

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТИРЕНКО

«_____» _____ 2021 р.

Дипломний проєкт

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Комп'ютерні системи та мережі»
спеціальності 123 «Комп'ютерна інженерія»**

на тему: «Компоненти Linux-подібної операційної системи»

Виконав: студент 4 курсу, групи ІВ-71

Поляков Марк Сергійович

Керівник Порєв В.М.

Консультант (нормоконтроль) Симоненко В. П.

Рецензент _____

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент _____

Київ – 2021 р.

Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Факультет Інформатики та обчислювальної техніки
Кафедра обчислювальної техніки
Освітньо-кваліфікаційний рівень бакалавр
Напрямок підготовки 6.050102 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедри

С.Г. Стіренко

(підпис)

(ініціали, прізвище)

“ ___ ” _____ 20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Полякова Марка Сергійовича

1. Тема роботи «Компоненти Linux-подібної операційної системи», керівник проєкту Порєв Віктор Миколайович, к.т.н., старший викладач, затверджені наказом по університету від “ 11 ” травня 2021 р. № 1139-с

2. Термін подання студентом закінченого проєкту _____

3. Вихідні дані до проєкту: технічна документація;

4. Зміст проєкту: опис предметної області, дослідження, технічної бази, розробка ядра та компонентів операційної системи.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо): структурна схема взаємодії, принципова схема роботи системи, функціональна схема платформи цільової системи

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	д.т.н., проф. Сімоненко В.П.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітки
1	<i>Затвердження теми роботи</i>	<i>10.12.2020-15.12.2020</i>	
2	<i>Вивчення та аналіз завдання</i>	<i>15.12.2020-15.03.2021</i>	
3	<i>Написання вступної частини та огляду предметної частини</i>	<i>15.03.2021-25.03.2021</i>	
4	<i>Розробка архітектури системи</i>	<i>25.03.2021-5.04.2021</i>	
5	<i>Програмна реалізація системи</i>	<i>05.04.2021-15.04.2021</i>	
6	<i>Виправлення помилок</i>	<i>15.04.2021-20.05.2021</i>	
7	<i>Оформлення документації дипломної роботи</i>	<i>25.04.2021</i>	
8	<i>Передзахист</i>	<i>29.05.2021</i>	
9	<i>Захист</i>	<i>15.06.2021</i>	

Студент

Марк ПОЛЯКОВ

Керівник проєкту

Віктор ПОРЄВ

АНОТАЦІЯ

Ця робота являє собою дослідження та огляд різних компонентів операційних систем, притаманних Unix-подібним операційним системам. Робота складається з трьох основних частин, кожна з яких підрозділяється на набори менших компонентів: ядро, стандартна бібліотека мови програмування C та набір користувацьких програм. Проект спирається на ряд технік з імплементації та оптимізації для того, щоб зробити це ядро гідною базою для побудови операційної системи, при цьому зберігаючи простоту розробки та читабельність коду.

АННОТАЦИЯ

Эта работа представляет собой исследование и обзор различных компонентов операционных систем, свойственных Unix-подобным операционным системам. Работа состоит из трёх частей, каждая из которых подразделяется на наборы меньших компонентов: ядро, стандартная библиотека языка программирования C и набор пользовательских программ. Проект опирается на ряд техник по имплементации и оптимизации для того, чтобы сделать это ядро достойной базой для построения операционной системы, при этом сохраняя простоту и читаемость кода.

ABSTRACT

This work represents a research and review of different operating system components common to Unix-like operating systems. The work consists of three main parts, each subdivided into a set of subcomponents: the kernel, the C standard library and the user-space program set. The project employs a range of implementation and optimization techniques to make the kernel a viable option for operating system base while maintaining the ease of development and code readability at the same time.

Опис альбому до дипломного проекту

на тему: «Компоненти Linux-подібної операційної системи»

	Формат.	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ екзем.	Примітка
			<u>Документація загальна</u>			
			<u>Розроблена заново</u>			
1	A4	ІАЛЦ.467200.001 ОА	Опис альбому	1		
2	A4	ІАЛЦ.467200.002 ТЗ	Технічне завдання	4		
3	A4	ІАЛЦ.467200.003 ПЗ	Пояснювальна записка	78		
4	A4	ІАЛЦ.467200.004 Д1	Додаток А	1		
5	A4	ІАЛЦ.467200.005 Д2	Додаток Б	1		
6	A4	ІАЛЦ.467200.006 Д3	Додаток В	1		
7	A4	ІАЛЦ.467200.007 Д4	Додаток Г	16		
8	A4	ІАЛЦ.467200.008 Д5	Додаток Д	14		

					<i>ІАЛЦ.467200.002 ТЗ</i>			
Зм.	Арк.	№ докум	Підп.	Дата				
Розробив		Поляков М.С.			Компоненти Linux-подібної операційної системи	Літ.	Аркуш	Аркушів
Перевірів		Порєв В.М.				Т	І	
						НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71		
Н.контр.		Сімоненко В.П.						
Затв.		Порєв В.М.						

Технічне завдання до дипломного проекту

на тему: «Компоненти Linux-подібної операційної системи»

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ.....	2
2. ПІДСТАВИ ДЛЯ РОЗРОБКИ.....	2
3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
4. ДЖЕРЕЛА РОЗРОБКИ.....	2
5. ТЕХНІЧНІ ВИМОГИ.....	2
5.1. ВИМОГИ ДО ТЕХНІЧНОГО ПРОДУКТУ, ЩО РОЗРОБЛЯЄТЬСЯ.....	2
5.2. ВИМОГИ ДО ІНСТРУМЕНТАЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	3
5.3. ВИМОГИ ДО АПАРАТНОЇ ЧАСТИНИ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ.....	4
6. ЕТАПИ РОЗРОБКИ.....	4

					<i>ІАЛЦ.467200.002 ТЗ</i>			
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум</i>	<i>Підп.</i>	<i>Дата</i>				
<i>Розробив</i>		<i>Поляков М.С.</i>			<i>Компоненти Linux-подібної операційної системи</i>	<i>Літ.</i>	<i>Аркуш</i>	<i>Аркушів</i>
<i>Перевірів</i>		<i>Порєв В.М.</i>				<i>Т</i>	<i>І</i>	
						<i>НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71</i>		
<i>Н.контр.</i>		<i>Сімоненко В.П.</i>						
<i>Затв.</i>		<i>Порєв В.М.</i>						

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання описує операційне середовище, що являє собою ядро та набір користувацьких програм, зокрема систему ініціалізації, інтерфейс командного рядку, бібліотеки та інші. Областю застосування цього проєкту можуть слугувати системи сумісні з IBM PC на базі x86-64 процесорів AMD або Intel.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для виконання цього проєкту є завдання бакалаврської дипломної роботи, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою цього проєкту є створення мінімального, але достатнього набору програмних компонентів, що б слугували базою для операційної системи.

4. ДЖЕРЕЛА РОЗРОБКИ

В ході цієї роботи було використано різні джерела інформації, що включають в себе публікації в мережі Інтернет, монографії, наукову літературу, стандарти програмного забезпечення та специфікації апаратного забезпечення.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до технічного продукту, що розробляється

Далі надано основні вимоги до проєкту в цілому:

- Продукт має надати користувачеві мінімальний, але достатній набір програмних застосунків, щоб мати можливість взаємодіяти з виділеним ресурсом апаратної платформи через інтерфейс командного рядку;
- З перспективи користувача продукт має бути максимально наближеним до UNIX та специфікацій, що стандартизують подібні системи (POSIX), а також в своїй побудові керуватись їх принципами;

					<i>ІАЛЦ.467200.002 ТЗ</i>	Арк.
						2
Зм.	Арк.	№ докум.	Підп.	Дата		

- Потрібно надати супутні компоненти для збірки за допомогою компілятора C та утиліт від GNU програм для виконання в цьому середовищі, тобто надати просте середовище для крос-компіляції;

Виходячи з того, що проєкт складається з декількох частин, для кожної з них далі описано окремі вимоги.

Ядро цього середовища має:

- приймати керування системою від завантажувача, що підтримує необхідні протоколи завантаження;
- завантажувати програми з бінарних файлів;
- виконувати розподіл пам'яті, процесорного часу та інших ресурсів системи;
- абстрагувати драйвери резидентних файлових систем, взаємодію з апаратними та псевдо-пристроями через так звану "віртуальну" файлову систему;
- надати для користувацьких програм набір системних викликів для використання привілейованих функцій.

Програмне середовище, що буде виконуватись за допомогою цього ядра, мусить містити систему для ініціалізації, інтерфейс командного рядку та мінімальний набір програм-утиліт для роботи з системою. Для всіх програм має бути надано стандартну бібліотеку для мови C, що з сумісною зі стандартом POSIX.1-2017 (але не має реалізовувати його цілком) в мірі, достатній для розробки нового програмного забезпечення, або використання вже існуючих рішень.

5.2. Вимоги до інструментального програмного забезпечення

- компілятор C GNU GCC версії 9.3.0;
- набір GNU binutils версії 2.33.1;
- GNU Make;
- GNU GRUB версії 2 або новіше з xorriso;
- будь-який текстовий редактор для конфігурації завантажувача та подальшої розробки;

Для виконання у віртуалізованому або емульованому середовищі потрібно використовувати програми для віртуалізації платформи x86, що можуть реалізувати вказаний нижче набір апаратних вимог. Прикладом такого ПЗ є qemu або bochs.

					<i>ІАЛЦ.467200.002 ТЗ</i>	Арк.
						3
Зм.	Арк.	№ докум.	Підп.	Дата		

5.3. Вимоги до апаратної частини обчислювальної системи

- процесор Intel або AMD з підтримкою наступних технологій:
- AMD64 або Long Mode, наявна в AMD починаючи з мікроархітектури K8 та Intel починаючи з NetBurst;
- підтримка інструкцій SYSCALL та SYSRET;
- підтримка інструкцій FXSAVE та FXRSTOR;
- оперативна пам'ять об'ємом не менше 64 Мб;
- для взаємодії з командним рядком має бути наявна клавіатура з підтримкою PS/2;

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення літератури	15.12.2020
Складання і узгодження технічного завдання	25.01.2021
Розробка компонентів цільової системи	20.02.2021
Тестування компонентів роботи	15.03.2021
Виправлення помилок	15.04.2021
Оформлення документації до дипломної роботи	19.05.2021

					<i>ІАЛЦ.467200.002 ТЗ</i>	Арк.
						4
Зм.	Арк.	№ докум.	Підп.	Дата		

Пояснювальна записка до дипломного проекту

на тему: «Компоненти Linux-подібної операційної системи»

ЗМІСТ

ЗМІСТ.....	1
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	3
ВСТУП.....	5
РОЗДІЛ 1. ОГЛЯД ПОДІБНИХ РІШЕНЬ В НІШІ ОПЕРАЦІЙНИХ СИСТЕМ.....	8
1.1. Проблемно-орієнтовані ядра (Unikernel).....	8
1.2. Спільні принципи проблемно-орієнтованих операційних систем.....	9
1.3. Використання гіпервізорів в поєднанні з проблемно-орієнтованими ядрами.....	11
1.4. Приклади рішень на проблемно-орієнтованій архітектурі.....	12
1.5. Ядро Linux.....	14
1.5.1. Використання існуючого оточення.....	15
1.5.2. Розробка власного оточення.....	17
1.6. Розробка модулів ядра для Linux.....	18
1.7. Ключові відмінності монолітних ядер.....	20
ВИСНОВКИ ДО РОЗДІЛУ 1.....	21
РОЗДІЛ 2. АРХІТЕКТУРА СЕРЕДОВИЩА.....	22
2.1. Опис необхідних для побудови компонентів системи.....	22
2.2. Високорівневий опис взаємодії компонентів.....	23
2.3. Внутрішня архітектура ядра системи.....	25
2.4. Опис набору системних викликів.....	27
2.5. Віртуальна файлова система.....	31
2.5.1. Спеціальні файлові системи ядра.....	35
2.6. Архітектура стандартної бібліотеки мови C.....	36
2.7. Архітектура програмного оточення ядра.....	38
ВИСНОВКИ ДО РОЗДІЛУ 2.....	42
РОЗДІЛ 3. РОЗРОБКА КОМПОНЕНТІВ ОПЕРАЦІЙНОЇ СИСТЕМИ.....	43
3.1. Ядро системи.....	43
3.1.1. Основні етапи роботи ядра.....	43
3.1.2. Огляд менеджера фізичної пам'яті.....	45
3.1.3. Огляд менеджера віртуальної пам'яті.....	47
3.1.4. Робота з перериваннями та виключеннями процесора.....	48
3.1.5. Огляд драйверів пристроїв та файлових систем.....	50
3.1.6. Робота з процесами та потоками.....	59

					<i>ІАЛЦ.467200.003 ПЗ</i>							
Зм.	Арк.	№ докум	Підп.	Дата								
Розробив		Поляков М.С.			<i>Компоненти Linux-подібної операційної системи</i>			Літ.		Аркуш	Аркуші	
Перевірів		Порєв В.М.							Т		І	
								<i>НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71</i>				
Н.контр.		Сімоненко В.П.										
Затв.		Порєв В.М.										

3.1.7. Реалізація інтерфейсу системних викликів.....	62
3.2. Завантажувач операційної системи для UEFI.....	64
3.2.1. Протокол для завантажувача UEFI.....	65
3.2.2. Програма завантажувача для UEFI.....	67
3.3. Стандартна бібліотека мови C.....	68
3.3.1. Ініціалізація програми та бібліотеки.....	68
3.3.2. Робота з системними викликами.....	69
3.3.3. Робота з сигналами.....	69
3.3.4. Робота з динамічною пам'яттю.....	70
3.4. Програми користувацького оточення.....	71
3.4.1. Інтерфейс командного рядку.....	71
3.5. Демонстрація роботи результуючої системи.....	73
ВИСНОВКИ ДО РОЗДІЛУ 3.....	76
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	77

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						2
Зм.	Арк.	№ докум.	Підп.	Дата		

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACPI	Advanced Configuration and Power Interface — інтерфейс керування живленням та апаратним забезпеченням x86
AHCI	Advanced Host Controller Interface — стандарт хост-контролерів від Intel для підтримки SATA-дисків
APIC	Advanced Programmable Interrupt Controller — стандарт сучасного контролеру пріоритетних переривань для багатопроцесорних систем від Intel
AVX	Advanced Vector Extensions — набір інструкцій від Intel та AMD, що є нащадком SSE із 256-бітними регістрами
BIOS	Basic Input and Output System — системне програмне забезпечення, що виконується під час запуску комп'ютерних систем на платформі x86
CPL	Current Privilege Level — рівень привілей процесора
CoW	Copy-on-Write — техніка оптимізації копіювання даних
DMA	Direct Memory Access — можливість прямого доступу зовнішніх пристроїв до оперативної пам'яті системи
ELF	Executable and Linkable Format — формат виконуваних та об'єктних файлів, що є де-факто стандартом у Unix-подібних системах
IDE	Integrated Drive Electronics — найстаріша версія стандарту інтерфейсу для дискових носіїв, що була замінена ATA
IPI	Inter-Processor Interrupt — переривання, що пересилається між двома процесорами
MSI	Message-Signalled Interrupt — ефективний метод доставки переривань на процесори з шини PCI Express
PCI	Peripheral Component Interconnect — стандарт шини для підключення зовнішніх пристроїв

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						3
Зм.	Арк.	№ докум.	Підп.	Дата		

PE	Portable Executable — формат виконуваних файлів від Microsoft
PIC	Position-Independent Code — код, що зібраний незалежно від його планованого розміщення у пам'яті
POSIX	Portable Operating System Interface — сімейство стандартів від IEEE для надання сумісності між комп'ютерними системами
RAM	Random Access Memory — пам'ять з довільним доступом
RSDP	Root System Description Pointer — структура даних пам'яті, що описує базову інформацію про систему з підтримкою ACPI
RTC	Real-Time Clock — годинник реального часу
SATA	Serial AT Attachment — сучасний стандарт інтерфейсу підключення дискових пристроїв
SMP	Symmetric Multi-Processing — метод реалізації багатопроцесорних систем
SSE	Streaming SIMD Extensions — розширений набір векторних інструкцій від Intel та AMD
UEFI	Unified Extensible Firmware Interface — стандарт програмних інтерфейсів, що ставить за мету заміну застарілих методів завантаження BIOS
USB	Universal Serial Bus — стандарт шини для підключення зовнішніх пристроїв
Unikernel, libOS	Проблемно-орієнтовані бібліотечні операційні системи
VFS	Virtual File System — файлова система, що абстрагує конкретні драйвери від розробника та користувача
vnode	Вузол деякого об'єкту у віртуальній файловій системі

ВСТУП

На даний момент для реалізації певних низькорівневих продуктів є необхідною наявність мінімальної платформи, на базі якої ці технологічні рішення буде вирішено будувати. На сьогоднішній день можна явно спостерігати нішу для розробки, коли використання повноцінних та громіздких систем на кшталт Linux є завеликою витратою ресурсів або ж не дає такої можливості до модифікацій, а написання повноцінного рішення з нуля неможливе з міркувань часу або бюджету. На сьогоднішній день існують деякі рішення, що ставлять за мету вирішення цієї проблеми, але всі вони мають ряд переваг та недоліків, що залишають цю нішу відкритою до рішень. Виходячи з цього факту, актуальним є розробка програмного рішення, що буде за своєю технологічною складністю займати область між написанням ОС з нуля та використанням існуючих ядер. До того ж, надання вже знайомого користувачам та розробникам ПЗ стандартизованого функціоналу зможе значно прискорити інтеграцію такого продукту з їх цільовими рішеннями.

Одним відомим прикладом такого стандарту функціоналу є Єдина Специфікація UNIX, також відома як POSIX [1]. Цей стандарт було реалізовано в багатьох операційних системах через його простоту, відкритість до модифікацій та розширень, а також за можливості, які через цей інтерфейс надаються користувачу та розробнику. Така специфікація визначає ряд правил, що було в свій час встановлено в системі UNIX від корпорації AT&T, та що стали де-факто стандартом в комп'ютерній індустрії. Однією з причин на це можна виділити факт того, що цей стандарт є відкритим для публічного доступу, тобто будь-хто може реалізувати будь-який з компонентів, що так чи інакше надають функціонал, визначений в ньому, або його використовують.

Ця специфікація, крім набору правил, також покладає певну філософію розробки ядер та компонентів програмного забезпечення ОС. Одним з її основних елементів є прагнення до надання якомога більшого функціоналу ОС через файли. Тобто реальні або віртуальні пристрої, параметри системи та інші

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						5
Зм.	Арк.	№ докум.	Підп.	Дата		

значення надаються користувачеві як звичайні файли, що він, маючи необхідні права, зможе прочитати або записати, тим самим вносячи якісь зміни у стан системи. Такий концепт, відомий як "Все є файл", вимагає від операційної системи певної побудови, найвідомішим та найуспішнішим з таких рішень є концепція віртуальної файлової системи, що абстрагує якісь конкретні об'єкти та їх відповідні драйвери за сутностями звичайних файлів, директорій, або файлів-пристроїв. Також, система UNIX є багатокористувацькою, тому системи за її специфікацією мають надавати можливість для її одночасного використання декількома користувачами через так звані сесії, а також вводити контроль доступу для об'єктів. Система UNIX є багатозадачною системою, тому система на її базі мусить мати можливості для одночасного виконання багатьох процесів.

Крім підтримки з боку ядра, ця специфікація також визначає базовий набір утиліт, що входять до користувацького пакету та є необхідними для керування комп'ютерною системою. Такі програми можуть бути використані через інтерфейс командного рядку, що є також описаним цією специфікацією. Філософія UNIX вимагає від програм абсолютного мінімалізму — програма мусить виконувати лише найменший необхідний набір операцій, для якого вона створювалась. Також всі програми розроблюються так, щоб працювати над потоками даних, а інтерфейс командного рядку надає можливості будувати ланцюжки (конвеєри) з програм, виходи яких направлені на входи наступних. Ця можливість дає гнучку базу для використання та побудови рішень з точки зору користувача. Так, наприклад, маючи програми для виводу вмісту файлів (cat), сортування вхідних рядків за алфавітом (sort) та сторінкового виводу на дисплей (less або more) можна скомбінувати виклики цих утиліт так, щоби отримати на екрані розбитий на сторінки алфавітно відсортований за рядками вміст файлу.

При розробці нового проєкту операційного середовища використання вже існуючого та популярного стандарту програмного забезпечення означає можливість використання вже написаних раніше програм, якщо цей стандарт

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						6
Зм.	Арк.	№ докум.	Підп.	Дата		

реалізувати в достатній для цього мірі. Для цього програма може бути просто скомпільована відповідним інструментарієм з вихідного коду, в який у більшості випадків навіть не потрібно вносити ніякі зміни. Таким чином, наприклад, можна зібрати частини проєкту GNU, інтерпретатори різних скриптових мов, різні бібліотеки під абсолютно нову операційну систему без особливих змін до коду цих проєктів, якщо ця система є сумісною із стандартом.

Виходячи з вищенаведених переваг такого стандарту, він найкраще підійде для розробки продукту для цільової галузі. Таким чином користувач або розробник з мінімальним попереднім вивченням зможе використовувати вже звичні інтерфейси для розробки своїх рішень.

Ніша, яку займає цей проєкт, передбачає актуальність цієї роботи, бо розроблюване середовище є достатньо високорівневим для відносної простоти використання, але водночас дає змогу для внесення простих та швидких змін до будь-якого з його компонентів. Така простота також надає переваги в інтеграції цього продукту в інші програмні рішення. Для цього продукту буде розроблено чотири основні компоненти, які є незалежними один від іншого при збірці, але необхідними для побудови повноцінного операційного середовища: ядро, завантажувач, набір програмних компонентів, а також бібліотеки для їх побудови та інтеграції з ядром.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						7
Зм.	Арк.	№ докум.	Підп.	Дата		

РОЗДІЛ 1. ОГЛЯД ПОДІБНИХ РІШЕНЬ В НІШІ ОПЕРАЦІЙНИХ СИСТЕМ

1.1. Проблемно-орієнтовані ядра (Unikernel)

Нішу операційних систем «середнього рівня» в деякій частині займають так звані «Unikernel», або проблемно-орієнтовані ядра. Такі ядра мають дуже специфічну будову та надають достатній, але мінімальний набір функцій для коду, що буде з ними використовуватись. Як випливає з назви, такі ядра використовуються при побудові систем з конкретними специфікаціями та задачами. Такі ядра надають програмам спеціалізоване середовище з єдиним адресним простором оперативної пам'яті, а взаємодія користувацького коду з самим ядром проходить через «бібліотеку ОС», що необхідно приєднувати до коду проєкту. Першими представниками цього напрямку розробки є Exokernel та Nemesis, що були розроблені Массачусетським інститутом технологій та Університетом Кембриджу відповідно в кінці 90-х років. На даний момент існує ціла низка рішень у цій сфері, кожне з яких має свої переваги та недоліки.

Основним принципом таких систем є зменшення розміру коду, що є «носієм» для користувацького проєкту до абсолютно необхідного мінімуму. Цей принцип надає таким рішенням можливість до значного зменшення потенціальної площі для хакерських атак, бо непотрібний функціонал просто відсутній. Це є значною перевагою при побудові рішень для хмарних обчислень або проєктів, де безпека є критично важливою вимогою.

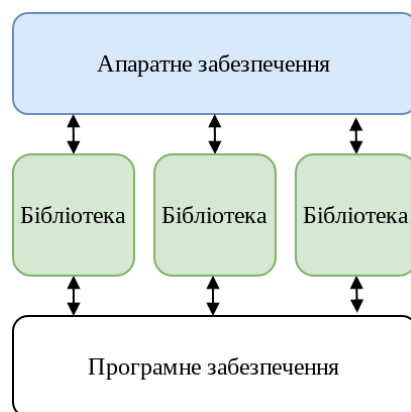


Рис. 1.1 Узагальнена архітектура libOS

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						8
Зм.	Арк.	№ докум.	Підп.	Дата		

На відміну від традиційних операційних систем, що визначають апаратну конфігурацію та завантажують відповідні драйвери та встановлюють необхідні налаштування під час старту, системи на бібліотечній архітектурі натомість конфігуруються безпосередньо на етапі збірки, часто тісно інтегруючись з елементами середовища розробки та компілятором.

Деякі з таких систем взагалом не мають якоїсь центральної сутності, а цілком складаються з бібліотек, що також слугують драйверами для апаратної платформи, тому такі системи ще часто називають "бібліотечними операційними системами". Приєднуючи певні бібліотеки до коду проєкту, користувач або розробник отримує підтримку відповідного функціоналу, що можна використовувати.

1.2. Спільні принципи проблемно-орієнтованих операційних систем

В бібліотечних операційних системах, межі безпеки зміщуються на найбільш низькі з доступних рівнів, щоби бібліотеки-драйвери мали щонайменший розмір, з чого випливає можливість до їх більш простого аудиту з питань безпеки. Вся відповідальність та можливі проблеми з безпекою покладаються на, власне, проєкт, що розробляється на основі цих бібліотек, який може з їх використанням напряду працювати з апаратними компонентами та ресурсами системи. Для подальшого поліпшення рівня безпеки таких систем, на рівні таких модулів також вводяться набори політик та ізоляції привілей різних компонентів на рівні прикладного програмного забезпечення.

З точки зору швидкодії такі системи мають як переваги, так і недоліки, що є наслідками безпосередньо такої архітектури. Як вже було зазначено вище, в таких системах абсолютно відсутній розподіл адресних просторів, весь код має доступ до єдиного регіону оперативної пам'яті, а самі інструкції виконуються на найвищому рівні привілей процесора. З цього, по-перше, виходить основна перевага в швидкодії таких систем — немає необхідності для переміщення даних від користувацького коду до коду ядра. Таким чином, бібліотечна система не потребує перемикань процесора між режимами ядра та

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						9
Зм.	Арк.	№ докум.	Підп.	Дата		

користувача для доступу до апаратних ресурсів або використання привілейованих інструкцій. З цього ж, однак, випливає й недолік того ж плану — за відсутності такого розділення адресних просторів, побудова систем, де одночасно виконуються декілька програм з ізольованими ресурсами або привілеями, є майже неможливою задачею.

Водночас з цією особливістю, проблемою таких рішень є необхідність вибору конкретних драйверів для збірки операційних систем на їх базі, що дещо ускладнює розробку продуктів, а також значно погіршує здатність програм до виконання на різних апаратних платформах.

За допомогою зменшення розмірів таких операційних систем, окрім зменшення площини для потенціальних атак, також надається можливість для оптимізації системи в цілому шляхом тісної інтеграції коду розробника із драйверами системи. В цілому, розмір типових unikernel зазвичай складає не більше чотирьох відсотків від розмірів ядер звичних операційних систем.

Також однією проблемою, що впливає з такої сильної відмінності бібліотечних операційних систем від більш традиційних рішень, є велика складність перенесення існуючих проєктів для виконання на основі таких архітектур. Через сильну унікальність та відмінність компонентів таких систем, кожна з них мусить власноруч надати необхідні інструменти для підтримки мов програмування та їх інтерпретатори, якщо потрібно використовувати інтерпретовані мови або мови, що компілюються до байт-коду. Також такі системи мають надавати реалізації стандартних бібліотек відповідних мов, що також є нелегкою задачею. Виходячи з цього, більшість таких систем зазвичай розроблюються з метою підтримки лише однієї або декількох конкретних мов програмування.

Велика кількість проблемно-орієнтованих операційних систем становлять собою в першу чергу набір інструментів для компілятора, аби змусити його з мінімальними зусиллями збирати ефективні та оптимізовані образи таких операційних систем, одразу готові до запуску на гіпервізорі або реальній

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						10
Зм.	Арк.	№ докум.	Підп.	Дата		

апаратній платформі. При цьому багато рішень у цій сфері спираються на основні функції мов програмування, для яких вони розроблюються. Так, наприклад, важливими перевагами будуть модульність, метапрограмування та автоматичне керування пам'яттю.

В цілому, така архітектура є багато спільного в своєму принципі із мікроядерними архітектурами, бо також бере принцип розділення "відповідальностей" на окремі модулі, хоча й відрізняється тим, що в мікроядерній архітектурі присутнє ядро, що займається обміном повідомленнями між сервісами, а також розподілом ресурсів між ними.

Unikernel загалом є дуже хорошим засобом для розробки систем, що використовують сервісно-орієнтовані або мікросервісні принципи архітектури в своїй основі.

1.3. Використання гіпервізорів в поєднанні з проблемно-орієнтованими ядрами

Задля розширення можливостей операційних систем з проблемно-орієнтованою архітектурою, впливаючи з основного принципу їх побудови — єдиного адресного простору, можна використовувати апаратні можливості віртуалізації, що надаються процесорами. Таким чином, над екземплярами таких операційних систем ставиться окрема сутність з більшим рівнем привілей процесора — гіпервізор. Це дозволяє розділяти на більш низькому рівні ресурси апаратної платформи між гостьовими системами, при цьому не потребуючи особливих змін від самих систем. Це надає можливість одночасно виконувати декілька таких проблемно-орієнтованих рішень, а процесорний час та спільна пам'ять при цьому буде розподілена між ними гіпервізором.

Для повної підтримки гіпервізорів такими рішеннями додатково надається набір драйверів для взаємодії з віртуалізованими ресурсами, наприклад, для використання віртуального диску, що є частиною спільного дискового простору розподіленого між системами гіпервізором.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						11
Зм.	Арк.	№ докум.	Підп.	Дата		

Водночас з можливостями для одночасного виконання надається також більш детальний контроль за адресним простором, бо тепер гіпервізор може визначити права доступу для окремих ділянок оперативної пам'яті, що надаються системам-гостям.

Сучасні гіпервізори надають абстракції над ресурсами, що можуть динамічно масштабуватися як вертикально шляхом додавання більшої кількості фізичної пам'яті або більшої кількості процесорів, так і горизонтально шляхом запуску більшої кількості екземплярів віртуальних машин. Багато інших операційних систем та застосунків не мають можливості скористатися цими перевагами через те, що вони були спроектовані ще до того, як було створено сучасні гіпервізори. Дуже часто, до традиційних застосунків, що виконуються у віртуальних машинах, додаються зовнішні балансувальники навантаження прикладного рівня, або надати сервісу змогу еластичної реакції на запити шляхом запуску нових віртуальних машин по мірі зростання навантаження. Традиційні системи, однак, не є добре оптимізованими для швидкого запуску або їх розміру (так, Microsoft Windows може накладати патчі на систему прямо під час запуску), тому балансувальнику випадає нелегка задача, коли потрібно це враховувати, результатом чого становляться стрибки у завантаженості, що призводить до втрачених ресурсів [2].

Такий підхід є набагато гнучкішим, але все ж потребує наявності такої центральної сутності, що в деякій мірі відхиляється від основних принципів Unikernel. Через високу ефективність такого рішення, воно широко використовується та підтримується різними проблемно-орієнтованими середовищами.

1.4. Приклади рішень на проблемно-орієнтованій архітектурі

Далі наведено ряд існуючих рішень на архітектурах, що використовують бібліотеки та є проблемно-орієнтованими, кожне з яких відрізняється набором функцій, що надається розробнику, підтримуваними мовами програмування, а також ступенню розвиненості:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						12
Зм.	Арк.	№ докум.	Підп.	Дата		

- ClickOS — система, призначена для мережевих пристроїв з високою пропускну здібністю і є основою на відкритих технологіях віртуалізації. Образи програм на базі такої системи є доволі малими (5 Мб), завантажуються і починають виконання дуже швидко (від 20 мс) і можуть використовуватись на недорогому апаратному забезпеченні;
- Graphene — бібліотечна система, що ставить за мету сумісність з Linux і орієнтацію на безпеку в системах з багатьма процесами, які можуть представляти собою серверні програми або традиційні програми з інтерфейсом командного рядку. Graphene розподіляє багатопроцесні додатки на ряд пікопроцесів, надаючи їм при цьому низку абстракцій для міжпроцесної комунікації, таких як сигнали, черги повідомлень та семафори. З міркувань безпеки такі системи також можуть поміщати потенційно небезпечні або захищені процеси до "пісочниці" для їх ізоляції;
- IncludeOS — мінімалістична операційна система, що складається лише з заголовочних файлів коду та дозволяє побудову простих систем на мові програмування C++;
- MirageOS — набір бібліотек, що розроблюється на базі мови програмування OCaml, та являє собою середовище для побудови захищених систем для роботи з мережами на хмарних та мобільних платформах (див. рис. 1.2);
- Unikraft — проєкт, метою якого на високому рівні є надання можливості для дуже спрощеного в порівнянні з іншими аналогами процесу побудови систем для конкретних додатків, при цьому не вимагаючи високого рівня експертизи, якого вони потребують.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						13
Зм.	Арк.	№ докум.	Підп.	Дата		

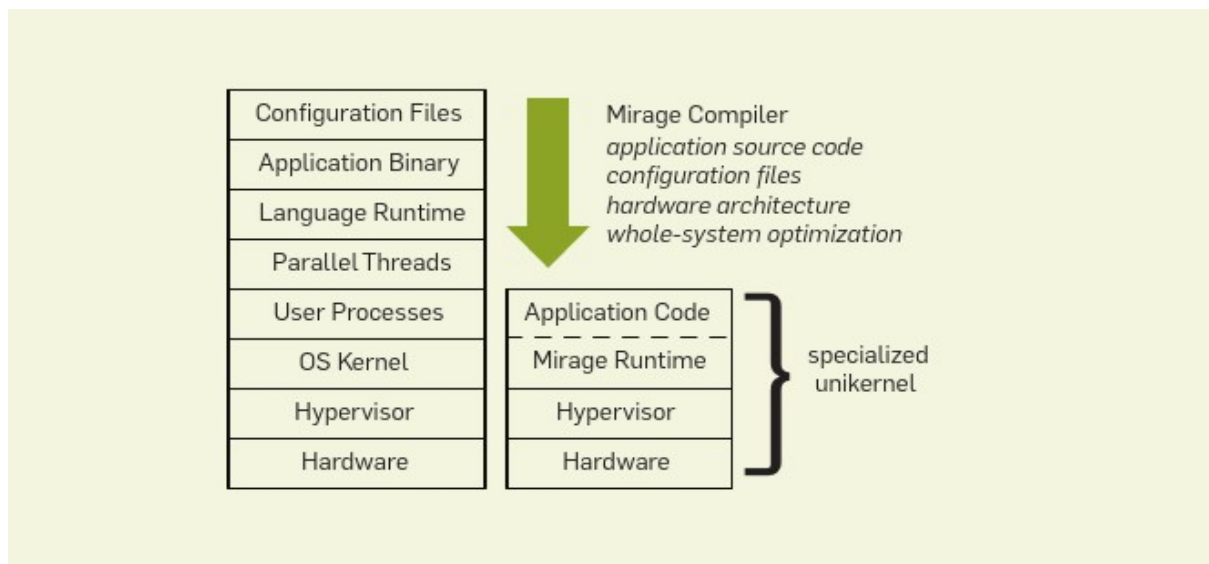


Рис. 1.2 Високорівнева архітектура системи MirageOS

Цей список не є вичерпним, бо існує безліч аналогічних проєктів, зазвичай орієнтованих на певні мови або технології, але переслідують схожі цілі.

1.5. Ядро Linux

Альтернативною платформою для розробки програмних рішень є використання ядра Linux. Цей проєкт є дуже поширеним в абсолютно всіх сферах та галузях програмної інженерії, через що став одним із де-факто стандартів для розробки програмного забезпечення. На відміну від попереднього рішення, це ядро є значно громіздкішим проєктом, а також має значні архітектурні відмінності.

Linux являє собою монолітне ядро з підтримкою модулів, що можна завантажувати під час виконання операційної системи на його базі. За роки своєї розробки Linux отримав підтримку на дуже широкому спектрі апаратних платформ та має безліч драйверів для взаємодії з апаратним забезпеченням таких систем.

Linux в своєму принципі наближається до вже зазначеної філософії UNIX, тому для нього є стійка специфікація, хоча й сам Linux вносить набір властивих лише йому функцій.

Виходячи з модульності та відкритості самого ядра, розробка драйверів, що мають працювати у привелейованому режимі, може бути виконана у форматі модулів ядра, хоча їх створення має дещо вищі вимоги до експертизи в порівнянні із написанням систем на базі unikernel.

Для написання програм для виконання у користувацькому режимі процесора можна використати два підходи.

1.5.1. Використання існуючого оточення

Для спрощення розробки додатків та значного її пришвидшення, для ОС на базі Linux можна скористатись вже існуючими програмними середовищами, на основі яких вже буде виконуватись додаток. Такі середовища містять програми, що будуть займатись ініціалізацією системи, створенням вузлів для пристроїв у файловому дереві, запуском допоміжних сервісів, якщо такі потрібні, та іншим.

Рішення з такими оточеннями зазвичай використовуються в системах, що віртуалізуються за допомогою технологій контейнеризації, тобто коли для виконання системи-хоста та гостьової системи використовується спільне ядро.

Спеціально для використання в таких цілях було розроблено деякі дистрибутиви операційних систем на базі Linux, що ставлять за мету досягнення найменшого розміру. Одним із таких спеціальних дистрибутивів є Alpine Linux, що використовує musl в якості стандартної бібліотеки мови C та систему ініціалізації OpenRC. Задля підвищення рівня безпеки, бібліотеки та програми, що надаються з цим проектом, збираються як Position Independent Code із захистом від переповнень стеку. Приклад скрипту для OpenRC наведено нижче:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						15
Зм.	Арк.	№ докум.	Підп.	Дата		

```
#!/sbin/openrc-run

name=$RC_SVCNAME

cfgfile="/etc/$RC_SVCNAME/$RC_SVCNAME.conf"

command="/usr/bin/my_daemon"

command_args="--my-daemon-args"

command_user="my_system_user"

pidfile="/run/$RC_SVCNAME/$RC_SVCNAME.pid"

start_stop_daemon_args="--args-for-start-stop-daemon"

command_background="yes"

depend() {
    need net
}

start_pre() {
    checkpath --directory --owner $command_user:$command_user --mode 0775 \
        /run/$RC_SVCNAME /var/log/$RC_SVCNAME
}
```

Рис. 1.3 Приклад скрипту ініціалізації OpenRC

Існують також більші і менші за розміром системи, але вони вже не можуть надати такого балансу функціоналу і розміру, як це робить Alpine, саме тому він є найбільш поширеним вибором при побудові систем для контейнерної віртуалізації.

Для таких дистрибутивів створення додатків полягає в написанні коду, що використовує стандартну бібліотеку C, що надається системою, а також створення відповідних конфігураційних файлів для її автоматичного запуску, якщо вона виконана у вигляді сервісу, наприклад, скриптів запуску для OpenRC або systemd.

Так як такий вид систем є найбільш узагальненим, майже неможливо провести їх детальну оптимізацію, тому вони зазвичай завантажуються довше

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						16
Зм.	Арк.	№ докум.	Підп.	Дата		

та використовують більший обсяг апаратних ресурсів. Виходячи з цього, далі буде наведено більш складний у реалізації, але більш оптимізований підхід до розробки цільових додатків.

1.5.2. Розробка власного оточення

Маючи ядро Linux, можна відмовитись від використання існуючих дистрибутивів взагалі, а натомість розробити власну програму, що буде виконуватись в якості процесу ініціалізації. Як відомо, після завантаження ядра Linux, воно передає керування виконанням певній програмі, що має назву `init`. В наведеному вище прикладі цією програмою є `OpenRC`, що проводить загальну ініціалізацію системи та інші операції при старті. Розробник, натомість, має можливість заміни цього `init` на власний, при цьому значно пришвидшуючи запуск такого рішення.

В такому випадку систему можна більш детально оптимізувати під потреби розробника або користувача, а використання ресурсів системи буде цілком залежати лише від коду самого додатку. Також, через зменшення загального розміру всієї системи, відповідно зменшиться й потенціальна площа для хакерських атак, що є значною перевагою для розробки систем, де безпека є критичною вимогою.

Така програма отримує доступ до системних викликів до ядра Linux, а також може скористатись вже існуючою стандартною бібліотекою необхідної мови програмування, щоби ці виклики абстрагувати. Також програми мають безпосередню можливість до створення необмеженої кількості процесів без використання будь-яких гіпервізорів, що є дуже важливим, якщо в системі потрібно використати декілька сервісів. Також в такій архітектурі програми мають чіткий розподіл адресних просторів, що також є великою перевагою з точки зору безпеки.

В цілому, такий підхід є дуже схожим на використання бібліотечних проблемно-орієнтованих операційних систем з гіпервізорами, але в цьому

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						17
Зм.	Арк.	№ докум.	Підп.	Дата		

випадку ключовий елемент перенесено на більш низький рівень привілей, а також має набагато більшу простоту використання.

1.6. Розробка модулів ядра для Linux

Ядро Linux надає власний інтерфейс для розробки модулів та драйверів, при цьому накладаючи на їх роботу особливі вимоги.

Драйвери пристроїв або файлових систем займають особливе місце у розробці операційної системи. Вони є свого роду “чорними ящиками”, що змушують елементи апаратного забезпечення певним чином реагувати на добре визначений програмний інтерфейс; вони абсолютно приховують деталі того, як працюють ці пристрої. Дії користувачів проводяться шляхом стандартизованих викликів, що не залежать від жодного конкретного драйверу, відображення таких викликів на специфічні для пристроїв операції, що виконуються на реальному апаратному забезпеченні — задача драйверів. Цей інтерфейс програмування є таким, щоби драйвери можна було збирати абсолютно окремо від основного ядра, а потім під’єднувати прямо під час виконання за мірою їх необхідності. Ця модульність робить написання драйверів для Linux значно легшою справою, завдяки чому на даний момент Linux підтримує більше тисячі різних пристроїв та апаратних платформ.

Основним принципом при розробці драйверів є надання “механізму, а не політики”. Це означає визначення максимально можливої гнучкості для такого драйвера, навіть шляхом потенціального подовження часу розробки. Ця ідея розрізнення механізму та політики також є однією із найкращих ідей, що створив Unix, бо більшість проблем програмування у цій сфері можна поділити на два питання: “які можливості потрібно надати” (механізм) та “як ці можливості дозволяється використовувати” (політика). Якщо програма або навіть різні програми спробують дати одночасну відповідь на ці два питання, програмні пакети стануть набагато простішими у розробці, їх стане легше адаптувати до власних потреб [3].

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						18
Зм.	Арк.	№ докум.	Підп.	Дата		

В системах Unix, деяка кількість одночасних процесів займається виконанням своїх різних задач. Кожен процес запитує у системи ресурси — обраховувальні можливості, пам'ять, робота з мережею тощо. Хоча ядро й не можна чітко поділити на окремі задачі, ролі, що виконуються у його коді, можна розподілити по наступним п'яти категоріям:

- Керування процесами
- Керування пам'яттю
- Файлові системи
- Керування пристроями
- Робота із мережею

З вище перерахованих пунктів, до окремих драйверів найдоцільніше було б виносити останні три, тим самим залишивши ядру лише базовий набір ролей. Це дуже сильно підвищує модульність та адаптивність проєкту, а сама система становиться більш оптимізованою та ресурсоефективною.

Найпростіший модуль ядра Linux виглядає наступним чином:

```
#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Mark Poliakov");

MODULE_DESCRIPTION("Hello world Linux module");

MODULE_VERSION("1.0.0");

static int __init mod_init(void){

    printk(KERN_INFO "Hello, World\n");

    return 0;

}

static void __exit mod_exit(void){

    printk(KERN_INFO "The module stopped\n");

}
```

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						19
Зм.	Арк.	№ докум.	Підп.	Дата		

```

}

module_init(mod_init);

module_exit(mod_exit);

```

Рис. 1.4 Приклад коду модуля ядра Linux

Такі модулі мають змогу завантажуватись прямо під час роботи ядра, що є можливим завдяки тому, що вони збираються, як незалежні від адреси завантаження, бінарні образи, а ядро Linux під час їх завантаження виконує необхідні операції для вирішення адрес зовнішніх посилань та виділяє для їх розміщення місце у пам'яті.

1.7. Ключові відмінності монолітних ядер

Монолітні ядра надають більш гнучке та просте, але водночас більш громіздке середовище для розробки користувацьких додатків. Багато функціоналу, що може бути реалізовано на монолітних або навіть мікроядрах, може бути виконано й на бібліотечних проблемно-орієнтованих системах, але це є доволі проблематичною та складною задачею.

Так, рівень привілей виконання таких ядер є нижчим, ніж той у гіпервізорів проблемно-орієнтованих систем, а можливість використання роздільних адресних просторів та багатозадачності з боку ядра значно спрощує розробку систем з багатьма сервісами.

Хоча ці системи й не мають такого потенціалу до оптимізації, вона все ще є можливою в деякій мірі шляхом використання власноруч розроблених оточень або хоча б вже існуючих легковажних варіантів.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						20
Зм.	Арк.	№ докум.	Підп.	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 1

Обидва методи, що наведено вище, мають власні переваги та недоліки, що впливають з дуже великої різниці на архітектурному рівні.

Проблемно-орієнтовні системи дозволяють зменшити розміри систем, потенціальні площі хакерських атак, а також оптимізувати використання ресурсів системи в дещо більшому масштабі, ніж це можливо з використанням монолітних ядер. В той самий час, такий підхід вимагає набагато більшої експертизи та має значно вищі вимоги для розробки.

Як вже було зазначено, проблемно-орієнтовані системи мають спільний адресний простір, але можуть бути використані разом із гіпервізором для надання детального розподілу регіонів пам'яті. В той же час, ядро Linux також має таку можливість, при цьому й саме ядро може займатись розподілом апаратних ресурсів.

На відміну від бібліотечних операційних систем, де кожен компонент для підтримки апаратного забезпечення потрібно приєднувати окремо, монолітні ядра централізовано надають безпосередньо або через модулі абстрагований набір функцій, що також значно спрощує розробку на їх основі. В той же час, монолітні та модульні ядра є значно громіздкішими в порівнянні з проблемно-орієнтованими системами, тому це є їх недоліком.

В цьому проєкті ці параметри враховуються, аби надати продукт, що заповнить технологічну нішу між цими рішеннями.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						21
Зм.	Арк.	№ докум.	Підп.	Дата		

РОЗДІЛ 2. АРХІТЕКТУРА СЕРЕДОВИЩА

2.1. Опис необхідних для побудови компонентів системи

В результаті оцінювання можливих альтернатив при проектуванні системи було обрано архітектуру з монолітним ядром. Цей вибір зумовлений відносною легкістю в різних аспектах імплементації та подальшої підтримки. Для створення системи на основі такої архітектури було розроблено наступні компоненти:

1. Ядро системи — основний компонент системи, що буде слугувати базою для розробки додатків в даному середовищі. Саме ядро поділяється на відповідні підсистеми:
 - a) Код для взаємодії із компонентами конкретної апаратної платформи (x86);
 - b) Підсистема керування та розподілу пам'яті;
 - c) Драйвери пристроїв, що не залежать від машинної архітектури, наприклад, драйвери PCI- або USB-пристроїв;
 - d) Драйвери файлових систем;
 - e) Драйвери псевдо- та віртуальних системних пристроїв;
 - f) Підсистема віртуальної файлової системи, що абстрагує попередньо зазначені підсистеми у вигляді файлів, директорій та файлів-пристроїв;
 - g) Підсистема системних викликів для взаємодії з боку користувацьких програм;
 - h) Підсистема роботи з процесами та потоками обчислення, сигналів;
 - i) Підсистема завантаження бінарних файлів;

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						22
Зм.	Арк.	№ докум.	Підп.	Дата		

2. Імплементация стандартної бібліотеки мови С — необхідна для значного полегшення розробки нових додатків і надає можливість для використання вже існуючих.
3. Набір користувацьких утиліт — потрібні як для відлагодження роботи результуючої системи, так і для загальних потреб користувача, як описано в POSIX.
4. Завантажувач для UEFI — надає можливість для завантаження ядра та інших частин системи з середовища UEFI та містить реалізацію узгодженого з ядром протоколу завантаження, за яким передається важлива інформація про систему.

2.2. Високорівневий опис взаємодії компонентів

Компоненти системи, що було наведено вище, поділяються на “кільця” виконання згідно з тим, на якому рівні привілей процесора вони виконуються. Так, не враховуючи рівень гіпервізора, архітектура x86 надає чотири таких рівні, що носять назви CPL0, CPL1, CPL2 та CPL3, де CPL — Current Privilege Level, тобто поточний рівень привілей процесора. Розроблювана в цій роботі система, а також більшість інших систем, використовує лише два з цих рівнів — CPL0 та CPL3.

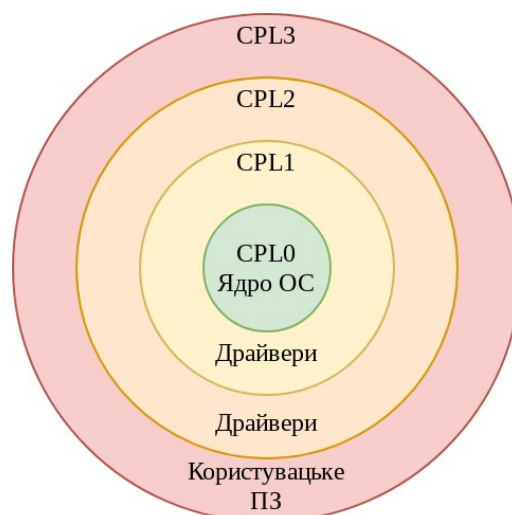


Рис. 2.1 Рівні привілей процесора у x86

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						23
Зм.	Арк.	№ докум.	Підп.	Дата		

На рівні CPL0 у цьому проєкті будуть виконуватись код завантажувача та ядро системи, а на CPL3 — користувацькі застосунки та код стандартної бібліотеки мови програмування C. Далі ці рівні будуть носити назву “рівень ядра” та “користувацький рівень”.

В такій архітектурі прямий доступ до апаратного забезпечення та багатьох спеціальних функцій процесора має лише рівень ядра, а користувацькі програми такої змоги не мають.

Для того, щоб дати користувацьким програмам можливості для взаємодії з апаратною частиною системи, з боку ядра надається інтерфейс системних викликів за допомогою інструкції SYSCALL [4]. Ця інструкція дозволяє підвищити рівень привілей процесора, а керування при цьому передається спеціальному обробнику системних викликів. Користувацька програма при цьому має змогу передати ядру параметри для таких викликів через регістри процесора та власний адресний простір пам’яті.

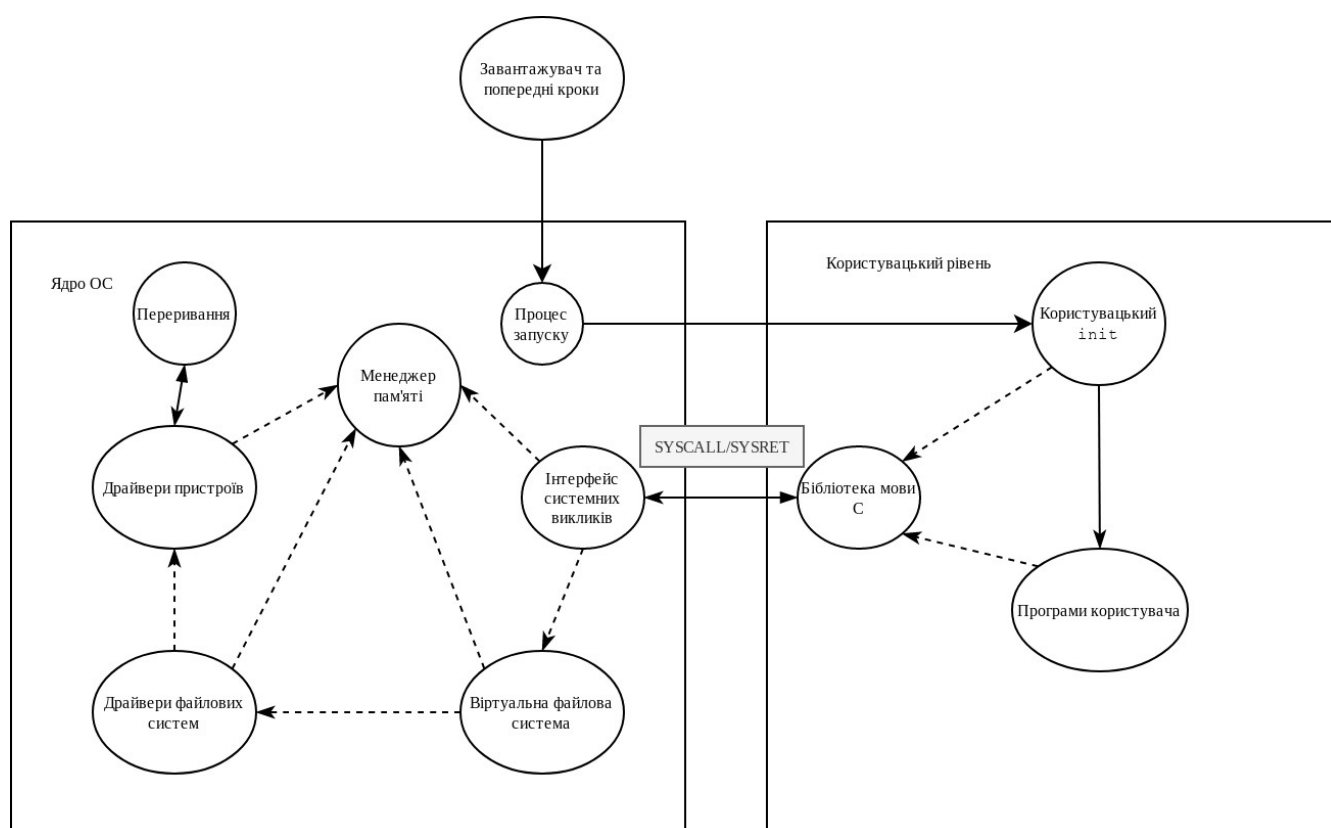


Рис. 2.2 Високорівнева схема взаємодії компонентів системи

Також було зазначено, що для користувацьких програм буде розроблено стандартну бібліотеку мови програмування C. Для спрощення переносу існуючих програм та надання найбільш знайомого інтерфейсу для прикладної розробки цю бібліотеку буде створено, керуючись стандартом POSIX.

В такому підході буде надано два основні типи функцій — функції, що є обгортками над системними викликами, такі як `openat()`:

```
int openat(int dfd, const char *pathname, int flags,
mode_t mode) {
    return SET_ERRNO(int, __syscall4(SYS_CALL_NR_OPENAT,
dfd, (long) pathname, flags, mode));
}
```

а також функції, що вносять додатковий функціонал, наприклад функції для роботи з датами та часом або функції, що використовують функції першого типу для більш високорівневих задач бібліотеки.

Додатково цією бібліотекою буде надано засоби для опрацювання сигналів від ядра ОС або від інших програм — функції для встановлення та видалення обробників сигналів, набір стандартних обробників тощо.

Окрім цього, бібліотека мови C надасть програмам стандартну обгортку для точки входу в програму, що буде при її старті ініціалізувати різні модулі цієї бібліотеки, наприклад, засоби для виділення пам'яті та обробники сигналів.

Повну таблицю системних викликів, що надаються ядром, буде наведено у розділі 2.4.

2.3. Внутрішня архітектура ядра системи

Базовою частиною, що буде виконувати розподіл ресурсів системи, контроль доступу та забезпечувати взаємодію програмних компонентів користувача із апаратною платформою, є ядро.

Ядро системи включає в себе велику кількість підсистем для забезпечення спрощених для користувацьких програм можливостей взаємодії з апаратним середовищем. Існують різні архітектури ядер:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						25
Зм.	Арк.	№ докум.	Підп.	Дата		

- Монолітні ядра — всі підсистеми ядра є частиною одного цілого
- Мікроядра — центральне ядро складається лише з абсолютного мінімуму: засоби для розподілу пам'яті, контролю процесів та передачі повідомлень між ними. Всі інші підсистеми мають вигляд "сервісів", що працюють як звичайні процеси та комунікують між собою через зазначене ядро.
- Гібридні ядра — суміш монолітної та мікроядерної архітектур. Розробник може сам обирати, які компоненти залишити всередині ядра, а які винести "назовні" в якості сервісів.
- Екзоядра, також зазначені раніше, як проблемно-орієнтовані системи або бібліотечні системи — всі основні компоненти для взаємодії з апаратною платформою винесено до бібліотек, що приєднуються до коду користувача. В такому випадку сам код користувача можна вважати ядром.

У даному проєкті буде розроблено саме монолітне ядро з причин, зазначених у розділі 2, а також з урахуванням потенціальних обмежень та недоліків альтернатив. Цей тип ядер є найбільш розповсюдженим в сучасних операційних системах. Ядрами такого типу є Microsoft NT (в системах Windows 95, 98 та ME), Linux, а також ядра різних варіантів систем BSD.

В якості формату образу ядра було обрано стандартний для Unix-подібних систем формат ELF (Executable and Linkable Format).

Разом із ядром буде розроблено і набір протоколів для його завантаження в різних середовищах, що наявні на цільовій апаратній платформі x86-64, а саме:

- Для завантаження в режимі сумісності (Legacy/BIOS mode) буде використовуватись існуючий протокол Multiboot 2, а в якості завантажувача пропонується використовувати GNU GRUB 2, що є еталонною імплементацією завантажувача за цим протоколом.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						26
Зм.	Арк.	№ докум.	Підп.	Дата		

- Для завантаження в режимі UEFI буде розроблено власну EFI-програму, що буде містити засоби для завантаження ядра у форматі ELF і його RAM-диску з користувацьким ПЗ. Розробка власного "прошарку" сумісності є необхідною, бо за специфікацією Microsoft UEFI не має внутрішніх засобів для завантаження бінарних файлів у форматі ELF, а натомість працює з PE (Portable Executable) від Microsoft.

Спираючись на вибір монолітної архітектури ядра, воно являтиме собою єдину кодову базу, але, як і у випадку з багатьма існуючими варіантами монолітних ядер, також включатиме до свого складу засоби для завантаження окремих модулів безпосередньо під час своєї роботи.

Ядро має реалізувати зазначені раніше компоненти і надавати їх користувацьким програмам через інтерфейс системних викликів, наведений у наступному розділі.

2.4. Опис набору системних викликів

Для забезпечення комунікації між апаратною платформою та програмами через ядро, ним буде надано інтерфейс системних викликів, що буде являти собою спеціальний, властивий конкретно цьому ядру інтерфейс для розробки застосунків.

Параметри системного виклику передаються наступним чином:

Таблиця 2.1

Формат розміщення аргументів системного виклику

Регістр	Призначення
RAX	Номер системного виклику у таблиці
RDI	Перший аргумент системного виклику
RSI	Другий аргумент системного виклику
RDX	Третій аргумент системного виклику
R10	Четвертий аргумент системного виклику
R8	П'ятий аргумент системного виклику
R9	Шостий аргумент системного виклику

Далі буде наведено таблицю системних викликів, що надаються ядром, що розроблюється.

Таблиця 2.2

Опис системних викликів, що має надавати ядро

RAX	Функція	Опис
Базові функції для роботи з файлами та пам'яттю		
0	read(fd, buf, len)	Зчитати len байтів з дескриптору fd у buf
1	write(fd, buf, len)	Записати len байтів з буферу buf до fd
2	openat(dfd, fp, flags, mode)	Відкрити файл із шляхом fp відносно dfd
3	close(fd)	Закрити fd
4	fstatat(dfd, fp, st, flags)	Записати до буферу st статус файлу із шляхом fp відносно dfd
8	lseek(fd, off, whence)	Перемістити позицію читання або запису до файлу fd
9	mmap(hint, len, prot, flags, fd, off)	Виділити або зарезервувати сторінки віртуальної пам'яті для поточного процесу
11	munmap(addr, len)	Видалити відображення віртуальної пам'яті з адресного простору поточного процесу
12	sbrk(brk)	Надається для старих програм, щоб збільшувати розмір сегменту даних вгору. Замінено на mmap(2)
16	ioctl(fd, cmd, args)	Робить спеціалізовані запити до драйверів пристроїв або файлових систем
21	faccessat(dfd, fp, mode, flags)	Перевіряє доступ до об'єкту за шляхом fp відносно dfd
22	pipe(filedes)	Створює двохсторонній буфер-конвейер для передачі даних між процесами
23	select(n, in, out, exc, tv)	Блокує процес до тих пір, поки не буде вичерпано час, або поки якийсь з дескрипторів не стане доступним для читання або запису
32	dup(from)	Дублікує вказаний файловий дескриптор
33	dup2(from, to)	Дублікує вказаний файловий дескриптор, при цьому записує його до to
76	truncate(fp, len)	Змінює розмір вказаного файлу за шляхом fp
77	ftruncate(fd, len)	Змінює розмір файлового дескриптору fd
79	getcwd(buf, lim)	Виводить до buf поточну директорію процесу
80	chdir(fp)	Змінює поточну директорію процесу
83	mkdirat(dfd, fp, mode)	Створює нову директорію за шляхом fp відносно dfd

Таблиця 2.2 (Продовження)

85	creat(fp, mode)	Створює та відкриває новий файл. Не використовується за наявності O_CREAT у openat(2)
87	unlinkat(dfd, fp, flags)	Видаляє об'єкт файлової системи за шляхом fp відносно dfd
89	readdir(fd, ent)	Зчитує запис з директорії fd у буфер ent
90	chmod(fp, mode)	Змінює права доступу до файлу
91	readlinkat(dfd, fp, buf, lim)	Зчитує зміст символічного посилання
92	chown(fp, uid, gid)	Змінює власника та групу власності файлу
133	mknod(fp, mode, dev)	Створює вузол у файловій системі для відображення пристрою
Функції для роботи зі спільною пам'яттю		
113	shmget(size, flags)	Виділяє регіон спільної пам'яті розміром size
114	shmat(id, hint, flags)	Розмічає спільний регіон id у адресний простір поточного процесу
115	shmdt(id)	Видаляє відображення спільного регіону id з адресного простору поточного процесу
Функції для загального керування системою та отримання інформації про неї		
35	nanosleep(req, rem)	Поміщає поточний процес у очікування на деякий період часу req. Якщо "сон" буде перервано, запише залишившийся час до rem
63	uname(name)	Отримує узагальнену інформацію про систему
96	gettimeofday(tv, tz)	Записує до буферу tv інформацію про поточний час, а до tz про сконфігурований часовий пояс
162	sync()	Примушує систему передчасно записати всі закешовані зміни за скинути стан кешів
165	mount(dev, fp, type, flags, data)	Під'єднує до директорії за шляхом fp файлову систему типу type з блочного пристрою dev
166	umount(fp)	Від'єднує від директорії за шляхом fp файлову систему
169	reboot(m1, m2, cmd, arg)	Надає керування живленню системи: перезапуск, зупинка або вимикання системи
175	module_load(fp, params)	Завантажує модуль ядра з файлу за шляхом fp
176	module_unload(name)	Видаляє з адресного простору ядра модуль з назвою name

Таблиця 2.2 (Продовження)

Функції для керування процесами, сигналами та користувачами		
13	sigentry(entry)	Встановлює адрес entry як точку входу у користувацький обробник сигналу
15	sigreturn()	Викликається із обробнику сигналу, аби повернутись у нормальний режим роботи
24	yield()	Віддає керування процесора іншому процесу
39	getpid()	Повертає номер поточного процесу
56	clone(entry, st, flags, arg)	Клонує потік, створюючи новий, при цьому встановлюючи його стек та точку входу
57	fork()	Клонує поточний процес, створюючи новий, повертаючи його номер
59	execve(fp, argp, envp)	Завантажує до адресного простору поточного процесу новий образ програми з файлової системи за шляхом fp
60	exit(status)	Завершує роботу поточного процесу або потоку із статусом status
62	kill(pid, signum)	Надсилає сигнал signum процесу pid. Якщо pid = 0, то сигнал надсилається самому собі
102	getuid()	Повертає номер користувача поточного процесу
104	getgid()	Повертає номер групи поточного процесу
105	setuid(uid)	Змінює номер користувача поточного процесу
106	setgid(gid)	Змінює номер групи поточного процесу
109	setpgid(pid, pgrp)	Змінює номер групи процесів для процесу pid. Якщо pid = 0, змінює для поточного
110	getppid()	Повертає номер батьківського процесу
112	setsid()	Починає нову сесію для поточного процесу
121	getpgid(pid)	Повертає номер групи процесів процесу pid
131	sigaltstack(ss, old_ss)	Встановлює новий адрес стеку, що використовується при обробці сигналів
247	waitpid(pid, status, flags)	Чекає на завершення процесу pid, після чого поміщає до буфера status його результат
Мережеві функції		
41	socket(dom, type, prot)	Створює новий об'єкт мережевого сокету
42	connect(fd, sa, len)	Під'єднує об'єкт сокету до адреси sa
45	accept(fd, sa, salen)	Очікує на під'єднання віддаленої частини до сокету, після чого записує її адресу до sa
44	sendto(fd, buf, len, sa, salen)	Надсилає дані з буферу buf до віддаленої частини сокету за адресою sa

Таблиця 2.2 (Завершення)

45	recvfrom(fd, buf, len, sa, salen)	Отримує в буфер buf дані від віддаленої частини сокету, записує її адресу до буферу sa
49	bind(fd, sa, len)	Прив'язує мережевий сокет до локальної адреси або порту
54	setsockopt(fd, lev, opt, val, len)	Встановлює розширені налаштування для мережевих сокетів
248	netctl(name, op, arg)	Викликає на мережевому інтерфейсі з назвою name функцію налаштування op
Функції для відлагодження		
249	trace(msg, lim)	Виводить до логу ядра повідомлення msg
250	instr(cmd, a0, a1, a2)	Надає набір утиліт для інструментації та відлагодження як ядра, так і адресних просторів та станів процесів через системні виклики

2.5. Віртуальна файлова система

Драйвер віртуальної файлової системи надає іншим модулям та користувацьким програмам зручний для використання інтерфейс, що абстрагує інші драйвери, керуючись принципом, що відомий як “Все є файл” та є одним з ключових елементів філософії UNIX.

Цей драйвер надає “середній прошарок” між конкретними драйверами пристроїв, файлових систем й інших підсистем ядра та високорівневим файловим деревом.

Драйвер віртуальної файлової системи визначає декілька основних структур даних, що забезпечують основне його функціонування: struct vnode та struct vnode_operations [5]. Перша структура містить узагальнену інформацію про об'єкт у дереві файлової системи:

- Назву об'єкту;
- Тип об'єкту:
 - Звичайний файл — об'єкт, що являє собою суцільний блок з даних з можливістю читання та запису;
 - Директорія — каталог з дочірніх елементів дерева;

- Мережевий сокет — точки підключення для локального мережевого протоколу Unix;
 - Файл-FIFO — надає спільний буфер для комунікації між процесами;
 - Блоковий пристрій — віртуальний пристрій, що абстрагує якийсь реальний пристрій, що є суцільним регіоном даних: фреймбуфер дисплею, генератор випадкових чисел, дані дискового пристрою тощо;
 - Символьний пристрій — віртуальний пристрій, що абстрагує реальний, що надає можливість для читання та запису лише по одному елементу за раз. Такими пристроями є псевдотермінали (tty), вивід повідомлень з драйверу миші PS/2 тощо.
- Інформація про батьківський об'єкт та список дочірніх елементів;
 - Права доступу, власник та група власності файла;
 - Посилання на `struct vnode_operations` та приватні дані драйвера конкретної файлової системи.

Друга структура, `struct vnode_operations`, містить посилання на імплементації функцій, що властиві драйверам всіх файлових систем: пошук, створення, відкриття, видалення файлів, операції читання та запису тощо.

Кожному об'єкту файлової системи, що завантажений у пам'ять, ставиться у відповідність його `struct vnode`, що разом формують ієрархічне дерево. Для можливості одночасного використання декількох файлових систем, об'єкти-директорії також можуть бути помічені як “точки монтування”. До таких точок можна під'єднувати додаткові файлові системи, при цьому в загальному дереві такі точки монтування будуть слугувати кореневими директоріями таких файлових систем.

Варто зауважити, що такими структурами представлені лише завантажені у даний момент об'єкти, а самі структури даних не є резидентними: коли код надає запит про якийсь файл, він спочатку шукається в цьому дереві, а якщо його там не буде знайдено, буде проведено спробу його завантаження з

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						32
Зм.	Арк.	№ докум.	Підп.	Дата		

резидентної файлової системи, якщо така там є. Іншими словами, віртуальна файлова система кешує у пам'яті базову інформацію про резидентні файли для пришвидшення доступу до них.

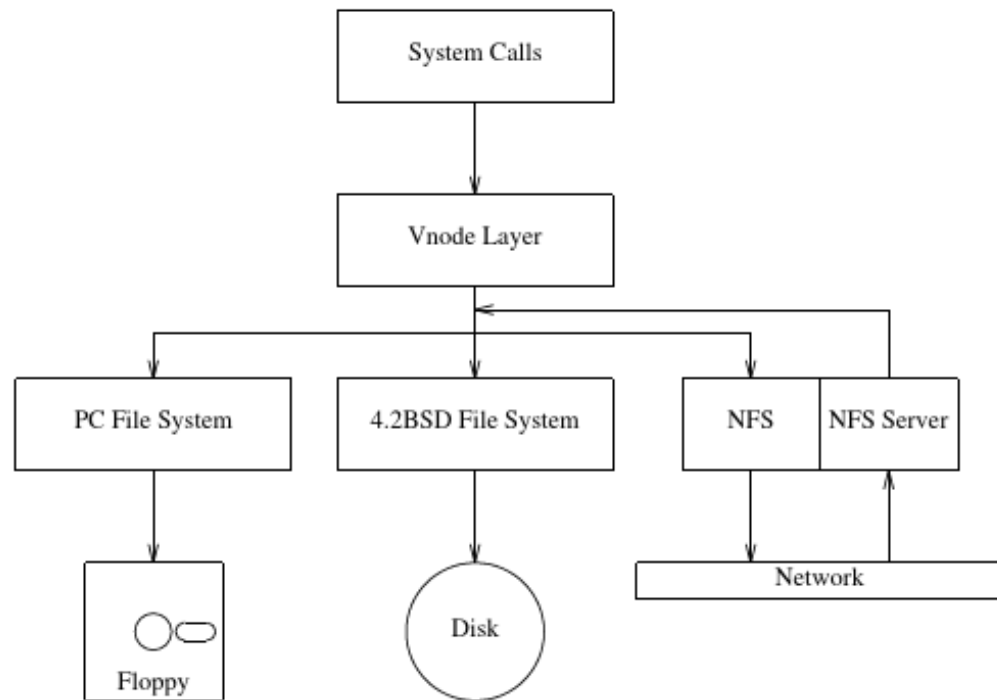


Рис. 2.3 Схематичне зображення віртуальної файлової системи [6]

Сама концепція віртуальної файлової системи була створена як засіб для спрощення реалізації підтримки декількох файлових систем на одному комп'ютері, та вперше була розроблена Sun Microsystems у їх SunOS 2.0 в 1985 [6]. Це дало простий та прозорий інтерфейс для доступу до локальних файлових систем формату UFS одночасно з мережевими директоріями у NFS. Сучасного вигляду ця концепція набула у 4.4BSD, її ж було взято за основу і в цій роботі.

Далі буде наведено перелік функцій, що формують базовий функціонал віртуальної файлової системи:

Таблиця 2.3

Набір операцій віртуальної файлової системи

Назва	Операція
vfs_setcwd	Встановлення поточного каталогу для процесу
vfs_mount	Під'єднання файлової системи до каталогу
vfs_umount	Від'єднання файлової системи від каталогу
vfs_link_resolve	Знаходження об'єкту, на який вказує посилання
vfs_find	Знаходження об'єкту у дереві
vfs_openat	Відкриття файлу за заданим відносним шляхом
vfs_close	Закриття дескриптору файлу
vfs_readdir	Читання змісту каталогу до пам'яті
vfs_mkdirat	Створення директорій
vfs_unlinkat	Видалення об'єктів з файлової системи
vfs_readlinkat	Читання текстових даних символічного посилання
vfs_chmod	Зміна режиму доступу до об'єкту
vfs_chown	Зміна власника або групи власності об'єкту
vfs_ioctl	Спеціальний запит до драйверу об'єкту
vfs_ftruncate	Зміна розміру об'єкту
vfs_faccessat	Перевірка доступу до об'єкту
vfs_fstatat	Отримання опису статусу об'єкту
vfs_write	Запис до об'єкту
vfs_read	Читання з об'єкту
vfs_lseek	Переміщення позиції читання або запису до об'єкту

Додатково з попередніми двома структурами надається також й структура, що описує активний дескриптор відкритого об'єкту у файловій системі struct ofile:

Таблиця 2.4

Структура-опис відкритого файлового дескриптору

Поле	Призначення
flags	Запис режиму відкриття об'єкту та інших особливостей

Таблиця 2.4 (Завершення)

refcount file або socket	Кількість активних користувачів цього об'єкту Якщо структура описує звичайний відкритий файл або каталог, містить інформацію про позицію у ньому; Якщо структура описує відкритий мережевий сокет, містить вкладену структуру, що описує стан сокету
-----------------------------	--

Детальний опис аналогічних структур, що використовуються у 4.4BSD та її похідних також наведено у відповідній літературі [5].

2.5.1. Спеціальні файлові системи ядра

Для спрощення доступу користувацьких програм до багатьох даних про систему, а також для їх можливостей до керування цією системою, ядро надає набір спеціальних віртуальних файлових систем: sysfs та procfs. Інформація з цих файлів отримується шляхом простого зчитування через системний виклик read(2), аналогічно робиться й запис.

Перша з них, sysfs, містить конфігураційні псевдо-файли для зчитування або запису до загального стану системи. Ця файлова система містить наступні файли:

Таблиця 2.5
Опис основних файлів, наданих sysfs

Шлях	R/W	Призначення
/sys/kernel/version	RO	Повертає поточну версію збірки ядра
/sys/kernel/uptime	RO	Повертає час з моменту запуску системи
/sys/kernel/modules	RO	Повертає список завантажених модулів ядра у форматі "ІМ'Я, РОЗМІР, АДРЕСА"
/sys/kernel/debug_serial	RW	Визначає рівень важливості подій логу, починаючи з якого повідомлення будуть виводитись у послідовний порт
/sys/kernel/debug_display	RW	Визначає рівень важливості подій логу, починаючи з якого повідомлення будуть виводитись на основний дисплей
/sys/kernel/smp	RO	Повертає кількість активних процесорів у системі

Таблиця 2.5 (Завершення)

/sys/mem	RO	Повертає детальну інформацію про використання фізичної пам'яті системою, при цьому описуючи різні підсистеми: • Сумарне використання та вільна пам'ять • Використання ядром • Використання програмами • Використання спільної пам'яті • Виділена пам'ять під адресні простори • Кількість сторінок у кеші диску • Купа пам'яті ядра • Дані від алокатора slab
----------	----	--

Окрім sysfs, надається ще файлова система, що містить список та детальну інформацію про поточні процеси, procfs:

Таблиця 2.6
Опис основних файлів procfs

Шлях	R/W	Призначення
/sys/proc/self	RO	Є символічним посиланням на файл з інформацією про поточний процес
/sys/proc/PID/name	RO	Повертає ім'я процесу за номером PID
/sys/proc/PID/parent	RO	Повертає номер батьківського процесу для процесу за номером PID
/sys/proc/PID/ioctx	RO	Повертає інформацію про користувача процесу PID та його файлові дескриптори

Також ядро, що розроблюється, містить файлову систему, що містить список важливих пристроїв, devfs, що будуть описані у наступному розділі.

2.6. Архітектура стандартної бібліотеки мови C

Стандартна бібліотека мови програмування C, що розроблюється для цього проєкту, буде в необхідній мірі реалізовувати частини із третього та другого розділів стандарту POSIX.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						36
Зм.	Арк.	№ докум.	Підп.	Дата		

Сама бібліотека умовно поділяється на наступні підсистеми згідно із заголовочними файлами, у яких вони визначені:

- Базове оточення виконання C — надає точку входу в програму та проводить ініціалізацію інших підсистем;
- ctype — функції для роботи із символами та їх класифікацією;
- dirent — функції для роботи із записами у каталогах;
- env — функції для роботи із змінними оточення;
- err — функції для обробки та виводу помилок;
- errno — функції для збереження помилкових кодів з інших підсистем;
- exit — функції для завершення та очищення стану програми;
- fcntl — функції для низькорівневого керування файлами та дескрипторами;
- malloc — аллокатор пам'яті загального призначення;
- math — загальні математичні та арифметичні функції;
- misc — узагальнені функції;
- pwd — функції для роботи із даними про користувачів системи та їх групами;
- signal — функції для налаштування, контролю та обробки сигналів від інших програм або ядра;
- stdio — засоби для забезпечення вводу та виводу;
- stdlib — стандартний набір засобів для операцій над файлами, каталогами та іншими об'єктами системи;
- string — функції для маніпуляцій над рядками або регіонами даних;
- sys — функції для низькорівневої взаємодії із системою;
- termios — функції для керування апаратним або віртуальним терміналом;

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						37
Зм.	Арк.	№ докум.	Підп.	Дата		

- time — функції для роботи із часом та часовими поясами;
- unistd — набір функцій із стандартного оточення Unix.

Стандартна бібліотека мови C надаватиме ці підсистеми через заголовочні файли .h, а також матиме набір додаткових файлів такого типу для більш гнучкого або низькорівневого доступу.

Основна задача цієї бібліотеки — максимально стандартизувати та спростити розробку або використання існуючих програмних рішень, а також абстрагувати від розробника інтерфейс системних викликів, аби забезпечити максимальну можливість до перенесення існуючого функціоналу з мінімальними змінами до вихідного коду. Завдяки наявності такої абстракції, такі програми можна буде розроблювати, спираючись на вже знайомі стандарти та інтерфейси. Так, наприклад, вихідний код спрощеної утиліти tee буде виглядати наступним чином:

```
#include <stdio.h>

int main(int argc, char **argv) {
    char buf[1024];
    FILE *input = stdin;
    FILE *files[argc];

    files[0] = stdout;
    for (int i = 1; i < argc; ++i) {
        files[i] = fopen(argv[i], "w");
        if (!files[i]) {
            perror(argv[i]);
            return -1;
        }
    }

    while (fgets(buf, sizeof(buf), input)) {
        for (int i = 0; i < argc; ++i) {
            fputs(buf, files[i]);
        }
    }
}
```

Маючи такий інтерфейс для побудови програм, розробка програмного оточення для цільової системи буде значно пришвидшена.

2.7. Архітектура програмного оточення ядра

Окрім ядра, необхідно розробити також і набір користувацьких утиліт, що покладуть основу для використання результуючої операційної системи.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						38
Зм.	Арк.	№ докум.	Підп.	Дата		

Кожна з цих утиліт буде складатись із власного незалежного вихідного тексту та спиратись на наявну у системі стандартну бібліотеку мови C. Разом ці утиліти будуть формувати пакет, що буде збиратись у спільний архів та використовуватись у якості образу базової файлової системи для операційної системи.

Програмне оточення складатиметься з наступних утиліт:

Таблиця 2.7

Опис утиліт оточення розроблюваної операційної системи

Утиліта	Призначення
/bin/cat	вивід та поєднання даних з файлів
/bin/com	надання засобів для комунікації через послідовні порти
/bin/cp	копіювання об'єктів файлової системи
/bin/date	вивід інформації про поточний час та дату
/bin/grep	фільтрування рядків
/bin/head	вивід перших рядків із входу
/bin/hexd	вивід шістнадцятеричного дампу вхідних даних
/bin/ls	вивід списку елементів каталогу
/bin/mkdir	створення каталогів
/bin/mkfifo	створення FIFO-файлів для міжпроцесної комунікації
/bin/netctl	керування мережевими інтерфейсами
/bin/netdump	вивід дампу пакетів із мережевого інтерфейса
/bin/netcat	вивід даних із віддаленої частини мережевого сокета
/bin/pager	надання можливості для сторінкового переглядання тексту із навігацією

Таблиця 2.7 (Завершення)

/bin/ps	вивід інформації про активні процеси
/bin/rm	видалення об'єктів із файлової системи
/bin/sha256sum	вивід SHA256—хешсуми вхідних даних
/bin/showkey	вивід інформації про ввід у термінал
/bin/sh	надання оболонки для взаємодії з усіма командами
/bin/sleep	очікування
/bin/su	підвищення привілей користувача або зміна користувача
/bin/table	форматування вхідних даних у вигляді таблиці
/bin/tee	дублювання вхідних даних у додаткові виводи
/bin/ucat	комунікація із віддаленим кінцем локального Unix-сокету
/bin/uname	вивід основної інформації про систему
/bin/zcat	вивід даних, що було зжато у формат gzip
/sbin/acpid	фоновий сервіс, що прослуховує події ядра з питань керування живленням, наприклад, натиснення особливих клавіш або перегрів процесора
/sbin/insmod	завантаження додаткових модулів ядра
/sbin/login	надання користувацької сесії на терміналі
/sbin/lspci	вивід інформації про під'єднані до шини PCI пристрої
/sbin/mount	під'єднання додаткових файлових систем
/sbin/reboot	керування живленням системи: перезавантаження або вимкнення
/sbin/umount	від'єднання файлових систем

Особливе значення має також утиліта `/bin/sh`, бо вона містить в собі інші вбудовані команди, а також забезпечує основну взаємодію користувача із системою через виклики інших команд.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						41
Зм.	Арк.	№ докум.	Підп.	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 2

У розділі 2 з урахуванням проведеного попереднього аналізу теоретичної бази було сформовано архітектуру для подальшої розробки компонентів операційної системи. Було описано ключові елементи цієї системи, розподіл привілей процесора між ними, а також приведено найбільш ефективні методи взаємодії між цими компонентами. У розділі наведено три основних компоненти:

- Ядро системи, що має безпосередній доступ до ресурсів системи та виконується на вищому рівні привілей;
- Стандартна бібліотека мови C, що слугує базою для побудови користувацьких програм, та абстрагує від розробника деталі імплементації конкретного ядра системи;
- Користувацьке програмне забезпечення та оточення системи, що виконується на нижчому рівні привілей та через інтерфейс системних викликів може робити запити до системних ресурсів.

Дана архітектура в достатній мірі реалізує мету цього проєкту, та є простою і гнучкою у можливостях її реалізації.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						42
Зм.	Арк.	№ докум.	Підп.	Дата		

РОЗДІЛ 3. РОЗРОБКА КОМПОНЕНТІВ ОПЕРАЦІЙНОЇ СИСТЕМИ

Спершу було розроблено найбазовішу частину операційної системи — її ядро. До його складу входять описані у Розділі 2 архітектурні компоненти та підсистеми, що будуть описані у далі цьому розділі.

Ядро було розроблено на мові програмування C з використанням компілятора GCC 9.3.0. Розробка виконувалася на ОС Gentoo Linux. Результатом збірки вихідного коду ядра є бінарний файл формату ELF (Executable and Linkable Format), де-факто стандартний формат виконуваних та об'єктних файлів у Unix-подібних системах, що далі буде завантажений відповідним модулем завантажувача.

Виходячи з неможливості завантаження ELF напряду через UEFI, для ядра також було розроблено додатковий завантажувач на мові програмування Rust. Основною його задачею буде завантаження у пам'ять бінарного образу ядра системи разом із початковим RAM-диском та реалізація спеціально розробленого протоколу завантаження.

Окрім, власне, ядра системи, було розроблено набір користувацьких утиліт та бібліотека мови C, на основі якої їх було побудовано. Архітектура, принципи роботи та інтерфейси в цьому розділі спираються на широкий ряд описаних у стандарті POSIX форматів та функцій.

3.1. Ядро системи

3.1.1. Основні етапи роботи ядра

Під час запуску апаратної платформи, після проходження процедур POST та необхідних етапів запуску, що визначені в BIOS системи, в пам'ять зчитується завантажувач системи, задача якого — довантажити власні частини, а потім завантажити ядро та його необхідні супутні файли, наприклад, початковий RAM-диск. Після цих процедур керування системою передається власне завантаженому ядру. Цей момент часу і є початком роботи ядра, дії після якого будуть описані.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						43
Зм.	Арк.	№ докум.	Підп.	Дата		

Після завантаження та прийняття керування процесором:

1. Ядро налаштовує базовий функціонал процесора та системи для подальшої роботи:
 1. Налаштування стеку для викликів та локальних змінних. Це виконується ще в асемблері, бо компілятори неявно використовують стек у результуючому машинному коді.
 2. Для отримання відлагоджувального виводу на етапах, коли вивід на екран ще не доступний, налаштовується послідовний інтерфейс. Для x86 використовується порт COM1.
2. Ядро визначає з інформації, що воно отримало від завантажувача, кількість та регіони доступної фізичної пам'яті, адресу та розміри буфера графічного дисплея та інші параметри, які згодом будуть використані.
3. Отримавши інформацію про наявний дисплей, ядро налаштовує вивід тексту на нього для ранніх етапів завантаження, коли повноцінне керування пам'яттю ще не є доступним.
4. Маючи список з доступних регіонів пам'яті, ядро додає їх до менеджера фізичної пам'яті.
5. Ядро налаштовує таблиці дескрипторів, переривань, а також встановлює таблиці для відображення віртуального адресного простору.
6. Після того, як ядро отримало повні можливості для керування пам'яттю, дисплей ініціалізується заново, але в цей раз з буфером повного розміру замість того, що був виділений статично для ранніх етапів.
7. Ініціалізуються різні підсистеми та драйвери: віртуальна файлова система, драйвери пристроїв апаратної платформи, пристроїв PCI та USB, отримується поточний час та інше.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						44
Зм.	Арк.	№ докум.	Підп.	Дата		

8. Налаштовується планувальник процесів, інтерфейс системних викликів, підсистема керування користувацькими процесами та створюється основний процес, що продовжить ініціалізацію системи — `init`.
9. Процес `init` монтує кореневу файлову систему і завантажує вказаний бінарний файл з користувацькою програмою ініціалізації, після чого запускає її, знижуючи власні привілеї процесора до рівня користувача.

Надалі основну частину процесорного часу займатимуть користувацькі програми, а ядро повністю завершило свій процес запуску. Після цього ядро може приймати керування процесором через:

1. апаратні переривання;
2. виключення процесора;
3. системні виклики;

3.1.2. Огляд менеджера фізичної пам'яті

Менеджер фізичної пам'яті виконує функцію виділення за звільнення її сторінок або ж нерозривних їх регіонів. Окрім цього, цей компонент відслідковує кількість користувачів сторінок, мету їх використання та інші параметри. Також ця підсистема має функції для раннього резервування окремих регіонів пам'яті, щоб захистити їх від розподілення та потенціального ушкодження. Такими регіонами є, наприклад, код самого ядра або образ RAM-диску в пам'яті.

Після резервування важливих частин пам'яті, менеджер фізичної пам'яті, з урахуванням інформації про доступні регіони від завантажувача, спершу визначає загальний розмір пам'яті у сторінках, після чого дізнається кількість сторінок, що необхідно виділити в найпершу чергу для власної роботи.

Для відслідковування та виділення сторінок менеджер фізичної пам'яті ставить у відповідність кожній фізичній сторінці розміром 4 Кб наступну структуру даних:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						45
Зм.	Арк.	№ докум.	Підп.	Дата		

Структура даних з описом фізичної сторінки пам'яті

Назва поля	Зсув	Розмір	Використання
flags	0	4	Визначення особливих рис сторінки, наприклад: • чи була вона виділена через <i>mmmap</i> (2) • чи зайнятий кадр цієї сторінки
usage	4	4	Мета, з якою цю сторінку було виділено: • відсутня (якщо сторінка не зайнята) • приватна для якогось процесу • спільна між процесами • використовується драйвером якогось пристрою • використовується безпосередньо ядром • містить структури для віртуальної пам'яті • є сторінкою кешу для блокових пристроїв
refcount	8	8	Кількість активних користувачів цієї сторінки

Структури даних подібного вигляду використовуються у ядрі Linux (struct page). Такий підхід дає можливість за мінімальну кількість дій отримати інформацію про будь-яку сторінку фізичної пам'яті, маючи лише її адрес.

Після виділення простору для власних структур, менеджер фізичної пам'яті налаштовує вказані вище структури даних для кожної з наявних сторінок пам'яті згідно з їх початковим станом.

Виділення сторінки після таких операцій зводиться до пошуку такої структури, в якій не виставлений флаг "зайнятий", позначення її, як виділеної, та повернення адреси відповідної сторінки викликачеві.

Звільнення сторінки складається з перевірки числа її поточних користувачів: сторінку насправді буде звільнено лише тоді, коли це значення досягне нуля. За відсутності користувачів, сторінка втрачає свій флаг "зайнятий" та повертається системі.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						46
Зм.	Арк.	№ докум.	Підп.	Дата		

Також варто відзначити використання цієї функції у комбінації з відповідними змінами до таблиць віртуальної пам'яті при роботі з клонуванням адресних просторів програм, що робить fork(2). При клонуванні приватних сторінок користувацьких програм, замість їх виділення нової сторінки та копіювання даних зі старої, відображення нової віртуальної сторінки налаштовується так, щоби використовувати ту саму "стару" фізичну сторінку, при цьому і для програми-оригіналу, і для програми-клона забороняється запис до цієї сторінки. Це дає змогу зберегти пам'ять від зайвих копій, особливо в регіонах пам'яті, куди ніколи не робиться запис, наприклад, сегменти коду програм. Якщо ж якась програма спробує провести запис до таких сторінок, це призведе до виключення процесора #PF (Page Fault), в результаті якого сторінка вже насправді буде сконована. Весь цей принцип є доволі поширеним та носить назву CoW, або Copy-on-Write.

3.1.3. Огляд менеджера віртуальної пам'яті

Для забезпечення можливостей для виконання декількох програм одночасно, необхідно забезпечити виділення для кожної з них власного віртуального адресного простору. В реалізації цього проєкту, кожен користувацький процес отримує власний адресний простір. Кожен з адресних просторів можна поділити на два основних регіони: локальні відображення, що властиві конкретним адресним просторам, та глобальний регіон, що відображає перші гігабайти фізичної пам'яті у верхній адресний простір.

Таблиця 3.2

Розподіл регіонів віртуальної пам'яті у операційній системі

<i>Регіон</i>	<i>Використання</i>
0000000000000000 ₁₆ .. 0000000100000000 ₁₆	Код, дані та стек програм
0000000100000000 ₁₆ .. FFFFFFF000000000 ₁₆	Динамічні регіони даних для програм
FFFFFFF000000000 ₁₆ .. FFFFFFFF00000000 ₁₆	Відображення нижньої фізичної пам'яті

Останнє відображення зумовлено тим, що такий підхід надає можливості для ядра без зайвих перемикань отримати доступ до будь-якого фізичного адресу шляхом простого додавання фіксованого зсуву. У проєкті в якості такого зсуву використовується число FFFFFFF000000000₁₆, що займає дві найвищі сторінки розміром 512 Гб всіх адресних просторів. Такий підхід також є доволі поширеним в інших операційних системах та їх ядрах, наприклад, Linux також використовує подібну модель як для 32-бітних адресних просторів, так і для 64-бітних.

Основні функції, якими займається менеджер віртуальної пам'яті — маніпуляції з такими адресними просторами: додавання віртуальних відображень, їх видалення, пошук фізичних адресів, до яких транслюються конкретні віртуальні адреси, а також клонування цих адресних просторів. Як вже було зазначено вище, клонування адресних просторів проводиться з використанням техніки CoW задля забезпечення швидкодії та зниження використання ресурсів системи.

3.1.4. Робота з перериваннями та виключеннями процесора

Виконання звичайного потоку інструкцій процесором може бути перервано рядом зовнішніх подій. Ці переривання можна розділити на два основні класи:

1. Апаратні переривання, викликані якимись діями користувача або сигналами про зміну стану апаратного забезпечення;
2. Програмні переривання — події, що були викликані виключеннями процесора, наприклад, спробою інструкції доступитись за невідображеною адресою або нестачею привілей процесора для її виконання.

Всі ці події мають відповідні обробники у таблиці дескрипторів переривань, що дозволяє задати до 256 так званих “векторів” переривань. Перші 32 з них завжди є зарезервованими для обробників виключень

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						48
Зм.	Арк.	№ докум.	Підп.	Дата		

процесора: ділення на нуль (мнемоніка #DE, номер 0), доступ за невідображеною адресою (#PF, 14), загальна помилка доступу (#GP, 13) тощо.

Для опрацювання зовнішніх переривань у системах x86 традиційно використовувалася пара із двох чипів Intel 8259. Кожен із них мав 8 фіксованих ліній переривань та регістр маскування переривань та регістр зміщення базового номеру вектора. Ролі цих контролерів поділялись на master та slave, що використовувалося у так званому каскадуванні переривань, тобто переривання, що проходило на slave-контролер, проходило також через master.

Далі буде наведено список переривань, доступних у Intel 8259:

Таблиця 3.3
Переривання у Intel 8259

№	Призначення	№	Призначення
0	Системний таймер	8	Таймер CMOS
1	Клавіатура PS/2	9	Периферія
2	Каскадування	10	Периферія
3	COM2	11	Периферія
4	COM1	12	Миша PS/2
5	LPT2	13	FPU/сопроцесор
6	Гнучкий диск	14	Основний диск АТА
7	LPT1	15	Додатковий диск АТА

Як вже можна побачити з цієї таблиці, такий підхід є дуже обмежуючим та не дозволяє ефективно використовувати велику кількість периферійних пристроїв. До того ж, Intel 8259 не може працювати із багатопроцесорними SMP-системами. Ця пара контролерів була розроблена із урахуванням лише однопроцесорних систем, тому не має ніяких засобів для маршрутизації переривань.

З урахуванням цих особливостей та вимог більш сучасних систем, Intel розробила специфікацію та чіпи для більш ефективної обробки переривань, які й використовуються у розробленому ядрі. Для цього у кожному із процесорів розміщується чіп Intel Local APIC (LAPIC), а на самій периферійній шині надається Intel I/O APIC (IOAPIC). Перший керує локальними перериваннями

та надає можливість викликати міжпроцесорні переривання (IPI). Другий дає можливість маршрутизувати глобальні лінії апаратних переривань на конкретні процесори та вектори. Це дозволяє дуже ефективно балансувати використання процесорного часу обробниками переривань, розподіляючи навантаження між ними.

У розробленому ядрі підсистема обробки апаратних переривань надає можливість зручно встановлювати функції-обробники переривань для глобальних ліній, автоматично балансуючи їх на найменш завантажені процесори. Також ця підсистема надає можливості для використання “старих” ліній переривань, що існують для сумісності, наприклад, лінії переривань для COM1 або клавіатури PS/2.

Додатково варто зазначити обробку переривань від пристроїв на шині PCI. Як було зазначено в розділі архітектуру системи, пристрої PCI можуть використовувати як звичайні лінії переривань, так й особливий метод Message-Signalled Interrupt (MSI). При використанні I/O APIC, операційна система мусить звернутись до таблиць маршрутизації у ACPI, аби дізнатись про те, як лінії переривань від пристроїв PCI з’єднуються із лініями глобальних переривань на контролері. Функція для призначення обробнику PCI-переривань автоматично враховує наявність можливості використання MSI, маршрути ліній переривань та встановлює найбільш ефективний метод для обробки переривання із кожного конкретного пристрою [7].

3.1.5. Огляд драйверів пристроїв та файлових систем

В цьому розділі буде розглянуто реалізовані у ядрі драйвери пристроїв та файлових систем. Будуть розглянуті наступні підсистеми:

- Драйвер контролеру клавіатур та миш PS/2;
- Драйвер послідовного порту;
- Загальний драйвер консолі;
- Текстовий дисплей x86;

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						50
Зм.	Арк.	№ докум.	Підп.	Дата		

- Графічний драйвер Vesa;
- Драйвер шини PCI;
- Драйвер підсистеми керування живленням ACPI;
- Драйвери системних таймерів;
- Драйвер годинника RTC;
- Дискова підсистема:
 - Драйвер хост-контролеру ATA AHCI;
 - Драйвер хост-контролеру ATA IDE;
- Драйвер мережевого інтерфейсу Realtek RTL8139;
- Драйвер файлової системи gamfs;
- Драйвер файлової системи ext2.

Далі буде описано деякі важливі підсистеми та драйвери апаратних засобів.

Драйвер контролеру клавіатур та миш PS/2

Цей драйвер відповідає за функціонування клавіатур та миш, що під'єднуються до порту PS/2 через контролер Intel 8042 та сумісні з ним. Основним інтерфейсом комунікації з контролером такого типу є порти вводу-виводу x86:

- Порт 64₁₆ використовується для інформування про статус та встановлення налаштувань контролера;
- Порт 60₁₆ використовується для читання змісту внутрішнього регістру даних контролеру, куди при натисканнях, повторях або віджиманнях клавіш надсилаються повідомлення;

Окрім цих портів, контролер також дає дві лінії для переривань для обох своїх каналів. Так, наприклад, можна одночасно під'єднати і клавіатуру, і миш через один й той самий контролер.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						51
Зм.	Арк.	№ докум.	Підп.	Дата		

Вихід клавіатури, якщо така наявна, програмно під'єднується до вводу до терміналу операційної системи, а для подій про переміщення та натискання кнопок миші створено файл пристрою /dev/ps2aux.

Драйвер послідовного порту

Цей драйвер дозволяє системі вести комунікацію через послідовні порти, наявні у апаратних платформах на базі x86. Основним таким портом є COM1. Цей порт використовується для виводу відлагоджувальної інформації та логу подій ядра ще з найперших етапів його запуску, аби мати якнайбільше корисної інформації при розробці. Окрім цього, цей порт надає можливість до вводу даних ззовні, тобто його можна використовувати як свого роду термінал. Так, наприклад, за допомогою програми login(1) можна запустити окрему сесію користувача із вводом та виводом на такий порт.

Контролер портів надає по парі регістрів для кожного порту: один для керування, інший для вводу та виводу даних. Через перший регістр можливе детальне підлаштування цього порту під необхідні параметри, наприклад, стоп-біти або частота роботи.

Окрім регістрів, контролер також має два переривання, щоби сигналізувати про помилки або наявність даних на вході портів.

Для кожного такого порту створено файл-пристрій псевдотерміналу /dev/ttySN, де N — номер порту від 0 до 3.

Загальний драйвер консолі

Драйвер консолі дозволяє під'єднувати пристрої вводу та виводу до відповідних частин операційної системи, аби вони разом формували набір користувацьких терміналів. На кожному з цих терміналів користувач може запустити окрему сесію, а самі сесії можуть виконуватись паралельно, надаючи можливості для одночасного використання системи багатьма користувачами.

Звичайні апаратні термінали надають можливості вводу та виводу в одному пристрої, тому є найпростішим способом комунікацій, але для цього

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						52
Зм.	Арк.	№ докум.	Підп.	Дата		

драйвера необхідно ще було реалізувати можливість під'єднання окремих пристроїв вводу до окремих пристроїв виводу, як, наприклад, клавіатур PS/2 або USB та окремих графічних або текстових дисплеїв.

Всі пристрої цього драйверу є символьними та дозволяють вводити та виводити лише по одному елементу за раз, а файли цих пристроїв мають префікс `/dev/tty*`, при цьому закінчуються вони на номер відповідного терміналу у системі, наприклад, `/dev/tty1` або `/dev/ttyS0`.

Текстовий дисплей x86

Апаратна платформа x86 для зручності розробників та сумісності із старими програмами має реалізацію текстового дисплею розміром 80×25 символів із 16 кольорами. Для виводу інформації на цей дисплей надається регіон у пам'яті відповідного розміру за адресою $B8000_{16}$. Цей прямокутний регіон з символів лінійно розміщується у пам'яті, кожен елемент такого буферу має розмір 2 байти: перший містить інформацію про сам символ, а другий зберігає атрибути кольорів фону та символу [8].

Далі описано кольори, що доступні для використання у такому режимі роботи дисплею:



Графічний дисплей Vesa

Альтернативним методом виводу інформації на дисплей буде використання графічного дисплею, що ядро отримує від завантажувача. Такі дисплеї є більш гнучкими у засобах відображення інформації, але через складність реалізації конкретних драйверів потребують запитувати доступ до такого дисплею ще на ранніх етапах завантаження.

Ці дисплеї видають системі буфер у фізичному адресному просторі, де лінійно розміщуються дані про кольори окремих пікселів на екрані. Зазвичай використовується 32-бітний або 24-бітний формати з доповненням до 32 біт, де

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						53
Зм.	Арк.	№ докум.	Підп.	Дата		

кожен з каналів (червоний, зелений, синій) кольору задається одним байтом. Цей регіон пам'яті називається фреймбуфером.

Для пришвидшення роботи таких дисплеїв, особливо при прокрутці консолі або малюванні великих зображень, ядро системи вимикає кешування пам'яті для уникання непотрібних операцій, а також налаштовує подвійну буферизацію для виключення “розривання” екрану під час малювання або очищення.

Для можливості відображення тексту, ядро містить вбудований шрифт, кожен символ якого являє собою бітову маску з пікселів, що будуть або не будуть виводитись для кожного з них. Таким чином досягається вивід текстових даних разом із графічними на той самий дисплей.

Окрім доступу до екрану з боку ядра, воно надає таку можливість і для користувацьких програм. Так, наприклад, за допомогою файлу-пристрою `/dev/fb0` програма може захопити контроль над дисплеєм, запитати в ядра відображення пам'яті фреймбуфера на власний адресний простір за допомогою `mmap(2)`, після чого має змогу власноруч малювати на дисплеї зображення.

Драйвер шини PCI

Шина PCI є одним із найпоширеніших способів для під'єднання зовнішніх периферійних пристроїв до систем на базі x86. Ядро, що розроблюється, підтримує цю технологію у двох її версіях: власне PCI та PCI Express.

Ця шина має контролер з боку процесора, що роздає пристроям адреси на шині та забезпечує різноманітні можливості до комунікації з ними, а також надає у пам'яті або у просторі портів вводу-виводу набір регістрів для отримання інформації про ці пристрої.

Основна задача, яку виконує ядро під час ініціалізації підсистеми для роботи з шиною PCI — отримання повного переліку пристроїв на цій шині. Окрім кореневого комплексу, до якого напряму під'єднуються пристрої, шина

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						54
Зм.	Арк.	№ докум.	Підп.	Дата		

також містить так звані “бріджі”, що дозволяють, жертвуючи невеликою частиною швидкодії, під’єднувати значно більші кількості пристроїв.

Вище вказані регіони даних про пристрої містять таку інформацію, як ідентифікатор виготовника та пристрою, його клас та підклас, а також дані про ресурси, які використовує цей пристрій, наприклад, адреси регіонів пам’яті, які система виділила пристрою, або набір переривань, що з нього можуть надійти.

В загальному випадку, типовий пристрій має блок даних наступного формату, що його описує:

Таблиця 3.4

Структура даних з описом пристрою PCI

Зсув	Біти 31..24	Біти 23..16	Біти 15..8	Біти 7..0
0	ID пристрою		ID виробника	
4	Статус		Команда	
8	Клас	Підклас	Прог. Інтерфейс	Версія пристрою
12	BIST	Тип заголовка	Latency Timer	Розмір лінії кешу
16	Регістр адреси №2			
20	Регістр адреси №1			
24	Регістр адреси №2			
28	Регістр адреси №3			
32	Регістр адреси №4			
36	Регістр адреси №5			
40	Показник Cardbus CIS			
44	ID підсистеми		ID виробника підсистеми	
48	Адреса додаткового ROM			
52	Зарезервовано			Вказівник Caps
56	Зарезервовано			
60	Макс. затримка	Мін. зат. видачі	Пін перерив.	Лінія перерив.

Функціонування переривань з боку системи також залежить від підтримки ядром зчитування інформації про маршрутизацію переривань відповідним контролером із підсистеми ACPI.

Основною відмінністю PCI Express від попередньої його версії є підняття частоти комунікації з пристроями та розширення можливостей контролерів. При цьому з боку ядра змінюється інтерфейс комунікації — якщо раніше для роботи з інформацією про пристрої ядро використовувало два порти вводу-виводу, то починаючи з PCI Express ця інформація надається через регістри у пам'яті. Також новою особливістю є підтримка нового типу переривань MSI — Message Signalled Interrupt. Ці переривання, якщо налаштовані для пристрою, можуть проходити повз контролер переривань, потрапляючи прямо на процесор, шляхом запису до заданого в конфігурації регіону пам'яті, тобто пристрій може ініціювати процедуру переривання, просто написавши відповідне значення у регістр локального APIC бажаного процесору. Це значно пришвидшує та спрощує обробку апаратних переривань від пристроїв на шині.

Окрім цього, ще одним основним методом комунікації пристроїв із хост-системою є прямий доступ до пам'яті (DMA). Для можливості використання цього режиму роботи з боку пристрою, ядро системи мусить ввімкнути біт “bus mastering” у конфігураційному просторі цього пристрою на шині. Після вмикання цієї функції пристрою буде дозволено проводити операції з пам'яттю напрямку, уникаючи процесор, що значно пришвидшує ввід-вивід даних та є конче необхідною функцією у сучасних системах.

Додатково ядро надає у sysfs можливості для перегляду списків та інформацію про наявні та під'єднані пристрої PCI, а також драйвери, що ними керують, у директорії /sys/pci/*.

Драйвер файлової системи ramfs

Для надання базового набору утиліт системи разом з ядром до пам'яті також завантажується образ файлової системи, що являє собою архів формату ustar, описаний у стандарті POSIX.1-1998. Після проходження базової ініціалізації, ядро системи “розпаковує” цей архів, формуючи з нього дерево ефемерної файлової системи, що існує лише у пам'яті комп'ютера. Драйвер файлової системи ramfs при розпаковці такого архіва збирає списки вказівників

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						56
Зм.	Арк.	№ докум.	Підп.	Дата		

на блоки з даними файлів, а із шляхів, вказаних у архіві, створює ієрархічну структуру.

Також, завдяки тому, що всі файли збираються у списки блоків, це надає можливість протягом роботи системи виділяти нові блоки та змінювати розмір файлів, що в свою чергу дозволяє вносити зміни у файли в цій файлової системі.

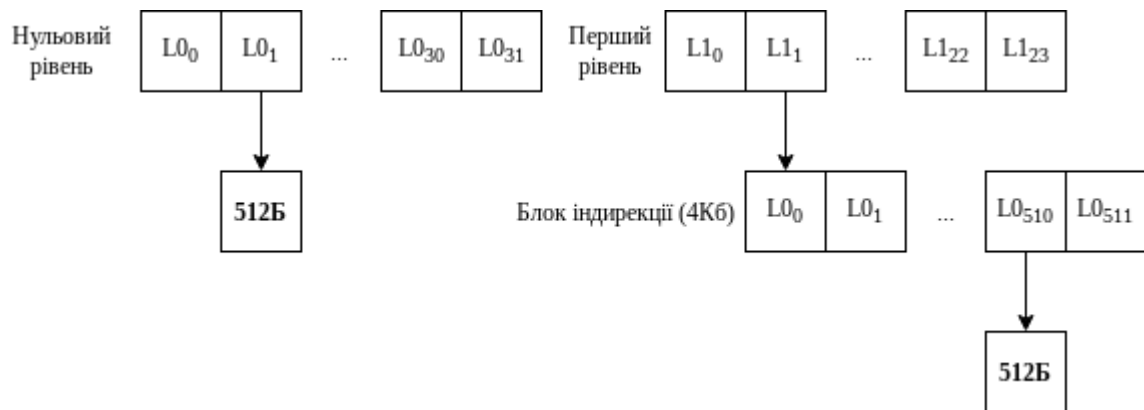


Рис. 3.1 Рівні вказівників на блоки у ramfs

Як видно із рис. 3.1, дані кожного файлу поділяються на блоки розміром 512 байт, що дорівнює розміру блока у архівах ustar. Для зберігання списку блоків використовується два типи вказівників — 32 прямих вказівники та 24 вказівники з першим рівнем індирекції. Кожен із непрямих вказівників посилається на блок розміром 4Кб, де зберігаються 512 прямих вказівників. Такий підхід дає можливість адресувати до 12320 блоків, що дає максимальний розмір файлу трохи більший за 6Мб, чого є цілком достатньо для виконання задач цієї файлової системи.

Драйвер файлової системи ext2

В якості резидентної файлової системи через великий ряд переваг (невелика складність імплементації, висока швидкодія та гнучкість) було обрано ext2 [9], що також стала де-факто стандартною файловою системою у Linux в 1990-2000х, після чого була замінена на ext3.

Основні структури даних, що визначає стандарт цієї ФС: блок, група блоків, inode. Кожен блок є фіксованого розміру, що визначається в параметрах

файлової системи, а самі блоки поєднуються в групи блоків. Стандартний розмір блока в ext2 — 1024 байти, але може бути змінений, а в наступних версіях цієї файлової системи його було підвищено до 4Кб. Кожна група блоків містить спеціальні блоки, що за допомогою бітмапів контролюють виділення блоків та inode у певній групі. В таких бітмапах кожен 0 відповідає вільному блоку або inode, а 1 — зайнятому. Файли у ext2 можуть мати 7 різних типів:

- Звичайний файл;
- Директорія;
- Мережевий сокет;
- Файл-FIFO;
- Блоковий пристрій (наприклад, диск);
- Символьний пристрій (наприклад, термінал);
- Символічне посилання.

Кожен із цих типів файлів може мати свій набір блоків даних, що він використовує, а інформація про сам об'єкт зберігається у inode, що окрім розміру та посилань на блоки, що він займає, містить також інформацію про його власника, час останньої модифікації та інше.

В кожному файлі для перших 12 блоків використовується безпосередня адресація блоків — це означає, що адресу блока з даними записано прямо до inode. Для блоків після дванадцятого використовується непряма адресація з кількома можливими рівнями — адреси блоків з реальними даними записуються до ще одного блока, що може містити багато таких посилань, а в inode зберігається посилання на цей блок індирекції. Якщо ж не вистачає й одного рівня індирекції, ext2 передбачає можливість “вкладення” таких блоків із вказівниками до третього рівня, коли inode “ланцюгом” вказує на певний блок через три проміжних рівня. Максимальний розмір файлу з таким підходом залежить від розміру блоку у конкретній файловій системі, для стандартного розміру в 1024 байти він становить 2048Гб (тобто 2Тб).

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						58
Зм.	Арк.	№ докум.	Підп.	Дата		

В самій структурі inode ніколи ж не зберігається інформація про, власне, назву файлу, бо ця задача відведена до ще однієї структури даних — dirent (Directory Entry). Ці записи зберігаються в блоках даних директорій, та зв'язують імена записів у директоріях з їх конкретними об'єктами у файловій системі за допомогою їх номерів inode на диску.

Ініціалізація носія з такою файловою системою починається із читання суперблоку, що є 1024-байтовим блоком за зсувом 2 логічних блоки від початку розділу, для отримання базової інформації про файлову систему. Після виділення відповідних структур даних у пам'яті та обчислення параметрів файлової системи, драйвер зчитує кореневу директорію (inode 2) з диску.

Для пришвидшення роботи з резидентними файловими системами на носіях інформації, підсистемою для роботи із блоковими пристроями також надається кешування блоків для файлових систем. Це є особливо корисним для роботи ext2, бо дозволяє значно прискорити читання та запис, особливо на не дуже швидких носіях.

3.1.6. Робота з процесами та потоками

Базовою одиницею виконання у розробленій операційній системі є потік. Потоки групуються у процеси, що дають для них загальні ресурси та оточення: адресний простір, файлові дескриптори тощо.

Кожен процес системи, з точки зору ядра, описується власною структурою struct process:

Таблиця 3.5
Структура даних процесу

Поле	Призначення
space	Адресний простір процесу
image_end, brk	Параметри, що описують кінець образу програми у пам'яті та кінець сегменту даних
proc_state	Стан процесу
thread_list, thread_count	Список потоків цього процесу

Таблиця 3.5 (Завершення)

shm_list	Список регіонів спільної пам'яті цього процесу
ioctx, fds	Файлові дескриптори та контекст процесу
pid_notify	Дескриптор очікування на завершення процесу
sigq	Черга сигналів для обробки цим процесом
fs_entry	Запис процесу у procfs
ctty	Термінал, до якого прив'язаний цей процес
name	Ім'я процесу
pid, pgid	Номер процесу та його групи
parent, first_child, next_child	Формують дерево процесів
exit_status	Код завершення процесу
g_link	Поеднує процеси у глобальний список

Кожен із потоків такого процесу має наступну структуру, що його описує (struct thread):

Таблиця 3.6
Структура даних потоку

Поле	Призначення
data	Контекст потоку: його стек, регістри та ін.
state	Стан потоку
sleep_deadline, sleep_notify	Інформація для блокування потоку на якийсь час
proc	Процес, до якого відноситься цей потік
wait_head	Дескриптор, на який в даний момент очікує потік
thread_link	Формує список із потоків у процесі
signal_entry, signal_stack_base, signal_stack_size	Контекст для обробки сигналів у цьому потоці
sigq	Черга локальних сигналів потоку
flags	Особливі флаги потоку
cpu	Номер процесору, на якому в даний момент виконується цей потік
sched_prev, sched_next	Формують список із потоків у черзі планувальника

Потоки в системі мають відповідні стани, що зображено нижче:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						60
Зм.	Арк.	№ докум.	Підп.	Дата		

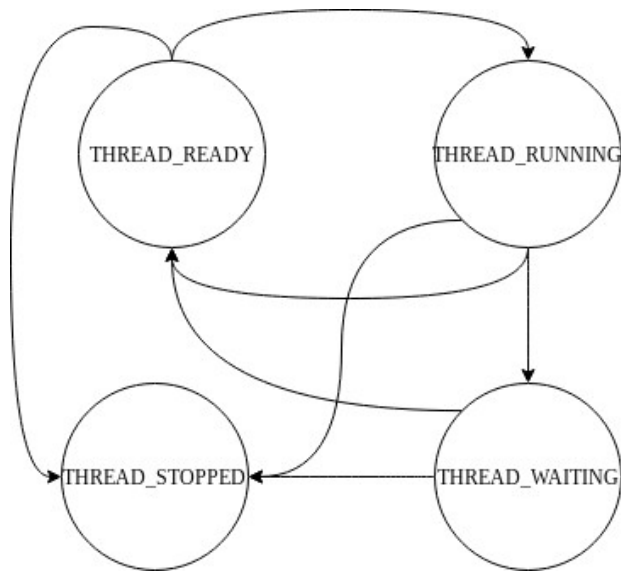


Рис. 3.2. Стани потоків

Як видно із рис. 3.2, після створення потік отримує статус “готовий”, після чого за рішенням планувальника може бути переведений у стан виконання. Із цього стану можливі наступні переходи:

- Якщо вплив час кванту виконання потоку, він буде поміщений назад до черги планувальника із статусом “готовий”;
- Якщо потік зробить системний виклик для власного завершення або отримає фатальний сигнал, що не зможе обробити, його буде завершено, та помічено, як “зупинений”;
- Якщо ж потік зробить системний виклик, що змусить його очікувати, наприклад, вводу від користувача або певного часу, він буде помічений як “очікуючий” та видалений із черги виконання до настання відповідної події.

Згідно із цією схемою станів, у чергах планувальника та у виконанні процесором можуть знаходитись потоки лише із двома можливими станами: “готовий” та “виконується”.

Аналогічно для станів потоків, було запроваджено стани процесів:

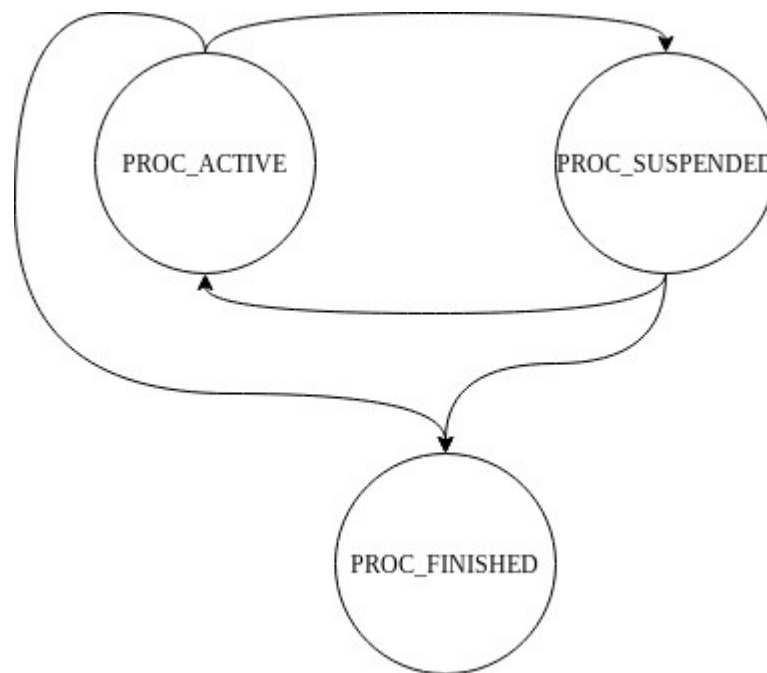


Рис. 3.3. Стани процесів

Процес, що тільки було створено, отримує статус “активний”. Це означає, що на даний момент у цього процесу є хоча б один “живий” потік та він може продовжувати виконання. Якщо користувач за допомогою терміналу вирішує поставити процес “на паузу”, йому надсилається відповідний сигнал, а сам процес становиться “призупиненим”. Якщо всі потоки процесу завершуються, він отримує статус “завершений”. Завершені процеси продовжують існувати у пам’яті до моменту, коли їх батьківський процес не збере їх за допомогою waitpid().

3.1.7. Реалізація інтерфейсу системних викликів

Як було вказано у Розділі 2, для надання можливостей взаємодії з апаратною платформою для користувацьких програм створено інтерфейс системних викликів. З боку ядра встановлюється обробник інструкції SYSCALL, що підвищує режим привілей роботи процесора та передає на цей обробник керування [4].

Робота цієї інструкції в 64-бітному режимі регулюється наступними трьома спеціальними регістрами:

Регістри налаштування інструкцій SYSCALL/SYSRET

<i>Регістр</i>	<i>Призначення</i>
STAR (C0000081 ₁₆)	Основи для сегментів переходів між CPL0 та CPL3
LSTAR (C0000082 ₁₆)	64-бітна адреса точки входу в обробник системного виклику
SFMASK (C0000084 ₁₆)	Бітова маска для очищення бітів у регістрі флагів під час входу в обробник. Використовується для блокування апаратних переривань перед входом в обробник системного виклику

На момент входу в обробник системного виклику, процесор поміщає до RCX старе значення регістру RIP, а у R11 зберігається старе значення регістру флагів RFLAGS. При цьому інші регістри залишаються без змін, в тому числі незмінним залишається показник стеку, що продовжує вказувати на користувацький стек, тому обробник системних викликів в першу чергу вносить зміни до RSP, аби зберегти користувацький стек та почати використання стеку ядра. На цьому стеці зберігається стан регістрів, необхідних для відновлення роботи після виходу з обробника, а після цього керування передається обробнику конкретного системного виклику відповідно до його номеру у таблиці.

Для передавання параметрів у обробник системних викликів вони розміщуються спеціальним чином, але порядок аргументів у регістрах є відмінним від звичного інтерфейсу викликів, тому узагальнений обробник перед передачею керування якомусь конкретному робить необхідні переміщення у регістрах, аби зробити виклик сумісним із стандартними процедурами викликів функцій у мові C [10].

Як видно з таблиці, RAX визначає номер конкретного обробника системного виклику з таблиці системних викликів. Якщо цей регістр має значення, що виходять за межі таблиці або посилаються на неіснуючі системні виклики, ядро посилає процесу сигнал про помилку.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						63
Зм.	Арк.	№ докум.	Підп.	Дата		

Завершенням виконання обробнику системного виклику є виконання інструкції SYSRET, що повертає керування назад до користувачької програми.

Для апаратної платформи на базі x86 існують також й альтернативні методи для надання інтерфейсу системних викликів, але для архітектур з пам'яттю без сегментації (мається на увазі сегментація, що надавалась у 16-бітних процесорах до Intel 80386 та 32-бітних x86-процесорах) пара з інструкцій SYSCALL та SYSRET є найбільш ефективною та рекомендується для використання у посібнику для розробників під цю платформу від AMD [4].

Однією з таких альтернатив є використання програмного переривання процесора, що було стандартним способом для виконання системного виклику у 32-бітних системах на базі x86. Як приклад такого методу можна навести системні виклики у 32-бітній версії операційної системи FreeBSD для x86: користувачька програма поміщає на стек параметри до системного виклику, а у EAX записує його номер, після чого виконує інструкцію `int 80h` [11]. Програми також можуть поміщати аргументи до регістрів замість стеку, тому що вони зазвичай працюють набагато швидше, як це робиться у випадку з ядром Linux.

Такий метод, хоча і є на перший погляд дуже схожим на виклик через SYSCALL, є набагато повільнішим, бо генерація та обробка програмних переривань старими методами виконує дуже велику кількість дій, що не були б необхідними для швидкого перемикання до режиму ядра.

3.2. Завантажувач операційної системи для UEFI

Через неможливість напряду завантажувати бінарні файли формату ELF, яким є ядро системи, у оточенні UEFI, було додатково розроблено завантажувач, що надає таку можливість. Окрім самого завантажувача, було розроблено окремий протокол завантаження, за яким після його відпрацювання ядро може отримувати необхідну інформацію про систему із ранніх етапів роботи: регіони вільної фізичної пам'яті, дані про відеобуфер тощо. В цьому розділі буде розглянуто побудову цього протоколу та завантажувача на його основі.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						64
Зм.	Арк.	№ докум.	Підп.	Дата		

3.2.1. Протокол для завантажувача UEFI

Як вже було зазначено, ядру під час ініціалізації необхідно мати інформацію, отримати яку можна лише через переривання BIOS або сервіси UEFI, що є неможливим на етапах його виконання. Саме тому потрібно було надати проміжний протокол, що дозволив би передавати таку інформацію між різними етапами роботи системи. Далі буде наведено структуру даних, що використовується для двох цілей: інформування завантажувача про необхідні дані та налаштування і передача цих даних ядру.

Таблиця 3.8
Протокол завантажувача UEFI

Поле	Джерело	Призначення
Заголовок структури		
kernel_magic	Ядро	“Магічне число” для ідентифікації коректного образу ядра: 00A197A9B007B007 ₁₆
loader_magic	Завантажувач	“Магічне число”, що залишає завантажувач для перевірки ядром: 00700B700B9A791A ₁₆
Інформація про графіку		
width	Обидва	Ширина фреймбуфера дисплею
height	Обидва	Висота фреймбуфера дисплею
format	Обидва	Формат елементу зображення дисплею, найчастіше RGB32 [12]
framebuffer	Завантажувач	Фізична адреса пам’яті фреймбуфера
pitch	Завантажувач	Зміщення між рядками у фреймбуфері
Інформація про фізичну пам’ять		
address	Обидва	Фізична адреса структури з описом вільних регіонів фізичної пам’яті
size	Обидва	Загальний розмір цієї структури
entsize	Завантажувач	Розмір одного елементу списку вільних регіонів

Таблиця 3.8 (Завершення)

Інформація про секції образу ядра		
symtab_hdr	Завантажувач	Фізична адреса опису секції з символами образу ядра
symtab_data	Завантажувач	Фізична адреса даних секції символів
strtab_hdr	Завантажувач	Фізична адреса опису секції із таблицею рядків образу ядра
strtab_data	Завантажувач	Фізична адреса даних секції таблиці рядків
Загальна інформація		
flags	Ядро	Флаги для завантажувача
initrd_base	Завантажувач	Фізична адреса RAM-диску
initrd_size	Завантажувач	Розмір RAM-диску у байтах
rsdp	Завантажувач	Фізична адреса структури ACPI RSDP [13]
cmdline	Завантажувач	Текстові параметри для ядра

Деякі поля цієї структури можуть бути заповнені як ядром, так і завантажувачем:

- Інформація про розмір та формат фреймбуферу буде заповнена завантажувачем тоді, коли ядро встановить всі поля цієї структури у 0, тим самим дозволяючи завантажувачу самому підібрати кращий формат графіки. Якщо ж ці поля будуть явно вказувати на роздільну здатність та формат фреймбуферу, завантажувач буде намагатись обирати саме ці параметри;
- Якщо ядро встановить значення адреси буферу для опису вільних регіонів фізичної пам'яті у 0, то завантажувач сам знайде вільний регіон пам'яті, де його розмістить. Якщо ж значення адреси й розміру явно вказані ядром, завантажувач розмістить ці структури саме за вказаною адресою.

Цієї інформації достатньо, аби надати ядру можливість незалежно продовжувати ініціалізацію та роботу на наступних етапах завантаження.

3.2.2. Програма завантажувача для UEFI

Для зазначеного вище протоколу було розроблено програму, що використовує програмний інтерфейс UEFI для завантаження ядра та виконання інших необхідних налаштувань.

Кожна програма для оточення UEFI являє собою бінарний файл формату PE та використовує інтерфейси, що зазначені у відповідній специфікації від Microsoft [12].

Програма завантажувача під час роботи завантажує один або два файли в залежності від флагів у структурі протоколу:

1. Ядро операційної системи: \kernel.elf;
2. Початковий RAM-диск: \initrd.img.

Окрім завантаження цих файлів у пам'ять, програма також виконує налаштування режиму роботи дисплею, відображення пам'яті та інші параметри системи.

Загальний алгоритм роботи програми завантажувача виглядає наступним чином:

1. Отримання вказівників на підсистеми програмних інтерфейсів UEFI;
2. Отримання інформації про регіони фізичної пам'яті системи;
3. Отримання вказівника на структуру даних ACPI RSDP;
4. Завантаження образу ядра у пам'ять та аналіз параметрів, що запитуються у структурі протоколу;
5. Якщо ядро запитує завантаження RAM-диску, виконується завантаження файлу \initrd.img до пам'яті;
6. Налаштування режиму графіки;
7. Завершення роботи сервісів завантаження UEFI [12];

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						67
Зм.	Арк.	№ докум.	Підп.	Дата		

8. Якщо ядро запитує налаштування віртуального відображення пам'яті до верхніх адрес, як описано у архітектурі системи, виконати його;
9. Налаштувати стек;
10. Передати керування системою завантаженому ядру системи.

Після етапу 7 використання більшої частини програмних інтерфейсів UEFI є неможливим: стає неможливим отримання інформації про фізичну пам'ять, налаштування дисплею тощо. За цих причин завантажувач є особливою частиною проєкту, що була розроблена, окрім власне завантаження ядра, спеціально для виконання задач, неможливих на етапах роботи ядра.

3.3. Стандартна бібліотека мови C

Для надання зручного інтерфейсу для розробки додаків для операційної системи було розроблено стандартну бібліотеку мови програмування C, що реалізує в необхідній мірі відповідну специфікацію у стандарті.

Підсистеми, на які поділяється ця бібліотека, описано у Розділі 2.

Результатом компіляції вихідного коду бібліотеки є статична бібліотека формату *ar* — архів із об'єктними файлами.

3.3.1. Ініціалізація програми та бібліотеки

Реалізація стандартної бібліотеки мови C містить у собі точку входу у програму: символ `_start`, до якого передається керування під час запуску програми, наприклад, за допомогою сімейства функцій `execve(2)`. Спеціальна точка входу дає можливість ініціалізувати та налаштувати компоненти бібліотеки перед тим, як передати керування користувачькому коду.

Ініціалізація складається з наступних етапів:

1. Отримання аргументів та параметрів оточення програми від ядра системи;
2. Налаштування обробників сигналів;

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						68
Зм.	Арк.	№ докум.	Підп.	Дата		

3. Виклик масиву ініціалізаторів — функцій, що є поміченими спеціальним атрибутом `__attribute__((constructor))` [14];
4. Ініціалізація алгоритмів керування пам'яттю та `malloc`, `free`.

Після виконання цих етапів, код бібліотеки викликає функцію `main`, де вже знаходиться користувацький код.

3.3.2. Робота з системними викликами

Для спрощення та абстрагування деталей реалізації конкретної операційної системи, стандартна бібліотека мови C реалізує набір обгорток для функцій, що відповідають певним системним викликам у ядрі.

Вихідний код ядра має заголовочний файл, де описані всі номери системних викликів, аби використовувати їх під час компіляції бібліотеки C та користувацького коду.

Для спрощення системних викликів та реалізації коректного розташування аргументів у регістрах, в коді стандартної бібліотеки надається набір функцій для різних кількостей аргументів. Так, наприклад, функція для виконання системного виклику із трьома параметрами виглядає наступним чином:

```
static inline long __syscall3(long n, long a1, long a2, long a3) {
    unsigned long ret;
    asm volatile ("syscall"
                  : "=a"(ret) : "a"(n), "D"(a1), "S"(a2), "d"(a3)
                  : "rcx", "r11", "memory");
    return ret;
}
```

Де `n` — номер системного виклику; `a1`, `a2` та `a3` — параметри, що передаються його обробнику.

3.3.3. Робота з сигналами

Реалізація обробки сигналів дещо відрізняється від звичайних підходів, що використовуються у системах на базі Linux та операційних системах

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						69
Зм.	Арк.	№ докум.	Підп.	Дата		

сімейства BSD. Замість зберігання таблиці з адресами обробників сигналів у структурах даних ядра, воно має лише одну адресу централізованого обробника сигналів, а самі адреси користувацьких обробників зберігаються у таблиці у пам'яті програми.

Обробка сигналів є детермінованою з точки зору потоків — всі сигнали завжди будуть оброблятися на основному потоці процесу. Цей підхід, хоча й є дещо повільнішим за надані у інших системах, гарантує безпеку та визначеність при обробці сигналів та виключає ряд потенціальних проблем у реалізації багатопоточних програм.

Точкою входу в центральний обробник сигналу є функція `__libc_signal_handle`, що визначає відповідний обробник із таблиці та викликає його, після чого виходить із контексту обробника сигналу шляхом системного виклика про завершення обробки сигналу. Якщо користувацький код не встановлює обробник для сигналу, центральна функція замість нього виконує стандартну процедуру обробки сигналу — вивід інформації про номер та назву сигналу та завершення програми із кодом помилки.

Окрім таблиці з обробниками сигналів, кожна програма має окремий стек, на який ядро перемикає контекст потоку, коли переводить його у обробник сигналу, аби запобігти проблемам із переповненням або невірними зсувами на стеці.

3.3.4. Робота з динамічною пам'яттю

Для роботи із динамічними блоками пам'яті, стандартна бібліотека реалізує звичні функції з сімейства `malloc` та `free`.

Виділення та відслідкування динамічної пам'яті працює наступним чином: вводиться поняття “зони” та “блоку” пам'яті. Блок — один регіон пам'яті, що використовується програмою або є вільним. Зона — група з таких регіонів. Розмір зон завжди є кратним розміру сторінки у операційній системі, тобто 4096 байтам.

Алгоритм вводять три класи зон за розмірами їх блоків:

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						70
Зм.	Арк.	№ докум.	Підп.	Дата		

1. Маленькі зони — блоки до 256 байт, розмір зони 16Кб;
2. Середні зони — блоки до 2Кб, розмір зони 64Кб;
3. Великі зони — блоки до 8Кб, розмір зони 128Кб.

Кожна зона є лінійним регіоном пам'яті, що бібліотека запитує у ядра за допомогою системного виклику `mmap()`, та представляється зв'язною структурою даних, що розміщується у початку цього регіону. Ця структура зв'язує зони між собою та носить інформацію про розмір зони.

Під час створення зони, вона поділяється на блоки із заданим максимальним розміром. Структури даних розміщуються на початках цих блоків та так само зв'язуються між собою у список.

Якщо в зоні якогось розміру закінчуються вільні блоки, виділяється нова зона та алгоритм продовжує виконання.

Якщо в якійсь зоні звільнюється останній зайнятий блок, відповідний регіон пам'яті повертається ядру.

Якщо користувацький код запитує виділення блоку, більшого за розміром за максимальний ліміт для блоків з “великої” зони блоків — цей розмір округлюється вверх до розміру сторінки пам'яті та виділяється напряму за допомогою `mmap()`.

3.4. Програми користувацького оточення

Для надання можливості використання операційної системи користувачем було розроблено набір програм та утиліт, описаних у розділі 2. Додатково було розроблено програму для реалізації інтерфейсу командного рядку у системі.

3.4.1. Інтерфейс командного рядку

Програма командного рядку (`/bin/sh`) надає користувачам можливість вводити команди та отримувати вивід із інших програм у форматі, що є сумісним із стандартом POSIX.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						71
Зм.	Арк.	№ докум.	Підп.	Дата		

Кожна команда складається з двох частин, що розділяються пробілом:

1. Назва команди
2. Параметри команди

Параметри розділяються пробілом, але цього можна уникнути, якщо поставити навколо них лапки.

У одному рядку може бути розміщено декілька команд за допомогою розділення їх різними операторами, наприклад, оператором конвеєра: '|'.
Також присутня можливість вводити коментарі у рядки з командами за допомогою символу '#'.
Далі наведено приклад того, як програма інтерфейсу командного рядку розбиває вхідний рядок з командами на внутрішнє представлення:

```
ls -lh | grep -n "abc def" | less # коментар
```

Перетворюється на наступне представлення:

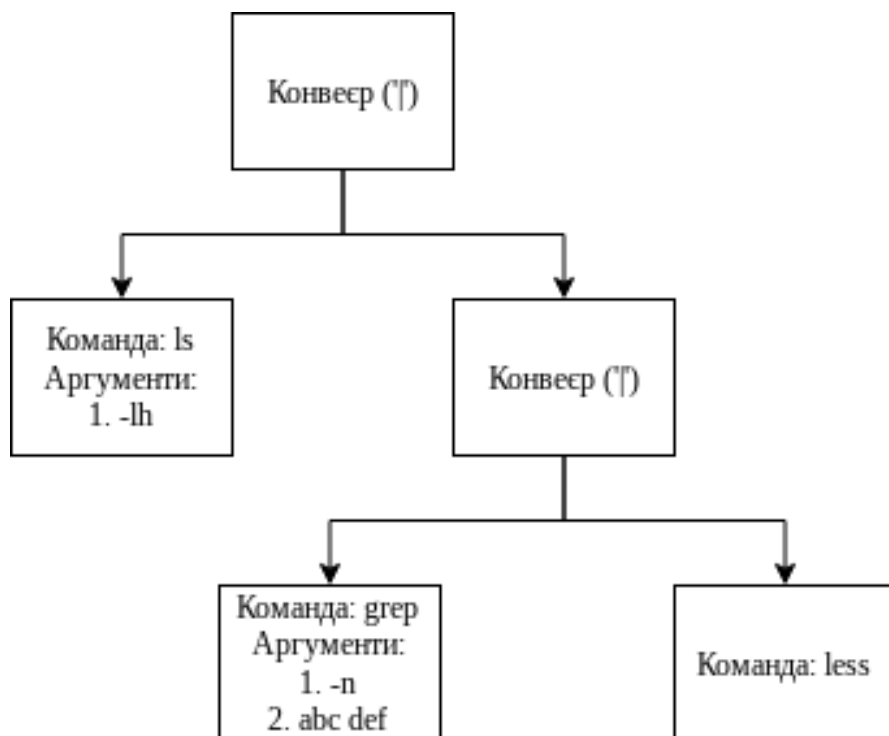


Рис. 3.4 Розбиття команд на внутрішнє представлення

Як видно із рис. 3.4, коментарі не мають внутрішнього представлення, бо всі символи рядку після символу коментаря ігноруються парсером команд.

Окрім конвеєрів, програма інтерфейсу командного рядку також дозволяє вести контроль активних задач за допомогою відповідних сигналів Unix: SIGSTOP та SIGCONT. Перший з них дозволяє за допомогою вводу із клавіатури поставити задачу на “паузу” та повернути керування терміналом програмі інтерфейса командного рядку, а друга надсилається цією програмою, коли користувач вводить команду “fg”, аби продовжити виконання призупиненої задачі та повернути їй термінал.

3.5. Демонстрація роботи результуючої системи

Для демонстрації роботи операційної системи та її компонентів було скомпільовано всі необхідні компоненти: ядро, бібліотека мови C та програми оточення, після чого останні було поміщено у архів формату tar, що далі використовується як RAM-диск для системи. Було зібрано два варіанти операційної системи:

- Образ для завантаження на Legacy BIOS-системах;
- Образ для завантаження на системах з підтримкою UEFI.

Додатково для перевірки роботи відповідних драйверів було створено тестовий образ диску, у якому знаходиться розділ з файловою системою ext2, а для перевірки роботи системи з числами із плаваючою крапкою та графічною підсистемою було створено демонстраційну програму, що візуалізує множини Жуліа за допомогою векторних розширень AVX та SSE.

Для тестування та демонстрації роботи системи було використано два середовища:

1. Програма для віртуалізації qemu;
2. Ноутбук Lenovo ThinkPad T430.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						73
Зм.	Арк.	№ докум.	Підп.	Дата		

Далі буде наведено знімки екрану з програми для віртуалізації qemu. Для доступу до образу з розділом ext2 до віртуальної машини було цей образ було під'єднано через вбудований у qemu емулятор хост-контролеру SATA AHCI.

```
[00000000 kernel_early_init] yggdrasil 2272bc5
[00000000 kernel_early_init] Boot time: 2021-05-26 13:13:11
[00000000 sched_set_ncpus] Setting ncpus to 1
[00000000 usb_daemon_start] Not starting USB daemon - no HCs
[00000000 sched_enter] cpu0 entering sched
/dev/acpi: No such file or directory
Mounting /dev
Mounting /sys
login: root
password:
root@nyan / # date
Tue. May 25 13:13:13 2021
root@nyan / # uname
yggdrasil nyan 2272bc5 x86_64
root@nyan / # ls /sys
kernel
mem
pci
proc
root@nyan / # ps
PID      UID:GID   NAME      PARENT
#14      0:0       ps         10
#10      0:0       sh         9
#9       0:0       login      1
#1       0:0       init       1
root@nyan / #
```

Рис. 3.5 Приклад типової користувацької сесії в системі

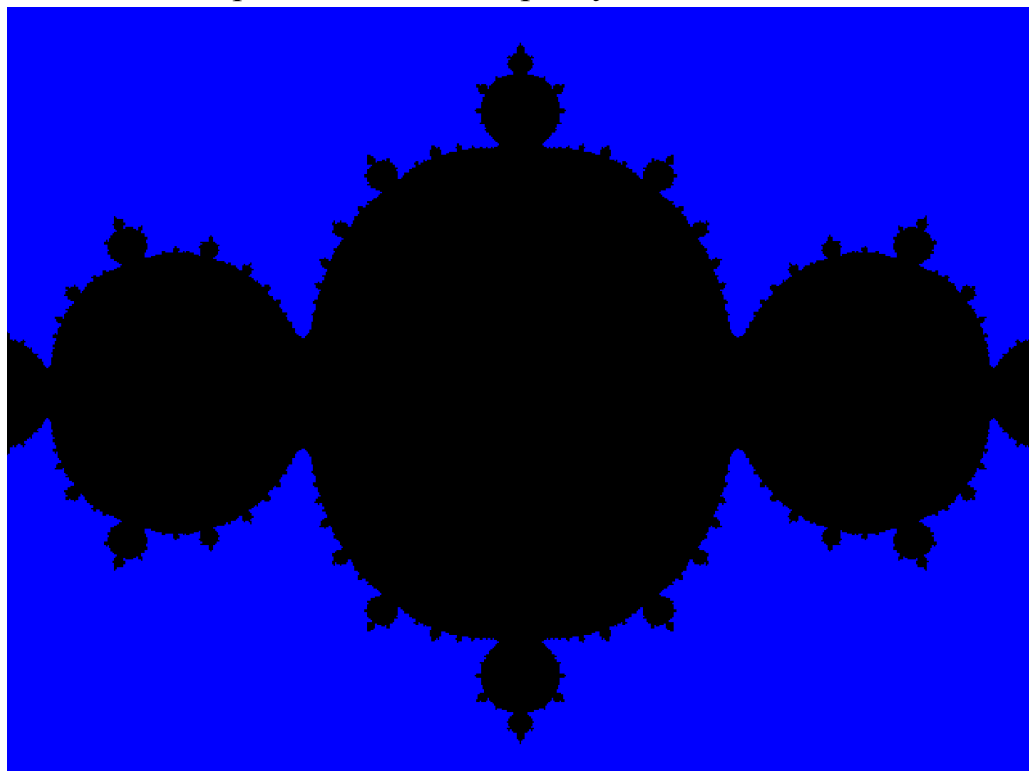


Рис. 3.6 Демонстраційна програма — зображення множини Жуліа

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						74
Зм.	Арк.	№ докум.	Підп.	Дата		

Протестовано також можливість запуску цієї системи на фізичному апаратному забезпеченні — ноутбучі Lenovo ThinkPad T430 із процесором Intel Core i3-3320M та 8 Гб оперативної пам'яті:

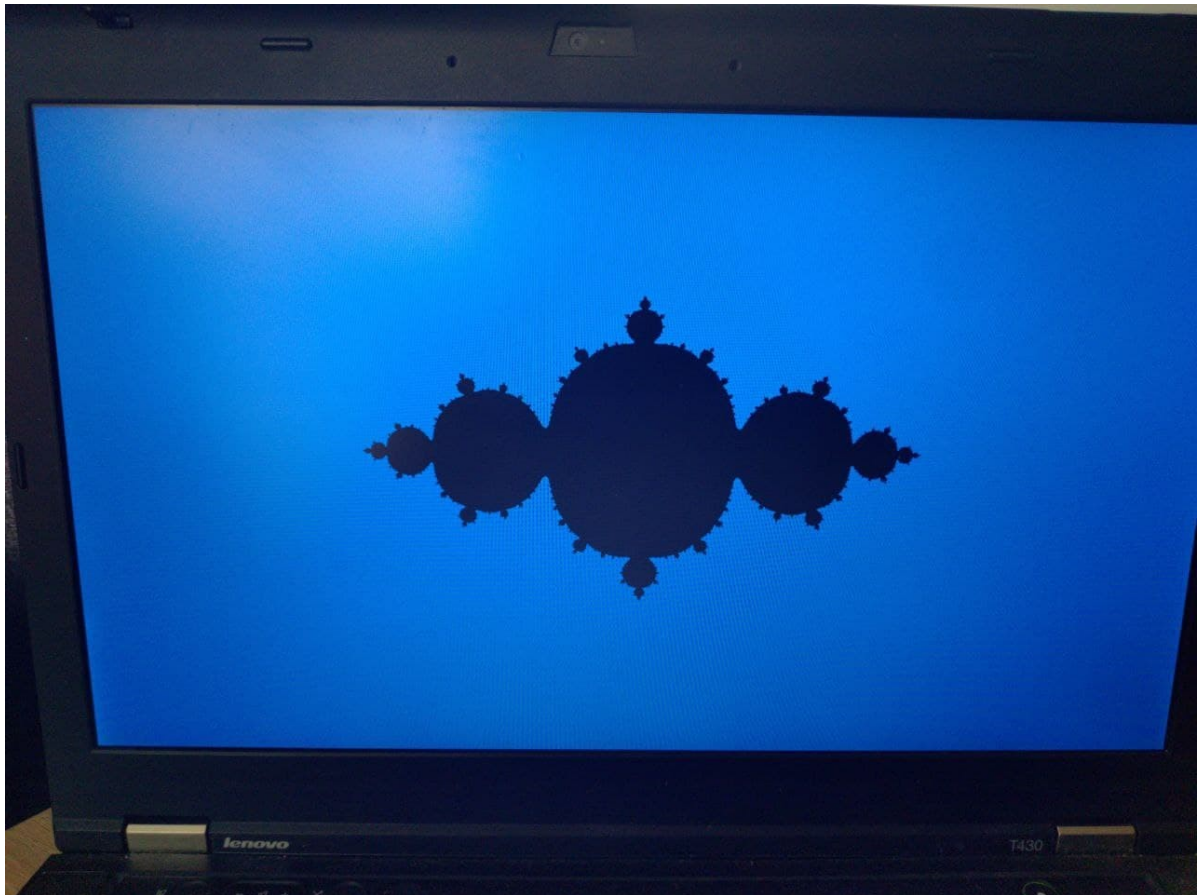


Рис. 3.7 Робота системи на реальному апаратному забезпеченні

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						75
Зм.	Арк.	№ докум.	Підп.	Дата		

ВИСНОВКИ ДО РОЗДІЛУ 3

Даний розділ аналізує та демонструє побудову ядра, протоколів завантажування, стандартної бібліотеки мови С та програмного оточення операційної системи. В ньому було показано основні підсистеми ядра та алгоритм, за яким воно працює.

Було розглянуто реалізацію драйверів фізичних пристроїв, віртуальних, тимчасових та резидентних файлових систем, а також необхідні для них структури даних.

Додатково було розглянуто імплементацію стандартної бібліотеки мови програмування С та програмного оточення розробленої операційної системи, що її використовує.

Виходячи із вимог, що були поставлені до проєкта на початку цієї роботи, розроблений набір програмних компонентів в достатній та необхідній мірі реалізує мету роботи, працює коректно та виконує поставлені задачі.

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						76
Зм.	Арк.	№ докум.	Підп.	Дата		

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Single Unix Specification Version 4, IEEE Std. 1003.1 [Електронний ресурс] // IEEE— 2019. — Режим доступу до ресурсу:
https://standards.ieee.org/project/1003_1.html
2. Unikernels: the rise of the virtual library operating system [Електронний ресурс] / Anil Madhavapeddy, David J. Scott. Commun // ACM 57, 1 (January 2014), 61–69 — 2014. — Режим доступу до ресурсу:
<http://unikernel.org/files/2014-cacm-unikernels.pdf>
3. Дж. Корбет, А. Рубіні, Г. Кроа-Хартман. Драйвери пристроїв для Linux = Linux Device Drivers: Where the Kernel Meets the Hardware. — 3 вид. — O'Reilly, 2005. — ISBN 978-0-5960-0590-0
4. AMD Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions [Електронний ресурс] / Advanced Micro Devices — 2017. — Режим доступу до ресурсу:
<https://www.amd.com/system/files/TechDocs/24594.pdf>
5. The Virtual Filesystem Interface in 4.4BSD [Електронний ресурс] / Marshall K. McKusick // The USENIX Association, Computing Systems — 1995. — Режим доступу до ресурсу:
https://www.usenix.org/legacy/publications/compsystems/1995/win_mckusick.pdf
6. Vnodes: An Architecture for Multiple File System Types in Sun UNIX [Електронний ресурс] / S.R. Kleiman // Sun Microsystems — 1986. — Режим доступу до ресурсу:
<https://web.archive.org/web/20140629172905/http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf>
7. PCI Express Base Specification Revision 5.0, Version 1.0 [Електронний ресурс] // PCI-SIG — 2019. — Режим доступу до ресурсу:
<https://members.pcisig.com/wg/PCI-SIG/document/13005>

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						77
Зм.	Арк.	№ докум.	Підп.	Дата		

8. IBM PC-Compatible CGA Video Reference [Електронний ресурс] / Frank Durda IV — 2001. — Режим доступу до ресурсу:
<https://web.archive.org/web/20051024001342/http://nemesis.lonestar.org/reference/video/cga.html>

9. Дизайн та імплементація розширеної файлової системи другої версії = Design and Implementation of the Second Extended Filesystem — Гронінгенський університет, 1995. — ISBN 90-367-0385-9

10. System V Application Binary Interface x86-64™ Architecture Processor Supplement Draft Version 0.21 [Електронний ресурс] / Jan Hubicka, Andreas Jaeger, Mark Mitchell — 2002. — Режим доступу до ресурсу:
https://refspecs.linuxbase.org/elf/x86_64-abi-0.21.pdf

11. FreeBSD Developers' Handbook [Електронний ресурс] // The FreeBSD Documentation Project — 1995. — Режим доступу до ресурсу:
<https://docs.freebsd.org/en/books/developers-handbook/index.html>

12. UEFI Specification Version 2.9 [Електронний ресурс] // Unified Extensible Firmware Interface Forum — 2021. — Режим доступу до ресурсу:
https://uefi.org/sites/default/files/resources/UEFI_Spec_2_9_2021_03_18.pdf

13. ACPI Specification Version 6.4 [Електронний ресурс] // Unified Extensible Firmware Interface Forum — 2021. — Режим доступу до ресурсу:
https://uefi.org/sites/default/files/resources/ACPI_Spec_6_4_Jan22.pdf

14. Common Function Attributes (Using the GNU Compiler Collection (GCC)) [Електронний ресурс] // GNU — 2020. — Режим доступу до ресурсу:
<https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/Common-Function-Attributes.html>

					<i>ІАЛЦ.467200.003 ПЗ</i>	Арк.
						78
Зм.	Арк.	№ докум.	Підп.	Дата		

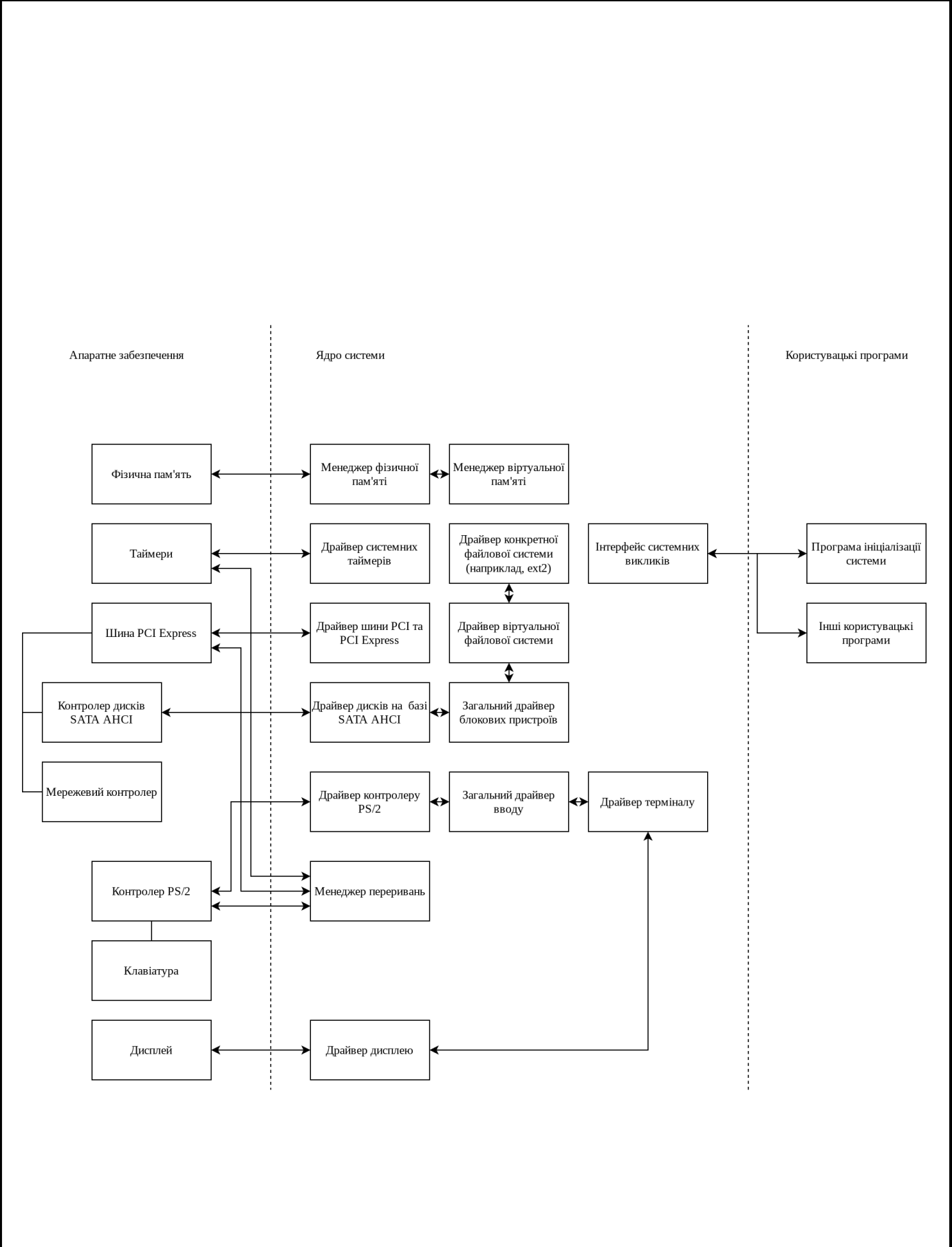
ДОДАТОК А

КОМПОНЕНТИ LINUX-ПОДІБНОЇ ОПЕРАЦІЙНОЇ
СИСТЕМИ

СТРУКТУРНА СХЕМА ВЗАЄМОДІЇ КОМПОНЕНТІВ

Аркушів 1

Київ – 2021



					ІАЛЦ.467200.004 Д1					
Зм.	Арк.	№ докум	Підп.	Дата						
Розробив		Поляков М.С.			Компоненти Linux-подібної операційної системи	Літ.		Аркуш	Аркушів	
Перевірів		Порєв В.М.					Т		І	
						НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71				
Н.контр.		Сімоненко В.П.								
Затв.		Порєв В.М.								

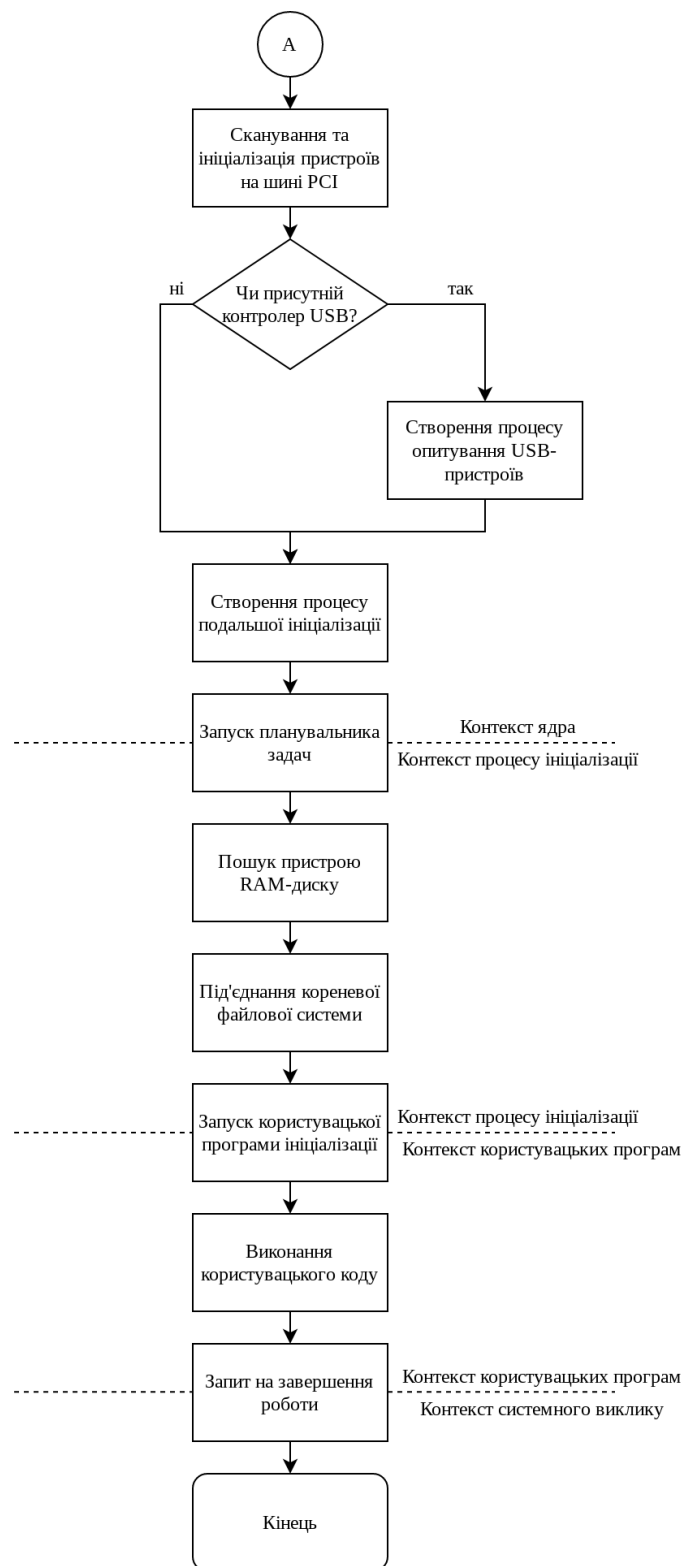
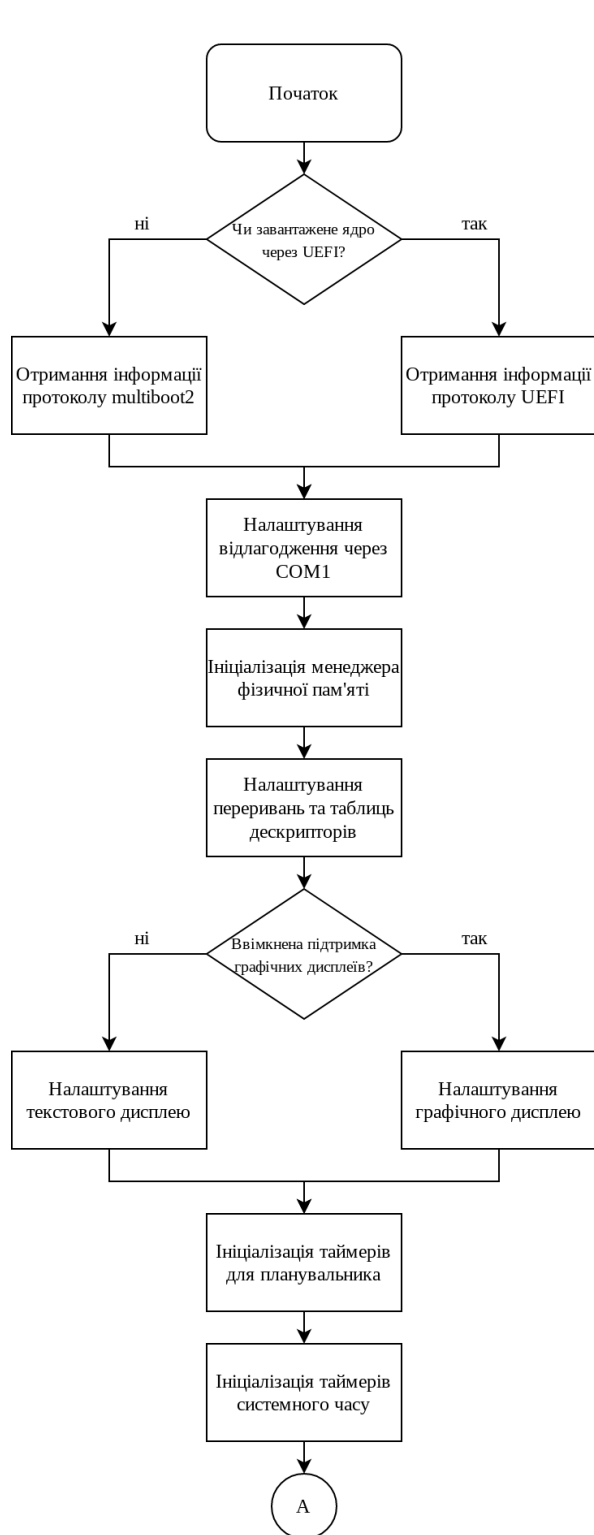
ДОДАТОК Б

КОМПОНЕНТИ LINUX-ПОДІБНОЇ ОПЕРАЦІЙНОЇ
СИСТЕМИ

ПРИНЦИПОВА СХЕМА РОБОТИ СИСТЕМИ

Аркушів 1

Київ – 2021



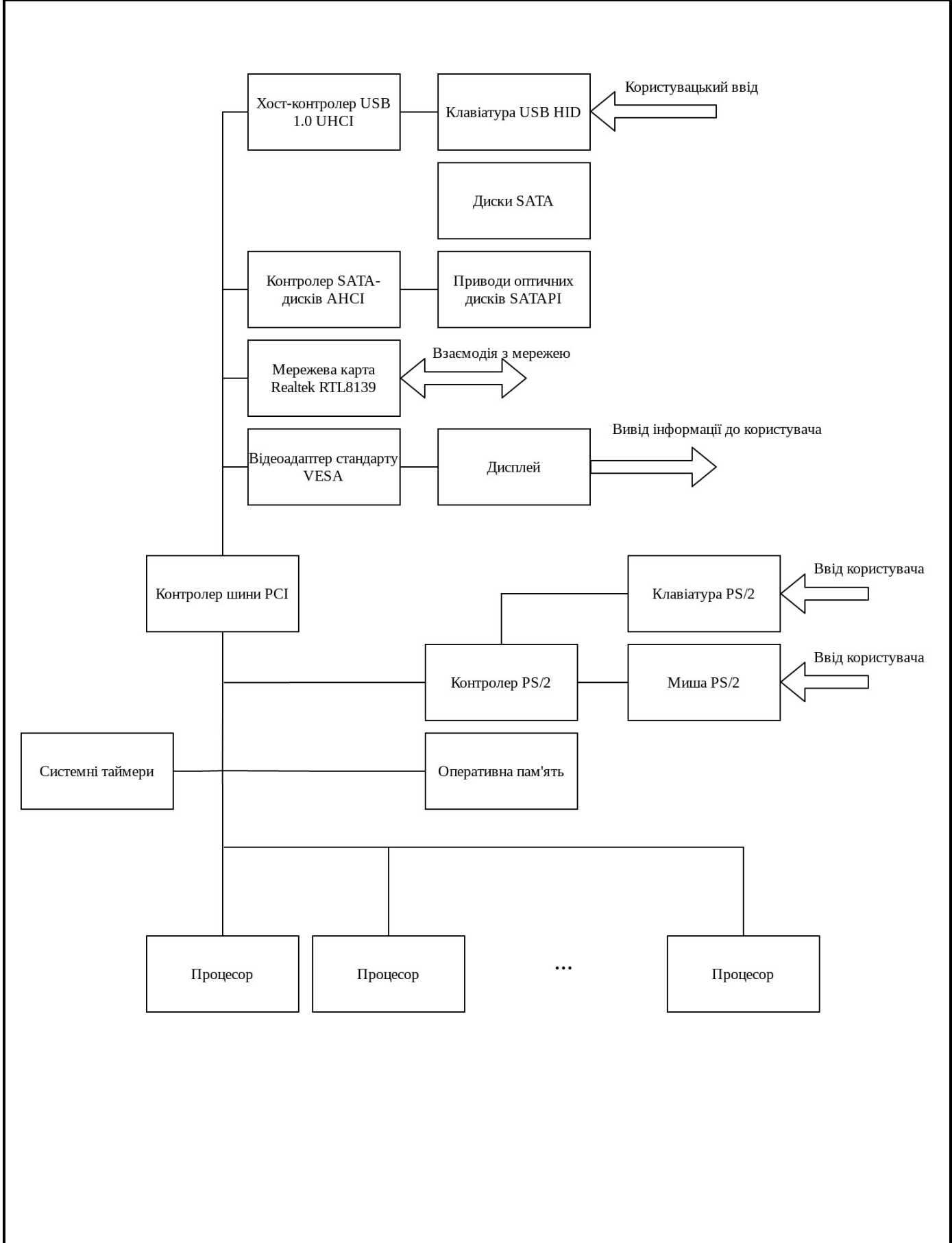
					ІАЛЦ.467200.005 Д2				
Зм.	Арк.	№ докум	Підп.	Дата	Компоненти Linux-подібної операційної системи	Літ.		Аркуш	Аркушів
Розробив	Поляков М.С.					Т		І	
Перевірів	Порєв В.М.					НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ			
						Група ІВ-71			
Н.контр.	Сімоненко В.П.								
Затв.	Порєв В.М.								

ДОДАТОК В

КОМПОНЕНТИ LINUX-ПОДІБНОЇ ОПЕРАЦІЙНОЇ
СИСТЕМИ

**ФУНКЦІОНАЛЬНА СХЕМА АПАРАТНОЇ
ПЛАТФОРМИ ЦІЛЬОВОЇ СИСТЕМИ**

Аркушів 1



					<i>ІАЛЦ.467200.006 ДЗ</i>			
Зм.	Арк.	№ докум	Підп.	Дата	Компоненти Linux-подібної операційної системи	Літ.	Аркуш	Аркушів
Розробив	Поляков М.С.					Т	І	
Перевірів	Порєв В.М.					НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71		
Н.контр.	Сімоненко В.П.							
Затв.	Порєв В.М.							

ДОДАТОК Г

КОМПОНЕНТИ LINUX-ПОДІБНОЇ ОПЕРАЦІЙНОЇ
СИСТЕМИ

ЛІСТИНГ КОДУ ЯДРА СИСТЕМИ

Аркушів 16

Київ – 2021

```
arch/amd64/boot/yboot.S:

#include <config.h>

#include <yboot2-proto/include/protocol.h>

.section .text

.global _entry_yboot

.type _entry_yboot, %function

_entry_yboot:

    cli

    leaq yboot_data(%rip), %rdi

    // Validate loader signature

    movq 8(%rdi), %rax

    movabsq $YB_LOADER_MAGIC_V1, %rsi

    cmp %rax, %rsi

    jne _bad_loader_magic

    // Setup upper paging

    leaq 1f(%rip), %rbp

    leaq boot_prepare_paging_64(%rip), %rdi

    jmp *%rdi

1:

    movq %rax, %cr3

    // Boot method 1

    mov $1, %rdi

    jmp kernel_pre_entry

_bad_loader_magic:

    cli

    hlt

    jmp _bad_loader_magic
```

					ІАЛІЦ.467200.007 Д4			
Зм.	Арк.	№ докум	Підп.	Дата				
Розробив		Поляков М.С.			Компоненти Linux-подібної операційної системи	Літ.	Аркуш	Аркушів
Перевірів		Порєв В.М.				T	I	
						НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71		
Н.контр.		Сімоненко В.П.						
Затв.		Порєв В.М.						

```

.size _entry_yboot, . - _entry_yboot

arch/amd64/boot/multiboot.S:

#include <config.h>

.section .multiboot

.set MAGIC,      0xE85250D6

.set ARCH,       0x0

.set HDRLEN,     16

.set CHKSUM,     -(MAGIC + ARCH + HDRLEN)


.long MAGIC

.long ARCH

.long HDRLEN

.long (CHKSUM & 0xFFFFFFFF)

#if defined(VESA_ENABLE)

.short 5

.short 0

.long 20

.long VESA_WIDTH

.long VESA_HEIGHT

.long VESA_DEPTH

#endif

.short 0

.long 8


.section .text

.global _entry_multiboot

.type _entry_multiboot, %function

_entry_multiboot:

.code32

cli

```

					ІАЛЦ.467200.007 Д4	Арк.
						2
Зм.	Арк.	№ докум.	Підп.	Дата		

```

    leal (multiboot_registers - 0xFFFFFFFF0000000000), %edi

    movl %eax, 0(%edi)

    movl %ebx, 4(%edi)


    mov %cr4, %eax

    or $(1 << 5) | (1 << 4), %eax

    mov %eax, %cr4


    leal 1f - 0xFFFFFFFF0000000000, %ebp

    leal boot_prepare_paging_32 - 0xFFFFFFFF0000000000, %eax

    jmp *%eax

1:

    movl %edi, %cr3


    mov $0xC0000080, %ecx

    rdmsr

    or $(1 << 8), %eax

    wrmsr


    mov %cr0, %eax

    or $(1 << 31), %eax

    mov %eax, %cr0


    lgdt (gdtr64 - 0xFFFFFFFF0000000000)

    ljmp $0x08, $(.long_reload - 0xFFFFFFFF0000000000)

.long_reload:

    mov $0x10, %ax

    mov %ax, %ds

    mov %ax, %es

    mov %ax, %fs

```

					ІАЛЦ.467200.007 Д4	Арк.
						3
Зм.	Арк.	№ докум.	Підп.	Дата		

```

        mov %ax, %gs

        mov %ax, %ss

.code64

        mov $0, %rdi

        jmp kernel_pre_entry

.size _entry_multiboot, . - _entry_multiboot

.code32

.align 16

gdt64:

        .quad 0

        .quad 0x00209A0000000000

        .quad 0x0000920000000000

gdt_end64:

.align 16

gdtr64:

        .short gdt_end64 - gdt64 - 1

        .long gdt64 - 0xFFFFFFFF0000000000

.section .bss

.global multiboot_registers

multiboot_registers:

        .skip 8

arch/amd64/kernel.c:

#include "arch/amd64/multiboot2.h"

#include "arch/amd64/boot/yboot.h"

#include "arch/amd64/hw/rs232.h"

#include "arch/amd64/hw/vesa.h"

#include "arch/amd64/hw/apic.h"

```

					<i>ІАЛЦ.467200.007 Д4</i>	Арк.
						4
Зм.	Арк.	№ докум.	Підп.	Дата		

```

#include "arch/amd64/smp/smp.h"

#include "arch/amd64/hw/acpi.h"

#include "arch/amd64/mm/phys.h"

#include "arch/amd64/hw/gdt.h"

#include "arch/amd64/hw/con.h"

#include "arch/amd64/hw/idt.h"

#include "arch/amd64/hw/rtc.h"

#include "arch/amd64/hw/ps2.h"

#include "arch/amd64/cpuid.h"

#include "arch/amd64/mm/mm.h"

#include "arch/amd64/fpu.h"

#include "sys/block/ram.h"

#include "sys/mem/phys.h"

#include "sys/console.h"

#include "sys/config.h"

#include "sys/assert.h"

#include "sys/kernel.h"

#include "sys/random.h"

#include "sys/string.h"

#include "sys/debug.h"

#include "sys/panic.h"

#include "sys/syms.h"

#include "sys/attr.h"

#include "sys/elf.h"

#include "sys/mm.h"


extern char _kernel_start, _kernel_end;


static struct elf_sections elf_sections;

static uintptr_t initrd_phys_start;

```

					ІАЛЦ.467200.007 Д4	Арк.
						5
Зм.	Арк.	№ докум.	Підп.	Дата		

```

static size_t initrd_size;

static struct boot_video_info boot_video_info = {0};

// Descriptors for reserved physical memory regions
static struct mm_phys_memory_map      phys_memory_map;
static struct mm_phys_reserved         phys_reserve_initrd;
static struct mm_phys_reserved         phys_reserve_kernel = {
    // TODO: use _kernel_start instead of this
    // I was kinda lazy to add an additional reserved region for
    // multiboot stuff, so I simplified things a bit:
    // multiboot is known (don't know if it's a standard) to place
    // its structures below the kernel, so if I reserve pages below the
    // kernel, nothing should be overwritten
    .begin = 0,
    .end = MM_PHYS(&_kernel_end)
};

static void amd64_make_random_seed(void) {
    random_init(15267 + system_time);
}

static void entry_multiboot(void) {
    extern struct {
        uint32_t eax, ebx;
    } __attribute__((packed)) multiboot_registers;

    uintptr_t multiboot_info_addr;

    struct multiboot_tag_framebuffer *multiboot_tag_framebuffer = NULL;

    struct multiboot_tag_module *multiboot_tag_initrd_module = NULL;

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						6
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

struct multiboot_tag_string *multiboot_tag_cmdline = NULL;

struct multiboot_tag_mmap *multiboot_tag_mmap = NULL;


// Check Multiboot2 signature

if (multiboot_registers.eax != MULTIBOOT2_BOOTLOADER_MAGIC) {

    panic("Invalid bootloader magic\n");

}

multiboot_info_addr = MM_VIRTUALIZE(multiboot_registers.ebx);


// Find all requested tags

uint32_t multiboot_len = *(uint32_t *) multiboot_info_addr;

size_t offset = 8; // Skip 2 fields

while (offset < multiboot_len) {

    struct multiboot_tag *tag = (struct multiboot_tag *)
(multiboot_info_addr + offset);

    if (tag->type == 0) {

        break;

    }

    switch (tag->type) {

case MULTIBOOT_TAG_TYPE_CMDLINE:

    multiboot_tag_cmdline = (struct multiboot_tag_string *) tag;

    break;

case MULTIBOOT_TAG_TYPE_ELF_SECTIONS:

    elf_sections.kind = KSYM_TABLE_MULTIBOOT2;

    elf_sections.tables.multiboot2 = (struct multiboot_tag_elf_sections
*) tag;

    break;

case MULTIBOOT_TAG_TYPE_MMAP:

    multiboot_tag_mmap = (struct multiboot_tag_mmap *) tag;

```

					<i>ІАЛЦ.467200.007 Д4</i>	Арк.
						7
Зм.	Арк.	№ докум.	Підп.	Дата		

```

        break;

    case MULTIBOOT_TAG_TYPE_MODULE:

        multiboot_tag_initrd_module = (struct multiboot_tag_module *) tag;

        break;

    case MULTIBOOT_TAG_TYPE_FRAMEBUFFER:

        multiboot_tag_framebuffer = (struct multiboot_tag_framebuffer *)
tag;

        break;

    default:

        kdebug("tag.type = %u, tag.size = %u\n", tag->type, tag->size);

        break;

    }

    offset += (tag->size + 7) & ~7;

}

if (multiboot_tag_cmdline) {

    // Set kernel command line

    kinfo("Provided command line: \"%s\"\n", multiboot_tag_cmdline-
>string);

    kernel_set_cmdline(multiboot_tag_cmdline->string);

}

if (!multiboot_tag_mmap) {

    panic("Multiboot2 provided no memory map\n");

}

if (multiboot_tag_initrd_module) {

    initrd_phys_start = multiboot_tag_initrd_module->mod_start;

    initrd_size = multiboot_tag_initrd_module->mod_end -

        multiboot_tag_initrd_module->mod_start;

```

					<i>ІАЛЦ.467200.007 Д4</i>	Арк.
						8
Зм.	Арк.	№ докум.	Підп.	Дата		

```

    }

    if (multiboot_tag_framebuffer) {

        struct multiboot_tag_framebuffer_common *info =
&multiboot_tag_framebuffer->common;

        if (info->framebuffer_type == 1) {

            boot_video_info.width = info->framebuffer_width;

            boot_video_info.height = info->framebuffer_height;

            boot_video_info.bpp = info->framebuffer_bpp;

            boot_video_info.pitch = info->framebuffer_pitch;

            boot_video_info.framebuffer_phys = info->framebuffer_addr;

        }

    }

    phys_memory_map.format = MM_PHYS_MMAP_FMT_MULTIBOOT2;

    phys_memory_map.address = multiboot_tag_mmap->entries;

    phys_memory_map.entry_size = multiboot_tag_mmap->entry_size;

    // TODO: I'm not really sure here

    phys_memory_map.entry_count = (multiboot_tag_mmap->size -

        offsetof(struct multiboot_tag_mmap, entries)) / multiboot_tag_mmap-
>entry_size;

}

static void entry_yboot(void) {

    if (yboot_data.rsdp == 0) {

        kwarn("Booted from UEFI and no RSDP was provided, will likely result in
error\n");

    } else {

        amd64_acpi_set_rsdp(MM_VIRTUALIZE(yboot_data.rsdp));

    }

}

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						9
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

kinfo("Provided command line: \"%s\"\n", yboot_data.cmdline);

kernel_set_cmdline(yboot_data.cmdline);


if ((yboot_data.flags & YB_FLAG_VIDEO) && yboot_data.video.framebuffer) {

    boot_video_info.width = yboot_data.video.width;

    boot_video_info.height = yboot_data.video.height;

    boot_video_info.bpp = 32;

    boot_video_info.pitch = yboot_data.video.pitch;

    boot_video_info.framebuffer_phys = yboot_data.video.framebuffer;

}


if (yboot_data.initrd_base) {

    kdebug("INITRD BASE: %p, INITRD SIZE: %S\n", yboot_data.initrd_base,
yboot_data.initrd_size);

    initrd_phys_start = yboot_data.initrd_base;

    initrd_size = yboot_data.initrd_size;

}


phys_memory_map.format = MM_PHYS_MMAP_FMT_YBOOT;

phys_memory_map.address = (void *)
MM_VIRTUALIZE(yboot_data.memory_map.address);

extern char yboot_memory_map_data[];

_assert((void *) MM_VIRTUALIZE(yboot_data.memory_map.address) ==
yboot_memory_map_data);

phys_memory_map.entry_size = yboot_data.memory_map.entsize;

phys_memory_map.entry_count = yboot_data.memory_map.size /
yboot_data.memory_map.entsize;

}


void kernel_early_init(uint64_t entry_method) {

```

					<i>ІАЛЦ.467200.007 Д4</i>	Арк.
						10
Зм.	Арк.	№ докум.	Підп.	Дата		

```

// Allows early output

amd64_console_init();


switch (entry_method) {

case 0:

    entry_multiboot();

    break;

case 1:

    entry_yboot();

    break;

default:

    panic("Unknown boot method: something's broken\n");

    break;

}

cpuid_init();


#ifdef VESA_ENABLE

vesa_early_init(&boot_video_info);

struct display *disp = vesa_get_display();

if (disp) {

    kdebug("Initialize early console!\n");

    console_init_early(disp);

}

#endif


// Reinitialize RS232 properly

rs232_init(RS232_COM1);

ps2_init();


// Before anything is allocated, reserve:

```

					ІАЛЦ.467200.007 Д4	Арк.
						11
Зм.	Арк.	№ докум.	Підп.	Дата		

```

// 1. initrd pages

// 2. multiboot tag pages

mm_phys_reserve("Kernel", &phys_reserve_kernel);


if (initrd_phys_start) {

    phys_reserve_initrd.begin = initrd_phys_start & ~0xFFF;

    phys_reserve_initrd.end = (initrd_phys_start + initrd_size + 0xFFF) &
~0xFFF;

    mm_phys_reserve("Initrd", &phys_reserve_initrd);

}


amd64_phys_memory_map(&phys_memory_map);


amd64_gdt_init();

amd64_idt_init(0);


amd64_mm_init();


#ifdef VESA_ENABLE

    vesa_add_display();

#endif


if (elf_sections.kind != KSYM_TABLE_NONE) {

    ksym_set(&elf_sections);

}


if (rs232_avail & (1 << 0)) {

    rs232_add_tty(0);

}

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						12
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

console_init_default();

ps2_register_device();

amd64_acpi_init();

// Print kernel version now
kinfo("yggdrasil " KERNEL_VERSION_STR "\n");

amd64_apic_init();

rtc_init();

// Setup system time
struct tm t;

rtc_read(&t);

system_boot_time = mktime(&t);

kinfo("Boot time: %04u-%02u-%02u %02u:%02u:%02u\n",

      t.tm_year, t.tm_mon, t.tm_mday,

      t.tm_hour, t.tm_min, t.tm_sec);

if (initrd_phys_start) {

    // Create ram0 block device

    ramblk_init(MM_VIRTUALIZE(initrd_phys_start), initrd_size);

}

amd64_make_random_seed();

amd64_fpu_init();

#ifdef AMD64_SMP

amd64_smp_init();

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						13
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

#endif

}

void kernel_main(uint64_t entry_method) {

    kernel_early_init(entry_method);

    main();

}

```

sys/kernel.c:

```

#include <config.h>

#include "arch/amd64/syscall.h"

#include "drivers/pci/pci.h"

#include "drivers/usb/usb.h"

#include "sys/char/tty.h"

#include "sys/console.h"

#include "sys/display.h"

#include "sys/assert.h"

#include "sys/sched.h"

#include "sys/panic.h"

#include "fs/sysfs.h"

#include "sys/init.h"

#include "net/net.h"

#include "fs/vfs.h"

```

```

void main(void) {

    pci_init();

    vfs_init();

    tty_init();

    display_postinit();

    sysfs_populate();

    syscall_init();
}

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						14
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    sched_init();

    usb_daemon_start();

    user_init_start();

    sched_enter();

    panic("This code should not run\n");
}

sys/init.c:

#include "user/fcntl.h"
#include "user/errno.h"
#include "sys/sys_proc.h"
#include "fs/ofile.h"
#include "sys/thread.h"
#include "sys/assert.h"
#include "fs/vfs.h"
#include "sys/config.h"
#include "sys/debug.h"
#include "sys/sched.h"
#include "sys/heap.h"
#include "sys/dev.h"

struct process user_init = {0};

static void user_init_func(void *arg) {

    kdebug("Starting user init\n");

    struct vfs_ioctx *ioctx = &thread_self->proc->ioctx;

    struct vnode *root_dev, *tty_dev;

    int res;

    if ((res = dev_find(DEV_CLASS_BLOCK, "ram0", &root_dev)) != 0) {

        panic("Fail\n");

    }

    if ((res = vfs_mount(ioctx, "/", root_dev->dev, "ramfs", 0, NULL)) != 0) {

        panic("Fail\n");

    }

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						15
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    }

    struct ofile *fd_stdout = ofile_create();

    struct ofile *fd_stdin = ofile_create();

    if ((res = dev_find(DEV_CLASS_CHAR, "tty0", &tty_dev)) != 0) {

        panic("Fail\n");

    }

    if ((res = vfs_open_vnode(ioctx, fd_stdin, tty_dev, O_RDONLY)) != 0) {

        panic("Fail\n");

    }

    if ((res = vfs_open_vnode(ioctx, fd_stdout, tty_dev, O_WRONLY)) != 0) {

        panic("Fail\n");

    }

    thread_self->proc->fds[0] = ofile_dup(fd_stdin);

    thread_self->proc->fds[1] = ofile_dup(fd_stdout);

    thread_self->proc->fds[2] = ofile_dup(fd_stdout);

    thread_self->proc->ctty = tty_dev;

    const char *argp[] = {

        (const char *) kernel_config[CFG_INIT], NULL

    };

    sys_execve(argp[0], argp, &argp[1]);

    while (1) {

        asm volatile ("hlt");

    }

}

void user_init_start(void) {

    process_init_thread(&user_init, (uintptr_t) user_init_func, NULL, 0);

    sched_queue(process_first_thread(&user_init));

}

```

					<i>ІАЛЦ.467200.007 Д4</i>	<i>Арк.</i>
						16
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

ДОДАТОК Д

КОМПОНЕНТИ LINUX-ПОДІБНОЇ ОПЕРАЦІЙНОЇ
СИСТЕМИ

ЛІСТИНГ КОДУ ПРОГРАМ ОТОЧЕННЯ

Аркушів 14

Київ – 2021

```
libc/crt/crt0.c:

#include <stdint.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <_libc/stdio.h>

#include <_libc/signal.h>


extern int main(int argc, char **argv);

extern char **environ;

extern uintptr_t __libc_orig_argp_base;

extern size_t __libc_orig_argp_count;

extern uintptr_t __init_array_start;

extern uintptr_t __init_array_end;


static void call_constructors(void) {

    size_t init_array_size = ((void *) &__init_array_end - (void *)
&__init_array_start) / sizeof(uintptr_t);

    for (size_t i = 0; i < init_array_size; ++i) {

        uintptr_t entry = (&__init_array_start)[i];

        ((void(*) (void)) entry) ();

    }

}

void _start(void *_arg) {

    uintptr_t arg = (uintptr_t) _arg;

    // arg: argp page number << 24 | 12 bit page count << 12 | 12 bit argc

    char **argp = (char **) ((arg >> 12) & ~0xFFFF);
```

					ІАЛЦ.467200.008 Д5				
Зм.	Арк.	№ докум	Підп.	Дата					
Розробив		Поляков М.С.			Компоненти Linux-подібної операційної системи	Літ.		Аркуш	Аркушів
Перевірів		Порєв В.М.				T		I	
						НТУУ «КПІ ім. Ігоря Сікорського» ФІОТ Група ІВ-71			
Н.контр.		Сімоненко В.П.							
Затв.		Порєв В.М.							

```

    int argc = arg & 0xFFF;

    __libc_orig_argp_count = (arg >> 12) & 0xFFF;

    __libc_orig_argp_base = (uintptr_t) argp;

    environ = &argp[argc + 1];

    __libc_signal_init();

    extern void _init(void);

    _init();

    call_constructors();

    exit(main(argc, argp));
}

```

programs/sbin/login.c:

```

#include <sys/ioctl.h>

#include <sys/wait.h>

#include <termios.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <assert.h>

#include <string.h>

#include <signal.h>

#include <fcntl.h>

#include <stdio.h>

#include <errno.h>

#include <pwd.h>

struct spwd {

    char *sp_namp;

    char *sp_pwdp;

};

static int attempt = 0;

```

					ІАЛЦ.467200.008 Д5	Арк.
						2
Зм.	Арк.	№ докум.	Підп.	Дата		

```

static char line_buf[64];

static ssize_t login_getline(char *buf, size_t lim, int echo) {

    ssize_t len;

    struct termios old_tc;

    struct termios tc;

    if (!echo) {

        if (tcgetattr(STDIN_FILENO, &old_tc)) {

            perror("tcgetattr()");

            return -1;

        }

        memcpy(&tc, &old_tc, sizeof(struct termios));

        tc.c_lflag &= ~(ECHO | ECHOE);

        if (tcsetattr(STDIN_FILENO, TCSANOW, &tc)) {

            perror("tcsetattr()");

            return -1;

        }

    }

    len = read(STDIN_FILENO, buf, lim);

    if (!echo) {

        if (tcsetattr(STDIN_FILENO, TCSANOW, &old_tc)) {

            perror("tcsetattr()");

            return -1;

        }

    }

    if (len <= 0) {

        return -1;

    }

    if (buf[len - 1] == '\\n') {

        buf[len - 1] = 0;

    }

}

```

					<i>ІАЛЦ.467200.008 Д5</i>	Арк.
						3
Зм.	Арк.	№ докум.	Підп.	Дата		

```

        return len;
    }

    static int getsppnam_r(const char *name, struct spwd *sp, char *buf, size_t
    buf_size) {

        FILE *fp = fopen("/etc/shadow", "r");

        if (!fp) {

            return -1;

        }

        while (fgets(buf, buf_size, fp) != NULL) {

            char *p0;

            if (*buf && (p0 = strchr(buf, '\n')) != NULL) {

                *p0 = 0;

            }

            p0 = strchr(buf, ':');

            if (!p0) {

                break;

            }

            char *p1 = strchr(p0 + 1, ':');

            if (!p1) {

                break;

            }

            *p0 = 0;

            *p1 = 0;

            if (strcmp(buf, name)) {

                continue;

            }

            fclose(fp);

            sp->sp_namp = buf;

            sp->sp_pwdp = p0 + 1;

            return 0;

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						4
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    }

    fclose(fp);

    errno = ENOENT;

    return -1;
}

static int loginuid(uid_t uid, gid_t gid, const char *sh) {

    int sh_pid = fork();

    if (sh_pid < 0) {

        return -1;

    }

    if (sh_pid == 0) {

        if (setuid(uid) || setgid(gid)) {

            perror("login");

            _exit(-1);

        }

        setpgid(0, 0);

        int pid = getpid();

        ioctl(STDIN_FILENO, TIOCSPGRP, &pid);

        const char *argp[] = { sh, NULL };

        _exit(execve(sh, (char *const *) argp, environ));

    } else {

        int st;

        waitpid(sh_pid, &st, 0);

        int pgid = getpgid(0);

        ioctl(STDIN_FILENO, TIOCSPGRP, &pgid);

        return 0;

    }

}

static int loginas(const char *name) {

```

					<i>ІАЛЦ.467200.008 Д5</i>	Арк.
						5
Зм.	Арк.	№ докум.	Підп.	Дата		

```

char pwbuf[512];

struct passwd pwd;

struct passwd *res;

const char *shell;


if (getpwnam_r(name, &pwd, pwbuf, sizeof(pwbuf), &res) != 0) {

    perror("getpwnam_r()");

    return -1;

}

shell = res->pw_shell;

if (access(shell, X_OK) != 0) {

    perror(shell);

    printf("Using /bin/sh instead\n");

    shell = "/bin/sh";

}

return loginuid(res->pw_uid, res->pw_gid, shell);

}

int main(int argc, char **argv) {

    if (getuid() != 0) {

        printf("login must be run as root\n");

        return -1;

    }

    if (argc != 2) {

        fprintf(stderr, "usage: login TTY\n");

        return -1;

    }

    if (setsid() < 0) {

        perror("setsid()");

        return -1;

    }

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						6
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

}

if (open(argv[1], O_RDONLY, 0) != STDIN_FILENO) {

    return -1;

}

if (open(argv[1], O_WRONLY, 0) != STDOUT_FILENO) {

    return -1;

}

if (dup(STDOUT_FILENO) != STDERR_FILENO) {

    return -1;

}

signal(SIGINT, signal_handler);

char spnam_buf[128];

struct spwd sp;

while (access("/etc/shadow", R_OK) != 0) {

    perror("/etc/shadow");

    if (loginuid(0, 0, "/bin/sh") != 0) {

        return -1;

    }

}

while (1) {

    if (attempt == 3) {

        printf("\033[2J\033[1;1f");

        fflush(stdout);

        attempt = 0;

    }

    printf("login: ");

    fflush(stdout);

    if (login_getline(line_buf, sizeof(line_buf), 1) < 0) {

        break;

    }

}

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						<i>7</i>
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    if (getspnam_r(line_buf, &sp, spnam_buf, sizeof(spnam_buf)) != 0) {

        perror("getspnam_r()");

        ++attempt;

        continue;

    }

    if (sp.sp_pwdp[0] != 0) {

        printf("password: ");

        fflush(stdout);

        if (login_getline(line_buf, sizeof(line_buf), 0) < 0) {

            ++attempt;

            continue;

        }

    } else {

        line_buf[0] = 0;

    }

    if (!strcmp(line_buf, sp.sp_pwdp) && loginas(sp.sp_namp) == 0) {

        // After returning from shell, go here

        attempt = 3;

        continue;

    } else {

        printf("Login failed\n");

        ++attempt;

    }

}

return -1;

}

```

programs/bin/hexd:

```

#include <string.h>

#include <unistd.h>

#include <ctype.h>

```

					ІАЛЦ.467200.008 Д5	Арк.
						8
Зм.	Арк.	№ докум.	Підп.	Дата		

```

#include <errno.h>

#include <stdio.h>

#define LINE_LENGTH      16

#define OPT_CHARS        (1 << 0)

static void line_print(int flags, size_t off, const char *line, size_t len) {
    uint8_t byte;

    printf("%08zx: ", off);

    for (size_t i = 0; i < LINE_LENGTH; ++i) {
        byte = line[i];

        if (i < len) {
            printf("%02hhx", byte);

        } else {
            printf("  ");
        }

        if (i % 2) {
            printf(" ");
        }
    }

    if (flags & OPT_CHARS) {
        printf("| ");

        for (size_t i = 0; i < len; ++i) {
            byte = line[i];

            if (isprint(byte)) {
                printf("%c", byte);
            } else {
                printf(".");
            }
        }
    }
}

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						9
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    }

}

printf("\n");

}

static int do_file(int flags, const char *pathname) {

    char buf[LINE_LENGTH];

    size_t bread, off;

    FILE *fp;

    if (!strcmp(pathname, "-")) {

        fp = stdin;

    } else {

        fp = fopen(pathname, "rb");

    }

    if (!fp) {

        perror(pathname);

        return -1;

    }

    off = 0;

    while ((bread = fread(buf, 1, sizeof(buf), fp)) > 0) {

        line_print(flags, off, buf, bread);

        off += bread;

    }

    if (fp != stdin) {

        fclose(fp);

    }

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						10
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    return 0;
}

static void usage(char **argv) {
    fprintf(stderr, "usage: %s [-q] [FILE...]", argv[0]);
}

int main(int argc, char **argv) {
    static const char *opts = "q";
    int flags = OPT_CHARS;
    int ch;

    while ((ch = getopt(argc, argv, opts)) != -1) {
        switch (ch) {
            case 'q':
                flags &= ~OPT_CHARS;
                break;
            case '?':
                usage(argv);
                return -1;
        }
    }

    if (optind == argc) {
        do_file(flags, "-");
    } else {
        for (int i = optind; i < argc; ++i) {
            do_file(flags, argv[i]);
        }
    }
}

```

					<i>ІАЛЦ.467200.008 Д5</i>	Арк.
						11
Зм.	Арк.	№ докум.	Підп.	Дата		

```

    }

    return 0;
}

programs/bin/cat:

#include <string.h>

#include <stdio.h>

#define CAT_NUMBER      (1 << 0)

static int do_cat(const char *filename, int flags) {

    FILE *fp;

    char buf[1024];

    //size_t bread;

    int line = 0;

    if (filename) {

        fp = fopen(filename, "r");

        if (!fp) {

            perror(filename);

            return -1;

        }

    } else {

        fp = stdin;

    }

    do {

        if (fgets(buf, sizeof(buf), fp) == NULL) {

            break;

```

					<i>ІАЛЦ.467200.008 Д5</i>	<i>Арк.</i>
						12
<i>Зм.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підп.</i>	<i>Дата</i>		

```

    }

    //bread = strlen(buf);

    if (flags & CAT_NUMBER) {
        printf("% 4d ", ++line);
    }

    fputs(buf, stdout);
} while (1);

if (filename) {
    fclose(fp);
}

return 0;
}

int main(int argc, char **argv) {
    int flags = 0;
    int any_printed = 0;
    int res = 0;

    // TODO: handle '-' argument

    // Parse options
    for (int i = 1; i < argc; ++i) {
        if (argv[i][0] == '-') {
            switch (argv[i][1]) {
                case 'n':
                    flags |= CAT_NUMBER;
                    break;
                default:

```

					ІАЛЦ.467200.008 Д5	Арк.
						13
Зм.	Арк.	№ докум.	Підп.	Дата		

```

        fprintf(stderr, "unknown option: '%c'\n", argv[i][1]);

        return -1;
    }

}

// Print any files from args
for (int i = 1; i < argc; ++i) {
    if (argv[i][0] != '-') {
        any_printed = 1;

        if (do_cat(argv[i], flags) != 0) {
            res = -1;
        }
    }
}

// If no argument specifies a file, print from stdin
if (!any_printed) {
    res = do_cat(NULL, flags);
}

return res;
}

```

					<i>ІАЛЦ.467200.008 Д5</i>	Арк.
						14
Зм.	Арк.	№ докум.	Підп.	Дата		