm-реалізації (на 3 та 2 тіка відповідно). Але ця різниця незначна, бо тут все впирається у вибір inode і оновлення метаданих, а абсолютні значення занадто малі для стабільної оцінки.

У підсумку: за 5000 ітерацій сумарний час у нативного драйвера - 55 тіків, у Wasm - 70 тіків. Тобто, накладні витрати Wasm-рантайму складають 27% від загального часу, якщо враховувати роботу з RAM-блоковим пристроєм із мінімальною затримкою. При роботі з реальним диском цей показник був би в рази менший.

4.5 Висновки до розділу

У цьому розділі я на практиці перевірів і функціональність, і продуктивність розробленої системи. Коротко підсумую:

- а) вдалося портувати стандартну бібліотеку C (mlibc), реалізувавши 42 системних виклики на sysdeps – цього вистачає, щоб виконувати звичайні програми з POSIX-інтерфейсом;
- б) скомпілював 18 базових Unix-утиліт із sbase і командний інтерпретатор oksh без зміни коду - тобто практична сумісність реалізованих системних викликів із вимогами реального софту справді є;
- в) розробив нативний драйвер ext2 на Odin і програму fstest для детального тестування: 5000 ітерацій, типові операції створення/запису/читання/видалення файлу розміром 4 КБ;
- г) порівняльне тестування показало: накладні витрати Wasm-рантайму на 5000 ітерацій дають +10 тіків при записі та читанні (але -3 і -2 на create/delete); сумарне "відставання" 27% цілком пояснюється копіюванням буферів між лінійною пам'яттю модуля і ядром.

Врешті, архітектурне рішення з виконанням драйверів у Wasm-пісочниці справді робоче й повноцінне. Воно гарантує ізоляцію драйверів без надмірних накладних витрат.

ВИСНОВКИ

У роботі розроблено ядро Unix-подібної операційної системи з підтримкою написання драйверів пристроїв на WebAssembly. За результатами виконаної роботи можна зробити такі висновки.

Аналіз існуючих підходів показав, що монолітні ядра (Linux) забезпечують високу продуктивність, але не мають ізоляції між драйверами і ядром, а мікроядра (MINIX 3) дають ізоляцію ціною суттєвих накладних витрат на IPC. WebAssembly запропонував третій шлях: sandbox всередині монолітного ядра з лінійною пам'яттю, явним імпорт/експорт-інтерфейсом і повною апаратною незалежністю.

Спроектовано та реалізовано такі підсистеми ядра:

- а) менеджер фізичної пам'яті на основі бітової карти з кешованим курсором та freelist-алокатор для динамічної пам'яті ядра;
- б) витісняючий планувальник Round-Robin з механізмом черг очікування для блокувальних операцій;
- в) підсистема процесів із підтримкою fork/exec, доставки сигналів і системних викликів POSIX-підмножини;
- г) віртуальна файлова система (VFS) з підтримкою монтування кількох файлових систем і симлінків;
- д) однопрохідний JIT-компілятор WebAssembly у x86-64 машинний код із жорстким Kernel API для драйверів.

Практичну придатність системи підтверджено портуванням стандартної бібліотеки C (mlibc) з реалізацією 42 системних викликів, компіляцією 18 Unix-утиліт (sbase) та командного інтерпретатора oksh без змін вихідного коду.

Порівняльне тестування Wasm-драйвера ext2 проти нативної реалізації (5000 ітерацій, операції create/write/read/delete для файлу 4 КБ) показало накладні витрати рантайму на рівні 27% при роботі з RAM-блоковим

пристроєм. Для реального дискового I/O цей показник був би суттєво нижчим, оскільки основний час займає апаратний доступ, а не копіювання буферів.

Отримані результати підтверджують, що виконання драйверів у Wasm-sandbox є практично реалізованим і функціонально повноцінним рішенням: воно забезпечує ізоляцію драйверів без надмірних накладних витрат і не потребує від розробника знання внутрішньої будови ядра.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Netcraft. Web Server Survey, April 2024. Netcraft. 2024. URL: <https://www.netcraft.com/blog/april-2024-web-server-survey/> (дата звернення: 01.05.2026).
2. Chen H., Mao Y., Wang X. et al. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. Proc. of the 2nd Asia-Pacific Workshop on Systems (APSys'11). 2011. URL: <https://pdos.csail.mit.edu/papers/chen-kbugs.pdf> (дата звернення: 05.05.2026).
3. Li Z., Zhu S., Levy M. et al. Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel. Proc. of the Annual Computer Security Applications Conference (ACSAC'24). 2024. URL: <https://mars-research.github.io/doc/2024-acsac-rfl.pdf> (дата звернення: 06.05.2026).
4. Herder J. N., Bos H., Tanenbaum A. S. MINIX 3: A Highly Reliable, Self-Repairing Operating System. ACM SIGOPS Operating Systems Review. 2006. Vol. 40, No. 3. С. 80–89.
5. Klein G., Elphinstone K., Heiser G. et al. seL4: Formal Verification of an OS Kernel. Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP'09). 2009. С. 207–220.
6. Mao Y., Chen H., Wang X. et al. Software Fault Isolation with API Integrity and Multi-Principal Modules. Proc. of the 23rd ACM Symp. on Operating Systems Principles (SOSP'11). 2011. С. 115–128.
7. W3C WebAssembly Working Group. WebAssembly Core Specification. W3C Recommendation. 2019. URL: <https://www.w3.org/TR/wasm-core-1/> (дата звернення: 03.05.2026).
8. Bytecode Alliance. WebAssembly System Interface (WASI). wasi.dev. 2022. URL: <https://wasi.dev> (дата звернення: 04.05.2026).
9. Goltzsche D., Neeles S., Kapitza R. KernelWasm: Efficient Sandboxing of Kernel Modules. Proc. of the European Workshop on Systems Security (EuroSec'22). 2022. С. 1–7.

10. IEEE / The Open Group. pax - portable archive interchange (ustar format). IEEE Std 1003.1-2017. 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html#tag_20_92_13_06 (дата звернення: 10.05.2026).
11. Tanenbaum A. S., Bos H. Modern Operating Systems. 4-те вид. Boston : Pearson, 2015. 1136 с.
12. The Odin Programming Language: Overview. odin-lang.org. 2024. URL: <https://odin-lang.org/docs/overview/> (дата звернення: 14.05.2026).
13. Limine Contributors. Limine Boot Protocol Specification. GitHub. 2024. URL: <https://github.com/limine-bootloader/limine/blob/v9.x/PROTOCOL.md> (дата звернення: 08.05.2026).
14. xb-bx. x86asm - x86-64 assembler library for Odin. GitHub. 2024. URL: <https://github.com/xb-bx/x86asm> (дата звернення: 16.05.2026).
15. Managarm Project Contributors. mlibc: A portable C standard library. GitHub. 2024. URL: <https://github.com/managarm/mlibc> (дата звернення: 11.05.2026).
16. suckless.org. sbase - suckless Unix tools. suckless.org. 2024. URL: <https://core.suckless.org/sbase/> (дата звернення: 12.05.2026).
17. Iribarren B. oksh - portable OpenBSD ksh. GitHub. 2024. URL: <https://github.com/ibara/oksh> (дата звернення: 13.05.2026).

Додаток А

Лістинг вихідного коду віртуальної файлової системи (VFS)

```

package kernel
import "base:intrinsic"
import "core:strings"
log2 :: intrinsic.constant_log2
O_RDONLY    :: 0x00000
O_WRONLY    :: 0x00001
O_RDWR     :: 0x00002
O_CREATE    :: 0x00040
O_EXCL      :: 0x00080
O_NOCTTY    :: 0x00100
O_TRUNC     :: 0x00200
O_NONBLOCK  :: 0x00800
O_APPEND    :: 0x00400
O_SYNC      :: 0x01000
O_ASYNC     :: 0x02000
O_CLOEXEC   :: 0x80000

FileRead    :: proc(file: ^VfsFile, buf: []u8) -> (int, Error)
FileWrite   :: proc(file: ^VfsFile, buf: []u8) -> (int, Error)
FileClose   :: proc(fs: ^FileSystem, file: ^VfsFile) -> Error
FileSeek    :: proc(fs: ^FileSystem, file: ^VfsFile, off: int,
whence: SeekWhence) -> (int, Error)
FileIoctl   :: proc(file: ^VfsFile, request: uint, arg: uintptr)
-> (int, Error)
FileGetDents :: proc(file: ^VfsFile, buf: []u8) -> (int, Error)
FileStat    :: proc(file: ^VfsFile, st: ^Stat) -> Error
FileOps     :: struct {
    read:      FileRead,
    write:     FileWrite,
    close:     FileClose,
    seek:      FileSeek,
    ioctl:     FileIoctl,
    getdents:  FileGetDents,
    stat:      FileStat,
}
DirEntryType :: enum {
    DT_UNKNOWN = 0,
    DT_FIFO    = 1,
    DT_CHR     = 2,
    DT_DIR     = 4,
    DT_BLK     = 6,
    DT_REG     = 8,
    DT_LNK     = 10,
    DT_SOCK    = 12,
    DT_WHT     = 14,
}
DirEntry    :: struct {
    inode: int,

```



```

        type: DirEntryType,
        name: u8, // first char
    }
    FsOpen      :: proc(fs: ^FileSystem, filename: string, flags:
int, mode: int) -> (^VfsFile, VfsUnresolvedSymlink, Error)
    FsMkdir     :: proc(fs: ^FileSystem, filename: string, mode:
int) -> (^VfsFile, VfsUnresolvedSymlink, Error)
    FsSymlink   :: proc(fs: ^FileSystem, filename: string, target:
string) -> (VfsUnresolvedSymlink, Error)
    FsUnlink    :: proc(fs: ^FileSystem, filename: string) ->
(VfsUnresolvedSymlink, Error)
    FsGetDents  :: proc(fs: ^FileSystem, fd: ^VfsFile, dents:
[]DirEntry) -> (int, Error)
    FsOps :: struct {
        open:      FsOpen,
        mkdir:     FsMkdir,
        symlink:   FsSymlink,
        unlink:    FsUnlink,
        rmdir:     FsUnlink,
        getdents:  FsGetDents,
    }
    VfsFile :: struct {
        ops:      ^FileOps,
        fs:       ^FileSystem,
        flags:    int,
        mode:     int,
        offset:   int,
        priv:     rawptr,
        ref_count: int,
        entry_type: DirEntryType,
        path:     string,
    }
}

vfs_close :: proc(file: ^VfsFile) -> Error {
    if file == nil do return .ENONE
    file.ref_count -= 1
    if file.ref_count <= 0 {
        if file.ops.close != nil {
            err := file.ops.close(file.fs, file)
            if err != .ENONE do return err
        }
        if file.path != "" { delete(file.path) }
        free(file)
    }
    return .ENONE
}

FileSystem :: struct {
    name:      string,
    mountpoint: ^Mountpoint,
    ops:       ^FsOps,
    priv:      rawptr,
}

Mountpoint :: struct {

```

```

    path: string,
    fs:    ^FileSystem,
}
VfsUnresolvedSymlink :: struct {
    symlink:      string,
    symlink_was_allocated: bool,
}
NativeFsMountProc :: proc(source: string) -> (^FileSystem,
Error)
native_fs_types: map[string]NativeFsMountProc

mountpoints: map[string]^Mountpoint
vfs_init :: proc() {
    mountpoints      = make(map[string]^Mountpoint)
    wasm_fs_types    = make(map[string]^WasmFsType)
    native_fs_types  = make(map[string]NativeFsMountProc)
    ext2_register()

    devfsfs := devfs_init()
    mountpoints["/dev/"] = new_clone(Mountpoint{path = "/dev/",
fs = devfsfs})
    devfsfs.mountpoint = mountpoints["/dev/"]
}

native_fs_do_mount :: proc(mount_fn: NativeFsMountProc, source:
string, mountpoint: string) -> Error {
    fs, err := mount_fn(source)
    if err != .ENONE do return err
    mp := mountpoint
    if len(mp) == 0 || mp[len(mp)-1] != '/' {
        mp = strings.concatenate({mountpoint, "/"})
    } else {
        mp = strings.clone(mp)
    }
    mp_entry := new_clone(Mountpoint{path = mp, fs = fs})
    fs.mountpoint = mp_entry
    mountpoints[mp] = mp_entry
    return .ENONE
}

vfs_get_mount_fs :: proc(filename: string) -> (fs: ^FileSystem,
stripped_name: string) {
    fs = nil
    last_len := 0

    // Check for exact match (e.g., filename="/dev" matches
mount="/dev/")
    for k, v in mountpoints {
        // Strip trailing slash for comparison
        mount_str := k
        if len(mount_str) > 0 && mount_str[len(mount_str)-1] ==
 '/' {
            mount_str = mount_str[:len(mount_str)-1]

```

```

    }
    // Check: filename == mount_str OR filename == mount_str
+ "/"
    if filename == mount_str {
        fs = v.fs
        stripped_name = ""
        return fs, stripped_name
    }
    // Check if filename is mount_str + "/" (e.g., "/dev" vs
"/dev/")
    if len(filename) == len(mount_str) + 1 &&
filename[len(mount_str)] == '/' {
        if filename[:len(mount_str)] == mount_str {
            fs = v.fs
            stripped_name = ""
            return fs, stripped_name
        }
    }
}

// Check for prefix match (longest prefix wins)
for k, v in mountpoints {
    if strings.has_prefix(filename, k) && len(k) > last_len
{
        fs = v.fs
        last_len = len(k)
    }
}
if fs != nil do stripped_name = filename[last_len:]
return fs, stripped_name
}
vfs_mkdir :: proc(filename: string, mode: int, symlink_depth:
int = MAX_SYMLINKS) -> (^VfsFile, Error) {
    if symlink_depth <= 0 do return nil, .ELOOP

    fs, stripped_name := vfs_get_mount_fs(filename)
    if fs == nil do return nil, .ENOENT
    if fs.ops.mkdir == nil do return nil, .EROFS

    dir, symlink, err := fs.ops.mkdir(fs, stripped_name , mode)
    if err != nil do return nil, err

    if symlink.symlink != "" {
        defer if symlink.symlink_was_allocated do
delete(symlink.symlink)
        return vfs_mkdir(symlink.symlink, mode, symlink_depth -
1)
    } else {
        return dir, nil
    }
}

```

```

vfs_symlink :: proc(filename: string, target: string,
symlink_depth: int = MAX_SYMLINKS) -> Error {
    if symlink_depth <= 0 do return .ELOOP

    fs, stripped_name := vfs_get_mount_fs(filename)
    if fs == nil do return .ENOENT
    if fs.ops.symlink == nil do return .EROFS

    symlink, err := fs.ops.symlink(fs, stripped_name, target)
    if err != nil do return err

    if symlink.symlink != "" {
        defer if symlink.symlink_was_allocated do
delete(symlink.symlink)
        return vfs_symlink(symlink.symlink, target,
symlink_depth - 1)
    } else {
        return nil
    }
}

MAX_SYMLINKS :: 8
vfs_open :: proc(filename: string, flags: int, mode: int,
symlink_depth: int = MAX_SYMLINKS) -> (^VfsFile, Error) {
    if symlink_depth <= 0 {
        return nil, .ELOOP
    }
    fs, stripped_name := vfs_get_mount_fs(filename)

    if fs == nil do return nil, .ENOENT
    file, symlink, err := fs.ops.open(fs, stripped_name, flags,
mode)
    if err != nil do return nil, err
    if symlink.symlink != "" {
        defer if symlink.symlink_was_allocated do
delete(symlink.symlink)
        return vfs_open(symlink.symlink, flags, mode,
symlink_depth - 1)
    } else {
        if file != nil { file.path = strings.clone(filename) }
        return file, nil
    }
}

vfs_unlink :: proc(filename: string, symlink_depth: int =
MAX_SYMLINKS) -> Error {
    if symlink_depth <= 0 do return .ELOOP
    fs, stripped := vfs_get_mount_fs(filename)
    if fs == nil do return .ENOENT
    if fs.ops.unlink == nil do return .EPERM
    symlink, err := fs.ops.unlink(fs, stripped)
    if err != .ENONE do return err
    if symlink.symlink != "" {

```

```

        defer if symlink.symlink_was_allocated do
delete(symlink.symlink)
        return vfs_unlink(symlink.symlink, symlink_depth - 1)
    }
    return .ENONE
}

vfs_rmdir :: proc(filename: string, symlink_depth: int =
MAX_SYMLINKS) -> Error {
    if symlink_depth <= 0 do return .ELOOP
    fs, stripped := vfs_get_mount_fs(filename)
    if fs == nil do return .ENOENT
    if fs.ops.rmdir == nil do return .EPERM
    symlink, err := fs.ops.rmdir(fs, stripped)
    if err != .ENONE do return err
    if symlink.symlink != "" {
        defer if symlink.symlink_was_allocated do
delete(symlink.symlink)
        return vfs_rmdir(symlink.symlink, symlink_depth - 1)
    }
    return .ENONE
}

vfs_resolve_path :: proc(cwd: string, path: string) -> string {
    // Push path components onto a fixed stack, handling .
and ..
    stack: [128]string
    top := 0

    // For relative paths, seed with cwd components; for
absolute, start empty.
    for src in ([?]string{cwd if len(path) == 0 || path[0] !=
 '/' else "/", path}) {
        i := 0
        for i < len(src) {
            for i < len(src) && src[i] == '/' { i += 1 }
            j := i
            for j < len(src) && src[j] != '/' { j += 1 }
            if j == i do break
            switch src[i:j] {
            case ".":
                // current dir - skip
            case "..":
                if top > 0 { top -= 1 }
            case:
                if top < len(stack) { stack[top] = src[i:j]; top
+= 1 }
            }
            i = j
        }
    }
}

```

```

    if top == 0 do return strings.clone("/")
    parts := make([dynamic]string, 0, top * 2)
    defer delete(parts)
    for i in 0..

```

Додаток Б

Лістинг вихідного коду модуля розбору WebAssembly-модулів

```

package kernel
import "core:strings"
import "core:slice"
parse_metadata :: proc(wasm_bytes: []byte) -> (result:
WasmModuleMetadata, ok: bool) {
    bytes := slice.as_ptr(wasm_bytes)
    magic := (cast(^u32)(&bytes[0]))^
    if magic != WASM_MAGIC do return {}, false
    version := ((cast(^u32)&bytes[4])^
    if version != WASM_VERSION do return {}, false
    wasm_bytes := wasm_bytes[8:]
    mod := WasmModuleMetadata { start_function = -1 }
    for len(wasm_bytes) != 0 {
        wasm_bytes = parse_section(&mod, wasm_bytes) or_return
    }
    return mod, true
}
parse_section :: proc(mod: ^WasmModuleMetadata, bytes: []u8) ->
(result: []u8, ok: bool) {
    bytes := bytes
    type := SectionType(bytes[0])
    bytes = bytes[1:]
    ln: u32 = 0
    ln, bytes = parse_uleb_32(bytes)

    rest_bytes := bytes[ln:]
    #partial switch type {
    case .Type:
        types: u32 = 0
        types, bytes = parse_uleb_32(bytes)
        mod.types = make([]Type, types)
        for i in 0..

```

```

bytes = bytes[n1:]

n2, bytes = parse_uleb_32(bytes)
name := strings.string_from_ptr(&bytes[0],
int(n2))

bytes = bytes[n2:]

t := ExternTypeByte(bytes[0])
bytes = bytes[1:]
#partial switch t {
case .Func:
    idx: u32 = 0
    idx, bytes = parse_uleb_32(bytes)
    fntype := mod.types[idx]
    if f, ok := fntype.(FunctionType); ok {
        mod.imports[i] = Import {
            module_name = mod_name,
            name = name,
            type = f,
        }
        mod.first_own_fn_idx += 1
    } else {
        return nil, false
    }
case .Tag:
    bytes = bytes[1:]
    idx: u32 = 0
    idx, bytes = parse_uleb_32(bytes)
    fntype := mod.types[idx]
    if f, ok := fntype.(FunctionType); ok {
        mod.imports[i] = Import {
            module_name = mod_name,
            name = name,
            type = Tag(f),
        }
    } else {
        return nil, false
    }
case .Mem:
    limit := Limit {}
    limit, bytes = parse_limit(bytes)
    mod.imports[i] = Import {
        module_name = mod_name,
        name = name,
        type = Memory { limit, nil },
    }
case .Table:
    ref := RefType{}
    limit := Limit{}
    ref, bytes = parse_ref_type(mod, bytes)
    limit, bytes = parse_limit(bytes)

    mod.imports[i] = Import {

```



```

        module_name = mod_name,
        name = name,
        type = ExtTable { ref, limit },
    }
    case .Global:
        tb, bytes := parse_value_type(mod, bytes)
        mut := bool(bytes[0])
        bytes = bytes[1:]
        mod.imports[i] = Import {
            module_name = mod_name,
            name = name,
            type = ExtGlobal { tb, mut },
        }
    case:
        panic("todo")
    }
    }
    return rest_bytes, true
case .Function:
    funcs: u32 = 0
    funcs, bytes = parse_uleb_32(bytes)
    mod.functions = make([]Type, funcs)
    for i in 0..<funcs {
        idx: u32 = 0
        idx, bytes = parse_uleb_32(bytes)
        if idx < 0 || int(idx) >= len(mod.types) do
return nil, false
        mod.functions[i] = mod.types[idx]
    }
    return bytes, true
case .DataCount:
    mod.data_count, bytes = parse_uleb_32(bytes)
    return bytes, true
case .Code:
    codes: u32 = 0
    codes, bytes = parse_uleb_32(bytes)
    mod.code = make([]Code, codes)
    for i in 0..<codes {
        ln: u32 = 0
        ln, bytes = parse_uleb_32(bytes)
        code := bytes[:ln]
        orig_len := len(bytes)
        code_off := 0
        rest := bytes[ln:]
        locals: u32 = 0
        locals, bytes = parse_uleb_32(bytes)
        locs := make([]Local, locals)
        for j in 0..<locals {
            count: u32 = 0
            count, bytes = parse_uleb_32(bytes)
            type: Type = {}
            type, bytes = parse_value_type(mod, bytes)
            locs[j] = Local { int(count), type }

```

```

    }
    code_off = orig_len - len(bytes)
    code = code[code_off:]
    mod.code[i] = Code{
        bytes = code,
        locals = locs,
    }
    bytes = rest
}
return bytes, true
case .Data:
    datas: u32 = 0
    datas, bytes = parse_uleb_32(bytes)
    mod.data = make([]Data, datas)
    for i in 0..

```

```

    }
    return bytes, true
case .Global:
    globals: u32 = 0
    globals, bytes = parse_uleb_32(bytes)
    mod.global = make([]MetadataGlobal, globals)
    for i in 0..

```

```

        section_name := strings.string_from_ptr(&bytes[0],
int(length))
        bytes = bytes[length:]
        return rest, true
    case .Table:
        tables: u32 = 0
        tables, bytes = parse_uleb_32(bytes)
        mod.table = make([]Table, tables)
        for i in 0..

```

```

    }
    if flag.expressions {
        exprs: u32 = 0
        if flag_byte == 5 || flag_byte == 7 ||
flag_byte == 6 do elem.type, bytes = parse_ref_type(mod, bytes)
        exprs, bytes = parse_uleb_32(bytes)
        elem.exprs = make([][]Instruction, exprs)
        for expri in 0..

```

```

case .i64_const:
    c: i64 = 0
    c, bytes = parse_leb_64(bytes)
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = c
    append(&instrs, instr)
case .i32_const:
    c: i32 = 0
    c, bytes = parse_leb_32(bytes)
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = c
    append(&instrs, instr)
case .f32_const:
    b := cast(^f32)&bytes[0]
    bytes = bytes[4:]
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = b^
    append(&instrs, instr)
case .f64_const:
    b := cast(^f64)&bytes[0]
    bytes = bytes[8:]
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = b^
    append(&instrs, instr)
case .br_table:
    count: u32 = 0
    default: u32 = 0
    count, bytes = parse_uleb_32(bytes)
    labels := make([]u32, count)
    for i in 0..

```

```

        append(&instrs, instr)
    case .ref_null:
        rt: RefType = {}
        rt, bytes = parse_ref_type(mod, bytes)
        rtt: Maybe(Type) = rt
        instr := Instruction { opcode = opcode, argc =
1 }

        instr.args[0] = rtt
        append(&instrs, instr)

case .local_set, .local_get, .local_tee, .br, .br_if, .global_ge
t, .global_set, .call, .memory_fill, .memory_size, .memory_grow,
.data_drop, .ref_func, .elem_drop, .table_get, .table_fill, .tab
le_set:
        c: u32 = 0
        c, bytes = parse_uleb_32(bytes)
        instr := Instruction { opcode = opcode, argc =
1 }

        instr.args[0] = c
        append(&instrs, instr)

case .nop, .i32_gt_s, .i32_ge_s, .i32_ge_u, .i32_gt_u, .i32_sub,
.i32_add, .i32_mul, .i32_rem_u, .i32_rem_s, .i32_div_s, .i32_div
_u, .return, .i32_shl, .i32_shr_u, .i32_rotl, .i32_rotr, .i64_r
otl, .i64_rotr, .i32_lt_s, .i32_lt_u, .i32_le_s, .i32_le_u, .sele
ct, .i64_gt_s, .i64_ge_s, .i64_ge_u, .i64_rem_u, .i64_rem_s, .i6
4_div_s, .i64_div_u, .i64_le_s, .i64_le_u, .i64_add, .i64_sub, .
i64_shl, .i64_ne, .i64_and, .i64_gt_u, .i64_extend_i32_s, .i32_a
nd, .i32_xor, .i64_xor, .i64_or, .i32_or, .i32_ne, .i32_eqz, .i6
4_eqz, .i64_eq, .i32_wrap_i64, .i32_shr_s, .i64_lt_s, .i64_lt_u,
.unreachable, .drop, .i64_mul, .i32_eq, .i64_extend_i32_u, .i64_
extend8_s, .i64_extend16_s, .i64_extend32_s, .i64_shr_u, .i64_sh
r_s, .i32_popcnt, .i32_ctz, .i32_clz, .i64_ctz, .i64_clz, .i32_e
xtend8_s, .i32_extend16_s, .f64_convert_i32_u, .f64_convert_i32_
s, .f64_convert_i64_s, .f64_convert_i64_u, .f64_add, .f64_mul, .
f64_sub, .f64_gt, .f64_ge, .f64_eq, .f64_lt, .f64_le, .f32_sqrt,
.f32_gt, .f32_ge, .f32_lt, .f32_le, .f32_eq, .f32_sub, .f32_add,
.f32_mul, .f32_div, .f64_reinterpret_i64, .f32_reinterpret_i32,
.i64_popcnt, .f32_min, .f32_max, .f32_floor, .f32_ceil, .f32_tru
nc, .f32_nearest, .f64_min, .f64_max, .f64_floor, .f64_ceil, .f6
4_trunc, .f64_nearest, .f64_div, .f64_sqrt, .f32_copysign, .f64_
copysign, .f32_abs, .f64_abs, .f32_neg, .f64_neg, .f32_ne, .f64_
ne, .ref_is_null, .i64_reinterpret_f64, .f32_demote_f64, .f64_pr
omote_f32, .f32_convert_i32_s, .f32_convert_i32_u, .f32_convert_
i64_s, .f32_convert_i64_u, .i32_trunc_f64_s, .i32_trunc_f64_u, .
i64_trunc_f32_s, .i64_trunc_f32_u, .i64_trunc_f64_s, .i64_trunc_
f64_u, .i32_trunc_f32_s, .i32_trunc_f32_u, .i32_reinterpret_f32,
.i32_trunc_sat_f32_s, .i32_trunc_sat_f32_u, .i32_trunc_sat_f64_s
, .i32_trunc_sat_f64_u, .i64_trunc_sat_f32_s, .i64_trunc_sat_f32
_u, .i64_trunc_sat_f64_s, .i64_trunc_sat_f64_u:
        instr := Instruction { opcode = opcode, argc =
0 }

        append(&instrs, instr)

```

```

case .f64_store, .f64_load, .i64_store, .i64_store32, .i64_store
16, .i64_store8, .i64_load32_s, .i64_load8_u, .i64_load8_s, .i64
_load32_u, .i32_store8, .i64_load, .i32_load8_u, .i32_load8_s, .
i32_store16, .i32_load16_u, .i32_load16_s, .i32_store, .i32_load
, .i64_load16_u, .i64_load16_s, .f32_store, .f32_load:
    align: u32 = 0
    align, bytes = parse_uleb_32(bytes)
    memidx: u32 = 0
    if align & (1 << 6) != 0 {
        memidx, bytes = parse_uleb_32(bytes)
    }
    offset: u64 = 0
    offset, bytes = parse_uleb_64(bytes)
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = Memarg {
        offset = offset,
        align = 1 << (align & 0b111),
        memidx = memidx,
    }
    append(&instrs, instr)
case .loop, ._if, .block:
    type: Maybe(Type) = nil
    type, bytes = parse_block_type(mod, bytes)
    instr := Instruction { opcode = opcode, argc =
2 }

    instr.args[0] = type
    append(&instrs, instr)
    append(&block_stack, u32(len(instrs) - 1))
case ._else:
    block := block_stack[len(block_stack) - 1]
    instrs[block].args[2] = u32(len(instrs))
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = block
    append(&instrs, instr)
case .end:
    if len(block_stack) == 0 { stop = true ; break }
    block := pop(&block_stack)
    instrs[block].args[1] = u32(len(instrs))
    if v, ok := instrs[block].args[2].(u32); ok &&
v != 0 {
        instrs[v].args[1] = u32(len(instrs))
        instrs[v].argc = 2
    }
    instr := Instruction { opcode = opcode, argc =
1 }

    instr.args[0] = block
    append(&instrs, instr)
case:
    printf("%v\n", opcode)
    panic("todo")

```



```
        }  
    }  
    return instrs[:], bytes  
}
```

Додаток В

Лістинг вихідного коду Wasm-драйвера файлової системи ext2

```

package ext2

import kapi "../kapi"

// Ext2 file-type constants (dirent d_type field)
FT_UNKNOWN :: u8(0)
FT_REG      :: u8(1)
FT_DIR      :: u8(2)

// DirEntryType values that the kernel expects in
WasmDirent.type
DT_UNKNOWN :: i32(0)
DT_DIR     :: i32(4)
DT_REG     :: i32(8)

// Seek whence values
SEEK_SET :: i32(0)

// Driver-wide state (one mount at a time)
blkfd:      i32 = -1
block_size: u32
inode_size:  u32
inodes_per_group: u32
blocks_per_group: u32
inodes_count:  u32
blocks_count:  u32
bgdt_block:    u32 // first block of the block-group
descriptor table

// Internal scratch buffers – never overlap with the registered
scratch
blk_buf:  [4096]u8 // one full block
inode_buf: [256]u8  // one inode
bgdt_buf:  [32]u8   // one BGDT entry
indir_buf: [4096]u8 // single-indirect block

// Registered scratch: kernel copies paths here; driver writes
stat/dirent here.
// Must be >= max(max_path, sizeof(WasmDirent)) = 512 is plenty.
scratch: [512]u8

Superblock :: struct #packed {
    s_inodes_count:  u32, // 0
    s_blocks_count:  u32, // 4
    s_r_blocks:      u32, // 8
    s_free_blocks:    u32, // 12
    s_free_inodes:    u32, // 16

```

```

s_first_data:      u32, // 20
s_log_block_size:  u32, // 24
s_log_frag_size:   u32, // 28
s_blocks_per_group: u32, // 32
s_frags_per_group:  u32, // 36
s_inodes_per_group: u32, // 40
s_mtime:           u32, // 44
s_wtime:           u32, // 48
s_mnt_count:       u16, // 52
s_max_mnt_count:   u16, // 54
s_magic:           u16, // 56
s_state:           u16, // 58
s_errors:          u16, // 60
s_minor_rev:       u16, // 62
s_lastcheck:       u32, // 64
s_checkinterval:   u32, // 68
s_creator_os:      u32, // 72
s_rev_level:       u32, // 76
s_def_resuid:       u16, // 80
s_def_resgid:       u16, // 82
s_first_ino:       u32, // 84
s_inode_size:      u16, // 88
}

Inode :: struct #packed {
    i_mode:      u16, // 0
    i_uid:       u16, // 2
    i_size:      u32, // 4
    i_atime:     u32, // 8
    i_ctime:     u32, // 12
    i_mtime:     u32, // 16
    i_dtime:     u32, // 20
    i_gid:       u16, // 24
    i_links:     u16, // 26
    i_blocks_512: u32, // 28
    i_flags:     u32, // 32
    i_osdl:      u32, // 36
    i_block:     [15]u32, // 40 (12 direct + 1 single + 1
double + 1 triple indirect)
    i_generation: u32, // 100
    i_file_acl:   u32, // 104
    i_size_high:  u32, // 108 (upper 32 bits of size for large
files)
}

read_bytes :: proc "contextless" (offset: i64, buf: []u8) ->
bool {
    if kapi.vfs_lseek(blkfd, offset, SEEK_SET) != offset do
return false
    return kapi.vfs_read(blkfd, buf) == i32(len(buf))
}

```

```

write_bytes :: proc "contextless" (offset: i64, buf: []u8) ->
bool {
    if kapi.vfs_lseek(blkfd, offset, SEEK_SET) != offset do
return false
    return kapi.vfs_write(blkfd, buf) == i32(len(buf))
}

load_inode :: proc "contextless" (ino: i64) -> bool {
    group := u32(ino - 1) / inodes_per_group
    local := u32(ino - 1) % inodes_per_group
    bgdt_off := i64(bgdt_block) * i64(block_size) + i64(group) *
32
    if !read_bytes(bgdt_off, bgdt_buf[:32]) do return false
    inode_table := (^u32)(&bgdt_buf[8])^
    disk_off := i64(inode_table) * i64(block_size) + i64(local)
* i64(inode_size)
    return read_bytes(disk_off, inode_buf[:inode_size])
}

write_inode :: proc "contextless" (ino: i64) -> bool {
    group := u32(ino - 1) / inodes_per_group
    local := u32(ino - 1) % inodes_per_group
    bgdt_off := i64(bgdt_block) * i64(block_size) + i64(group) *
32
    if !read_bytes(bgdt_off, bgdt_buf[:32]) do return false
    inode_table := (^u32)(&bgdt_buf[8])^
    disk_off := i64(inode_table) * i64(block_size) +
i64(local) * i64(inode_size)
    return write_bytes(disk_off, inode_buf[:inode_size])
}

// Allocate a free block; marks it used and decrements the BGDT
free count.
// Returns the absolute block number, or 0 on failure.
alloc_block :: proc "contextless" () -> u32 {
    num_groups := (blocks_count + blocks_per_group - 1) /
blocks_per_group
    for g in u32(0)..<num_groups {
        bgdt_off := i64(bgdt_block) * i64(block_size) + i64(g) *
32
        if !read_bytes(bgdt_off, bgdt_buf[:32]) do continue
        free_blks := (^u16)(&bgdt_buf[12])^
        if free_blks == 0 do continue
        bitmap_blk := (^u32)(&bgdt_buf[0])^
        if !read_bytes(i64(bitmap_blk) * i64(block_size),
blk_buf[:block_size]) do continue
        for bi in u32(0)..<block_size {
            if blk_buf[bi] == 0xFF do continue
            for bit in u32(0)..<8 {
                if blk_buf[bi] & (u8(1) << u8(bit)) == 0 {
                    blk_num := g * blocks_per_group + bi * 8
+ bit
                    blk_buf[bi] |= u8(1) << u8(bit)

```

```

        if !write_bytes(i64(bitmap_blk) *
i64(block_size), blk_buf[:block_size]) do return 0
            (^u16)(&bgdt_buf[12])^ = free_blks - 1
            _ = write_bytes(bgdt_off, bgdt_buf[:32])
            return blk_num
        }
    }
}
return 0
}

// Allocate a free inode; marks it used and decrements the BGDT
// free count.
// Returns the inode number (1-based), or 0 on failure.
alloc_inode :: proc "contextless" () -> i64 {
    num_groups := (inodes_count + inodes_per_group - 1) /
inodes_per_group
    for g in u32(0)..<num_groups {
        bgdt_off := i64(bgdt_block) * i64(block_size) + i64(g) *
32
        if !read_bytes(bgdt_off, bgdt_buf[:32]) do continue
        free_inos := (^u16)(&bgdt_buf[14])^
        if free_inos == 0 do continue
        bitmap_blk := (^u32)(&bgdt_buf[4])^
        if !read_bytes(i64(bitmap_blk) * i64(block_size),
blk_buf[:block_size]) do continue
        for bi in u32(0)..<(inodes_per_group / 8) {
            if blk_buf[bi] == 0xFF do continue
            for bit in u32(0)..<8 {
                if blk_buf[bi] & (u8(1) << u8(bit)) == 0 {
                    ino_num := i64(g * inodes_per_group + bi
* 8 + bit + 1)
                    blk_buf[bi] |= u8(1) << u8(bit)
                    if !write_bytes(i64(bitmap_blk) *
i64(block_size), blk_buf[:block_size]) do return 0
                        (^u16)(&bgdt_buf[14])^ = free_inos - 1
                        _ = write_bytes(bgdt_off, bgdt_buf[:32])
                        return ino_num
                    }
                }
            }
        }
    }
    return 0
}

// Returns the on-disk block number for block index blk_idx
// within a file.
// Returns 0 for sparse blocks or if indirection is unsupported.
file_block :: proc "contextless" (inode: ^Inode, blk_idx: u32)
-> u32 {
    ptrs := block_size / 4
    if blk_idx < 12 do return inode.i_block[blk_idx]

```

```

    idx := blk_idx - 12
    // Single indirect
    if idx < ptrs {
        indir := inode.i_block[12]
        if indir == 0 do return 0
        if !read_bytes(i64(indir) * i64(block_size),
indir_buf[:block_size]) do return 0
        return ([^]u32)(&indir_buf[0])[idx]
    }
    idx -= ptrs
    // Double indirect
    if idx < ptrs * ptrs {
        dindir := inode.i_block[13]
        if dindir == 0 do return 0
        if !read_bytes(i64(dindir) * i64(block_size),
indir_buf[:block_size]) do return 0
        l1 := ([^]u32)(&indir_buf[0])[idx / ptrs]
        if l1 == 0 do return 0
        // Reuse blk_buf for second level to avoid clobbering
indir_buf
        if !read_bytes(i64(l1) * i64(block_size),
blk_buf[:block_size]) do return 0
        return ([^]u32)(&blk_buf[0])[idx % ptrs]
    }
    return 0 // triple indirect not implemented
}

// Looks up a single name component in a directory inode.
Returns inode or 0.
dir_lookup :: proc "contextless" (dir_ino: i64, name: []u8) ->
i64 {
    if !load_inode(dir_ino) do return 0
    inode := (^Inode)(&inode_buf[0])
    dir_size := u32(inode.i_size)
    nblocks := (dir_size + block_size - 1) / block_size

    for bi in u32(0)..<nblocks {
        blkno := file_block(inode, bi)
        if blkno == 0 do continue
        if !read_bytes(i64(blkno) * i64(block_size),
blk_buf[:block_size]) do continue

        off: u32 = 0
        for off + 8 <= block_size {
            d_ino      := (^u32)(&blk_buf[off])^
            rec_len    := (^u16)(&blk_buf[off+4])^
            name_len := blk_buf[off+6]
            if rec_len == 0 do break
            if d_ino != 0 && int(name_len) == len(name) {
                match := true
                for i in 0..<int(name_len) {
                    if blk_buf[off+8+u32(i)] != name[i] { match
= false; break }

```

```

        }
        if match do return i64(d_ino)
    }
    off += u32(rec_len)
}
}
return 0
}

// Walks a VFS-relative path (no leading slash, may have
internal slashes).
// Returns inode number or negative errno (-2 = ENOENT).
path_lookup :: proc "contextless" (path: []u8) -> i64 {
    cur := i64(2) // ext2 root inode is always 2
    i := 0
    for i < len(path) {
        for i < len(path) && path[i] == '/' { i += 1 }
        if i >= len(path) do break
        j := i
        for j < len(path) && path[j] != '/' { j += 1 }
        component := path[i:j]
        if len(component) == 0 || (len(component) == 1 &&
component[0] == '.') {
            i = j
            continue
        }
        next := dir_lookup(cur, component)
        if next == 0 do return -2
        cur = next
        i = j
    }
    return cur
}

// Returns (parent_ino, last_component). parent_ino < 0 on
error.
path_lookup_parent :: proc "contextless" (path: []u8) ->
(parent_ino: i64, name: []u8) {
    last_slash := -1
    for i in 0..<len(path) {
        if path[i] == '/' do last_slash = i
    }
    if last_slash < 0 do return i64(2), path
    name = path[last_slash+1:]
    if last_slash == 0 do return i64(2), name
    parent_ino = path_lookup(path[:last_slash])
    return parent_ino, name
}

// Add a directory entry for (new_ino, name, file_type) to
directory dir_ino.
// Splits an existing entry's padding or allocates a new block.

```

```

dir_add_entry :: proc "contextless" (dir_ino: i64, new_ino: i64,
name: []u8, file_type: u8) -> bool {
    if !load_inode(dir_ino) do return false
    dir_inode := (^Inode)(&inode_buf[0])^ // stack copy;
inode_buf is reused below

    dir_size := u32(dir_inode.i_size)
    nblocks := (dir_size + block_size - 1) / block_size
    new_rec := u16((8 + len(name) + 3) &~ 3)

    for bi in u32(0)..<nblocks {
        blkno := file_block(&dir_inode, bi)
        if blkno == 0 do continue
        if !read_bytes(i64(blkno) * i64(block_size),
blk_buf[:block_size]) do continue

        off: u32 = 0
        for off + 8 <= block_size {
            d_ino := (^u32)(&blk_buf[off])^
            rec_len := (^u16)(&blk_buf[off+4])^
            if rec_len == 0 do break
            if d_ino == 0 {
                // Deleted slot: reuse if large enough
                if rec_len >= new_rec {
                    (^u32)(&blk_buf[off])^ = u32(new_ino)
                    (^u16)(&blk_buf[off+4])^ = rec_len
                    blk_buf[off+6] = u8(len(name))
                    blk_buf[off+7] = file_type
                    for i in 0..<len(name)
{ blk_buf[off+8+u32(i)] = name[i] }
                    return write_bytes(i64(blkno) *
i64(block_size), blk_buf[:block_size])
                }
            } else {
                name_len := blk_buf[off+6]
                min_rec := u16((8 + int(name_len) + 3) &~ 3)
                spare := rec_len - min_rec
                if spare >= new_rec {
                    (^u16)(&blk_buf[off+4])^ = min_rec
                    new_off := off + u32(min_rec)
                    (^u32)(&blk_buf[new_off])^ = u32(new_ino)
                    (^u16)(&blk_buf[new_off+4])^ = spare
                    blk_buf[new_off+6] = u8(len(name))
                    blk_buf[new_off+7] = file_type
                    for i in 0..<len(name)
{ blk_buf[new_off+8+u32(i)] = name[i] }
                    return write_bytes(i64(blkno) *
i64(block_size), blk_buf[:block_size])
                }
            }
            off += u32(rec_len)
        }
    }
}

```



```

    // No room in existing blocks - allocate a new directory
    block
    if nblocks >= 12 do return false // TODO: indirect
    directory blocks
    new_blkno := alloc_block()
    if new_blkno == 0 do return false

    for i in 0..<int(block_size) { blk_buf[i] = 0 }
    (^u32)(&blk_buf[0])^ = u32(new_ino)
    (^u16)(&blk_buf[4])^ = u16(block_size)
    blk_buf[6]          = u8(len(name))
    blk_buf[7]          = file_type
    for i in 0..<len(name) { blk_buf[8+u32(i)] = name[i] }
    if !write_bytes(i64(new_blkno) * i64(block_size),
    blk_buf[:block_size]) do return false

    // Reload dir inode (blk_buf was reused by alloc_block),
    then update its block list
    if !load_inode(dir_ino) do return false
    inode := (^Inode)(&inode_buf[0])
    inode.i_block[nblocks] = new_blkno
    inode.i_size            = (nblocks + 1) * block_size
    inode.i_blocks_512      += block_size / 512
    return write_inode(dir_ino)
}

@export
fs_mount :: proc "c" (src_ptr: i32, src_len: i32) -> i64 {
    src := scratch[:src_len]
    fd := kapi.vfs_open(string(src), kapi.O_RDWR)
    if fd < 0 do fd = kapi.vfs_open(string(src), kapi.O_RDONLY)
    kapi.printf("ext2: opened %s %i\n", string(src), fd)
    if fd < 0 do return -5

    kapi.printf("ext: 1\n")
    // Read superblock (1024 bytes into the image)
    sb_buf: [90]u8
    if kapi.vfs_lseek(fd, 1024, SEEK_SET) != 1024 {
        kapi.vfs_close(fd)
        return -5
    }
    kapi.printf("ext: 2\n")
    if kapi.vfs_read(fd, sb_buf[:]) != 90 {
        kapi.vfs_close(fd)
        return -5
    }
    kapi.printf("ext: 3\n")
    sb := (^Superblock)(&sb_buf[0])
    kapi.printf("ext: %x\n", int(sb.s_magic))
    if sb.s_magic != 0xEF53 {
        kapi.vfs_close(fd)

```

```

        return -5
    }
    kapi.printf("ext: 4\n")

    blkfd          = fd
    block_size      = u32(1024) << sb.s_log_block_size
    inodes_per_group = sb.s_inodes_per_group
    blocks_per_group = sb.s_blocks_per_group
    inodes_count     = sb.s_inodes_count
    blocks_count     = sb.s_blocks_count
    if sb.s_rev_level >= 1 {
        inode_size = u32(sb.s_inode_size)
    } else {
        inode_size = 128
    }
    // BGDt is at block 1 for block_size > 1024, or block 2 for
1024-byte blocks
    bgdt_block = block_size > 1024 ? 1 : 2

    return 2 // root inode is always 2
}

@export
fs_open :: proc "c" (path_ptr: i32, path_len: i32, flags: i32,
mode: i32) -> i64 {
    if blkfd < 0 do return -5
    path := scratch[:path_len]
    ino  := path_lookup(path)

    if ino > 0 {
        if flags & kapi.O_CREATE != 0 && flags & 0x80 != 0 do
return -17 // O_EXCL: EEXIST
        return ino
    }
    if flags & kapi.O_CREATE == 0 do return ino // propagate
error

    parent_ino, name := path_lookup_parent(path)
    if parent_ino <= 0 || len(name) == 0 do return -2

    new_ino := alloc_inode()
    if new_ino <= 0 do return -28 // ENOSPC

    for i in 0..<int(inode_size) { inode_buf[i] = 0 }
    inode := (^Inode)(&inode_buf[0])
    inode.i_mode = 0x8000 | u16(mode & 0x0FFF)
    inode.i_links = 1
    if !write_inode(new_ino) do return -5

    if !dir_add_entry(parent_ino, new_ino, name, FT_REG) do
return -5
    return new_ino

```

```

}

@export
fs_stat :: proc "c" (ino: i64, stat_ptr: i32) -> i32 {
    if blkfd < 0 || !load_inode(ino) do return -5
    inode := (^Inode)(&inode_buf[0])
    size  := i64(inode.i_size) | (i64(inode.i_size_high) << 32)
    mode  := inode.i_mode

    file_type: i32
    switch mode & 0xF000 {
    case 0x4000: file_type = DT_DIR
    case 0x8000: file_type = DT_REG
    case:       file_type = DT_UNKNOWN
    }

    st := (^kapi.WasmStat)(rawptr(uintptr(stat_ptr)))
    st.ino = ino
    st.size = size
    st.mode = i32(mode & 0x0FFF)
    st.type = file_type
    st.uid = i32(inode.i_uid)
    st.gid = i32(inode.i_gid)
    return 0
}

@export
fs_read :: proc "c" (ino: i64, offset: i64, buf_ptr: i32, count:
i32) -> i32 {
    if blkfd < 0 || !load_inode(ino) do return -5
    inode := (^Inode)(&inode_buf[0])
    file_size := i64(inode.i_size) | (i64(inode.i_size_high) <<
32)
    if offset >= file_size do return 0

    rem := int(min(i64(count), file_size - offset))
    out := ([^]u8)(uintptr(buf_ptr))
    total := 0

    for total < rem {
        pos      := offset + i64(total)
        blk_idx   := u32(pos / i64(block_size))
        blk_off   := int(pos % i64(block_size))
        to_copy   := min(rem - total, int(block_size) - blk_off)
        blkno     := file_block(inode, blk_idx)

        if blkno == 0 {
            for i in 0..<to_copy { out[total+i] = 0 }
        } else {
            if !read_bytes(i64(blkno) * i64(block_size),
blk_buf[:block_size]) do return -5

```

```

        for i in 0..

```

```

    file_size := i64(inode.i_size) | (i64(inode.i_size_high) <<
32)
    if new_end > file_size {
        inode.i_size      = u32(new_end & 0xFFFFFFFF)
        inode.i_size_high = u32(u64(new_end) >> 32)
    }
    for i in 0..<int(inode_size) { inode_buf[i] =
inode_copy[i] }
    if !write_inode(ino) do return -5
    return i32(total)
}

@export
fs_readdir :: proc "c" (ino: i64, index: i32, entry_ptr: i32) ->
i32 {
    if blkfd < 0 || !load_inode(ino) do return 0
    inode := (^Inode)(&inode_buf[0])
    dir_size := u32(inode.i_size)
    nblocks  := (dir_size + block_size - 1) / block_size
    cur_idx  := i32(0)
    ent      := (^kapi.WasmDirent)(rawptr(uintptr(entry_ptr)))

    for bi in u32(0)..<nblocks {
        blkno := file_block(inode, bi)
        if blkno == 0 do continue
        if !read_bytes(i64(blkno) * i64(block_size),
blk_buf[:block_size]) do continue

        off: u32 = 0
        for off + 8 <= block_size {
            d_ino      := (^u32)(&blk_buf[off])^
            rec_len    := (^u16)(&blk_buf[off+4])^
            name_len   := blk_buf[off+6]
            file_type  := blk_buf[off+7]
            if rec_len == 0 do break
            if d_ino != 0 {
                if cur_idx == index {
                    dt: i32
                    switch file_type {
                    case FT_DIR: dt = DT_DIR
                    case FT_REG: dt = DT_REG
                    case:      dt = DT_UNKNOWN
                    }
                    nlen := min(int(name_len), 255)
                    ent.ino      = i64(d_ino)
                    ent.namelen = i32(nlen)
                    ent.type    = dt
                    for i in 0..<nlen { ent.name[i] =
blk_buf[off+8+u32(i)] }
                    if nlen < 256 { ent.name[nlen] = 0 }
                    return 1
                }
            }
            cur_idx += 1
        }
    }
}

```

```

        }
        off += u32(rec_len)
    }
}
return 0
}

@export
fs_ref :: proc "c" (ino: i64) {}

@export
fs_unref :: proc "c" (ino: i64) {}

@export
driver_main :: proc "c" () {
    blkfd          = -1
    block_size      = 0
    inode_size      = 0
    inodes_per_group = 0
    blocks_per_group = 0
    inodes_count    = 0
    blocks_count    = 0
    bgdt_block      = 0
    kapi.fs_register("ext2", scratch[:])
}

```

Додаток Г

Лістинг вихідного коду драйвера клавіатури

```
package kbd
import "../kapi"

scratch: [4]u8
irq_slot: i32 = -1

@export
kbd_read :: proc "c" (buf_ptr: i32, count: i32) -> i32 {
    if irq_slot < 0 || count <= 0 do return 0
    buf := (cast([^]u8)uintptr(buf_ptr))[:count]
    return kapi.irq_read(irq_slot, buf)
}

@export
driver_main :: proc "c" () {
    irq_slot = kapi.irq_open(1, 0x60)
    if irq_slot < 0 {
        kapi.printf("kbd: irq_open failed: %i\n", irq_slot)
        return
    }
    kapi.devfs_register("kbd", scratch[:])
    kapi.print("kbd: ready\n")
}
```

Додаток Д

Лістинг вихідного коду драйвера TTY

```

package tty
import "../kapi"
import "base:runtime"

panic_proc :: proc (p: string, s: string, loc:
runtime.Source_Code_Location) -> ! {
    kapi.print(p)
    kapi.print(":")
    kapi.print(s)
    kapi.print("\n")
    unreachable()
}

cols:      i32
rows:      i32
cursor_x:  i32
cursor_y:  i32

FG :: u32(0xFFFFFFFF)
BG :: u32(0x000000)

MAX_COLS :: 256
MAX_ROWS :: 128
screen: [MAX_COLS * MAX_ROWS]u8

fb_pixels: []u32
fb_stride: i32
fb_w:      i32
fb_h:      i32

font_atlas: [128][FONT_W * FONT_H]u32

dirty_rows: [MAX_ROWS]bool

cooked_buf: i32 = -1

kbd_fd: i32 = -1

shift:      bool = false
ctrl:       bool = false
e0_prefix:  bool = false

line_buf: [256]u8
line_len: i32

fg_pgrp: i32

```



```
termios: [44]u8
```

```
icanon: bool
echo:    bool
isig:    bool
```

```
// ANSI/VT100 output escape-sequence parser state
EscState :: enum u8 { Normal, EscSeen, CSI }
esc_state:      EscState
esc_params:     [8]i32
esc_param_cnt:  int
esc_private:    bool
```

```
scratch: [4096]u8
```

```
init_font_atlas :: proc "contextless" () {
  for ch in 0..<128 {
    glyph := &font[ch]
    for row in 0..<FONT_H {
      bits := glyph[row]
      for col in 0..<FONT_W {
        font_atlas[ch][row * FONT_W + col] = FG if bits
& (u8(0x80) >> u8(col)) != 0 else BG
      }
    }
  }
}
```

```
tty_render_char :: proc "contextless" (cx, cy: i32, ch: u8) {
  src := font_atlas[ch & 0x7F][:]
  base := int(cy) * int(FONT_H) * int(fb_stride) + int(cx) *
int(FONT_W)
  for row in 0..<FONT_H {
    copy(fb_pixels[base + row * int(fb_stride):][:FONT_W],
      src[row * FONT_W:][:FONT_W])
  }
  dirty_rows[cy] = true
}
```

```
tty_flush_dirty :: proc "contextless" () {
  for cy in i32(0)..<rows {
    if !dirty_rows[cy] do continue
    dirty_rows[cy] = false
    y := cy * i32(FONT_H)
    start := int(y) * int(fb_stride)
    kapi.fb_blit(0, y, fb_stride, i32(FONT_H),
      fb_pixels[start : start + int(fb_stride) *
FONT_H])
  }
}
```

```

tty_flush_all :: proc "contextless" () {
    kapi.fb_blit(0, 0, fb_stride, fb_h, fb_pixels)
    for cy in i32(0)..<rows { dirty_rows[cy] = false }
}

tty_scroll_up :: proc "contextless" () {
    row_px := int(fb_stride) * FONT_H
    copy(fb_pixels, fb_pixels[row_px:])
    for i in len(fb_pixels) - row_px..<len(fb_pixels)
{ fb_pixels[i] = BG }
    n := int(cols * (rows - 1))
    for i in 0..<n { screen[i] = screen[i + int(cols)] }
    for i in n..<int(cols * rows) { screen[i] = ' ' }
    cursor_y -= 1
    tty_flush_all()
}

tty_clear :: proc "contextless" () {
    for i in 0..<int(cols * rows) { screen[i] = ' ' }
    cursor_x = 0
    cursor_y = 0
    for i in 0..<len(fb_pixels) { fb_pixels[i] = BG }
    tty_flush_all()
}

tty_erase_to_eol :: proc "contextless" () {
    for x := cursor_x; x < cols; x += 1 {
        screen[int(cursor_y)*int(cols)+int(x)] = ' '
        tty_render_char(x, cursor_y, ' ')
    }
    dirty_rows[cursor_y] = true
}

tty_erase_to_sol :: proc "contextless" () {
    for x := i32(0); x <= cursor_x; x += 1 {
        screen[int(cursor_y)*int(cols)+int(x)] = ' '
        tty_render_char(x, cursor_y, ' ')
    }
    dirty_rows[cursor_y] = true
}

tty_erase_line :: proc "contextless" (y: i32) {
    for x := i32(0); x < cols; x += 1 {
        screen[int(y)*int(cols)+int(x)] = ' '
        tty_render_char(x, y, ' ')
    }
    dirty_rows[y] = true
}

csi_param :: proc "contextless" (idx: int, def: i32) -> i32 {

```

```

        if idx >= esc_param_cnt || idx >= len(esc_params) { return
def }
        v := esc_params[idx]
        return def if v == 0 else v
    }

_tty_csi_feed :: proc "contextless" (ch: u8) {
    switch ch {
    case '?':
        esc_private = true
        return
    case '0'..'9':
        if esc_param_cnt == 0 { esc_param_cnt = 1 }
        idx := esc_param_cnt - 1
        if idx < len(esc_params) {
            esc_params[idx] = esc_params[idx]*10 + i32(ch - '0')
        }
        return
    case ';':
        if esc_param_cnt == 0 { esc_param_cnt = 1 }
        esc_param_cnt += 1
        if esc_param_cnt - 1 < len(esc_params) {
            esc_params[esc_param_cnt - 1] = 0
        }
        return
    }

    defer { esc_state = .Normal }

    switch ch {
    case 'H', 'f':
        cursor_y = clamp(csi_param(0, 1) - 1, 0, rows - 1)
        cursor_x = clamp(csi_param(1, 1) - 1, 0, cols - 1)
    case 'A': cursor_y = max(0, cursor_y - csi_param(0,
1))
    case 'B': cursor_y = min(rows - 1, cursor_y + csi_param(0,
1))
    case 'C': cursor_x = min(cols - 1, cursor_x + csi_param(0,
1))
    case 'D': cursor_x = max(0, cursor_x - csi_param(0,
1))
    case 'G': cursor_x = clamp(csi_param(0, 1) - 1, 0, cols - 1)
    case 'd': cursor_y = clamp(csi_param(0, 1) - 1, 0, rows - 1)
    case 'J':
        switch csi_param(0, 0) {
        case 0:
            tty_erase_to_eol()
            for y := cursor_y + 1; y < rows; y += 1
{ tty_erase_line(y) }
        case 1:
            for y := i32(0); y < cursor_y; y += 1
{ tty_erase_line(y) }
            tty_erase_to_sol()

```

```

        case 2:
            sx := cursor_x; sy := cursor_y
            tty_clear()
            cursor_x = sx; cursor_y = sy
        }
    case 'K':
        switch csi_param(0, 0) {
        case 0: tty_erase_to_eol()
        case 1: tty_erase_to_sol()
        case 2: tty_erase_line(cursor_y)
        }
    case 'h', 'l', 'm', 'r', 's', 'u': // no-op
    }
}

tty_putchar :: proc "contextless" (ch: u8) {
    switch esc_state {
    case .Normal:
        switch ch {
        case 0x1B: esc_state = .EscSeen
        case '\n':
            cursor_x = 0
            cursor_y += 1
            if cursor_y >= rows { tty_scroll_up() }
        case '\r': cursor_x = 0
        case 0x08:
            if cursor_x > 0 {
                cursor_x -= 1
            } else if cursor_y > 0 {
                cursor_y -= 1
                cursor_x = cols - 1
            } else do return
            screen[int(cursor_y)*int(cols)+int(cursor_x)] = ' '
            tty_render_char(cursor_x, cursor_y, ' ')
        case '\t':
            tab := (cursor_x + 8) & ~i32(7)
            for cursor_x < tab && cursor_x < cols {
                screen[int(cursor_y)*int(cols)+int(cursor_x)] =
                    ' '
                tty_render_char(cursor_x, cursor_y, ' ')
                cursor_x += 1
            }
            if cursor_x >= cols {
                cursor_x = 0
                cursor_y += 1
                if cursor_y >= rows { tty_scroll_up() }
            }
        case 0x20..=0x7E:
            screen[int(cursor_y)*int(cols)+int(cursor_x)] = ch
            tty_render_char(cursor_x, cursor_y, ch)
            cursor_x += 1
            if cursor_x >= cols {
                cursor_x = 0
            }
        }
    }
}

```

```

        cursor_y += 1
        if cursor_y >= rows { tty_scroll_up() }
    }
}
case .EscSeen:
    if ch == '[' {
        esc_state      = .CSI
        esc_param_cnt  = 0
        esc_params     = {}
        esc_private    = false
    } else {
        esc_state = .Normal
    }
case .CSI:
    _tty_csi_feed(ch)
}
}

termios_update :: proc "contextless" () {
    c_lflag := u32(termios[12]) |
        u32(termios[13]) << 8 |
        u32(termios[14]) << 16 |
        u32(termios[15]) << 24
    isig   = (c_lflag & 0x01) != 0 // ISIG
    icanon = (c_lflag & 0x02) != 0 // ICANON
    echo   = (c_lflag & 0x08) != 0 // ECHO
}

_write_esc :: proc "contextless" (b, c, d: u8) {
    kapi.waitbuf_write(cooked_buf, 0x1B)
    kapi.waitbuf_write(cooked_buf, '[')
    kapi.waitbuf_write(cooked_buf, b)
    if c != 0 { kapi.waitbuf_write(cooked_buf, c) }
    if d != 0 { kapi.waitbuf_write(cooked_buf, d) }
}

@export
tty0_input_loop :: proc "c" () {
    context = runtime.default_context()
    ctrl = false
    line_len = 0
    shift = false
    e0_prefix = false
    sc_buf: [1]u8
    for {
        for kapi.vfs_read(kbd_fd, sc_buf[:]) <= 0 {}
        sc := sc_buf[0]

        if sc == 0xE0 { e0_prefix = true; continue }

        rel := sc & 0x80 != 0
    }
}

```

```

key := int(sc & 0x7F)

if key == 0x2A || key == 0x36 { shift = !rel; continue }
if key == 0x1D { ctrl = !rel; continue }
if rel || key >= 128 { continue }

if e0_prefix {
    e0_prefix = false
    if !icanon {
        switch u8(key) {
            case 0x48: _write_esc('A', 0, 0)
            case 0x50: _write_esc('B', 0, 0)
            case 0x4D: _write_esc('C', 0, 0)
            case 0x4B: _write_esc('D', 0, 0)
            case 0x47: _write_esc('H', 0, 0)
            case 0x4F: _write_esc('F', 0, 0)
            case 0x53: _write_esc('3', '~', 0)
            case 0x49: _write_esc('5', '~', 0)
            case 0x51: _write_esc('6', '~', 0)
            case 0x52: _write_esc('2', '~', 0)
        }
    }
    continue
}

table := &sc_normal if !shift else &sc_shifted
ch := table[key]
if ch == 0 { continue }
if ctrl { ch = ch & 0x1F; if ch == 0 { continue } }

if isig && ch == 0x03 { // Ctrl-C = SIGINT
    if echo {
        tty_putchar('^')
        tty_putchar('C')
        tty_putchar('\r')
        tty_putchar('\n')
        tty_flush_dirty()
    }
    line_len = 0
    kapi.printf("killing %i\n", fg_pgrp)
    kapi.kill(fg_pgrp, kapi.SIGINT)
    continue
}

if !icanon {
    kapi.waitbuf_write(cooked_buf, ch)
    continue
}

// Canonical mode: line editing with optional echo.
switch ch {
case 0x08, 0x7F:

```

```

        if line_len > 0 {
            line_len -= 1
            if echo { tty_putchar(0x08); tty_flush_dirty() }
        }
    case '\r', '\n':
        if echo { tty_putchar('\n'); tty_flush_dirty() }
        for i in 0..

```

```

    return -kapi.ENOTTY
}

@export
driver_main :: proc "c" () {
    context = runtime.default_context()
    alloc := runtime.WASM_Allocator{}
    runtime.wasm_allocator_init(&alloc)
    context.allocator = runtime.wasm_allocator(&alloc)
    context.assertion_failure_proc = panic_proc

    fb_w      = kapi.fb_width()
    fb_h      = kapi.fb_height()
    fb_stride = fb_w
    cols      = fb_w / i32(FONT_W)
    rows      = fb_h / i32(FONT_H)

    fg_pgrp = 1

    fb_pixels = make([]u32, int(fb_w * fb_h))

    init_font_atlas()
    tty_clear()

    // Default termios: ISIG | ICANON | ECHO | ECHOE | ECHOK =
    0x3B
    termios[12] = 0x3B
    termios_update()

    cooked_buf = kapi.waitbuf_create()

    kbd_fd = kapi.vfs_open("/dev/kbd", kapi.O_RDONLY)
    if kbd_fd < 0 {
        kapi.printf("tty: failed to open /dev/kbd: %i\n",
kbd_fd)
        return
    }

    kapi.spawn_task("tty0_input_loop")
    kapi.devfs_register("tty0", scratch[:])
    kapi.print("tty: ready\n")
}

```