ПРАКТИЧНА РОБОТА 06

Файлові системи та організація вводу-виводу в сучасних операційних системах

План

1. Структура та операції файлової системи.
2. Реалізація директорій.
3. Методи виділення дискового простору.
4. Керування вільним дисковим простором.
5. Практичні завдання.

1. Структура та операції файлової системи

Для підвищення ефективності вводу-виводу передача даних між основною пам'яттю та носіями відбувається ***блоками***. Кожний блок жорсткого диску має 1 або декілька секторів. Залежно від носія, розмір сектору зазвичай складає 512 або 4096 байтів. NVM-пристрої мають блоки по 4096 байтів, а для передачі даних використовуються аналогічні методи, що й для жорстких дисків.

Файлові системи постачають ефективний та зручний доступ до пристрою зберігання інформації, дозволяючи зберігати, знаходити та отримувати дані. Файлова система має вирішувати дві досить різні проектні задачі: як виглядати для користувача та які алгоритми й структури даних застосовувати для відображення логічної структури файлової системи на фізичні носії даних.

Файлова система загалом має шарувату структуру, кожний рівень якої використовує можливості нижніх рівнів для створення нових можливостей, які будуть використовуватись вищими рівнями (рис. 1). ***Рівень керування вводом-виводом*** складається з драйверів пристроїв та обробників переривань для передачі інформації між основною пам'яттю та дисковою системою. Драйвер пристрою можна розглядати як транслятор високорівневих команд на зразок «отримати блок 123» в низькорівневі, специфічні для апаратного забезпечення інструкції, якими керується апаратний контролер (інтерфейс між пристроєм вводу-виводу та рештою системи). Драйвер пристрою зазвичай записує спеціальні бітові шаблони у спеціальні розташування в пам'яті контролера вводу-виводу, щоб повідомити контролер, з яким device location взаємодіяти та які дії виконувати.

Базовій файловій системі (у Linux – підсистема блочного вводу-виводу) потрібно лише створювати узагальнені команди відповідному драйверу пристрою на зчитування та запис блоків на пристрої зберігання інформації.

Команди формуються на основі адрес логічних блоків. Також на даному рівні відбувається планування запитів на ввід-вивід, керування буферами пам’яті та різними кешами, які містять блоки з файловою системою, папками чи даними. Блок у буфері виділяється до того, як зможе відбутись передача блоку носія інформації. Коли буфер заповнений, менеджер буферу повинен знайти більше пам’яті для буферу або очистити відповідний простір у буфері, щоб дозволити запиту на ввід-вивід завершитись. Кеші використовуються для тримання часто використовуваних метаданих файлової системи, таким чином підвищуючи продуктивність, тому керування їх вмістом критичне для оптимальної роботи файлової системи.

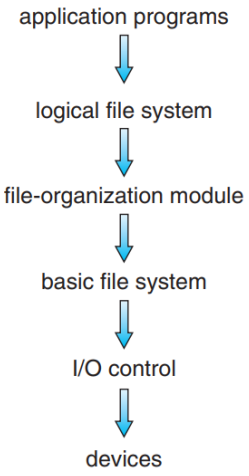

Рис. 1. Шарувата структура файлової системи

Модуль файлової організації знає про файли та їх логічні блоки. Кожний з логічних блоків файлу нумерується від 0 (або 1) до N. Також модуль включає менеджер вільного простору, який відстежує невиділені блоки та постачає їх модулю файлової організації при відповідному зверненні.

Логічна файлова система управляє метаданими, що включають усі структури файлової системи, крім поточних даних (вмісту файлів). Також нею керується структура директорій для постачання потрібної інформації модулю файлової організації, задаючи символьну назву файлу. Також підтримується файлова структура за допомогою ***блоків управління файлами (file-control blocks, FCB – inode у UNIX)***. Такі блоки містять інформацію про файл, включаючи власника, дозволи та розташування вмісту файлу. Також логічна файлова система відповідає за захист файлової системи в цілому.

При шаруватій структурі мінімізується дублювання коду. Код рівнів керування вводом-виводом та інколи базової файлової системи може використовуватись багатьма файловими системами. Кожна файлова система

може потім мати власну логічну файлову систему та модулі файлової організації. На жаль, шарування може внести більше накладних витрат ОС, що може призвести до зниження продуктивності. Використання шарування, включаючи рішення щодо кількості прошарків та їх можливостей – основний виклик при проектуванні нових систем.

Більшість сучасних ОС підтримують більш, ніж одну файлову систему. Наприклад, більшість CD-ROM-ів записують у стандартному форматі ISO 9660. Також разом з файловими системами зйомних носіїв інформації кожна ОС має одну або декілька дискових файлових систем. UNIX використовує UNIX file system (UFS), яка базується на Berkeley Fast File System (FFS). Windows підтримує формати дискових файлових систем FAT, FAT32 та NTFS (Windows NT File System), а також формати файлових систем CD-ROM та DVD. Хоч Linux підтримує понад 130 різних файлових систем, стандартна файлова система Linux відома як розширена (extended) файлова система. Найбільш поширеними є формати ext3 та ext4. Також існують розподілені файлові системи, у яких файлова система на сервері монтується (mount) до клієнтських комп'ютерів у мережі.

Дослідження файлових систем активно продовжуються. Компанія Google створила власну файлову систему для врахування своїх особливих потреб зберігання та отримування даних, що включає високопродуктивний доступ для багатьох клієнтів на дуже великій кількості дисків. Інший цікавий проєкт – файлова система FUSE, яка забезпечує гнучку розробку файлової системи та реалізує й виконує код файлових систем на рівні користувача, а не ядра. Використовуючи FUSE, користувач може додавати нову файлову систему до цілого ряду ОС та використовувати дану файлову систему для керування файлами.

Кілька on-storage та вбудованих (in-memory) структур використовуються для реалізації файлової системи. Ці структури розрізняються залежно від ОС та файлової системи, проте деякі загальні принципи все ж застосовні.

На носії даних файлова система може тримати інформацію про запуск ОС, загальну кількість блоків, кількість та розташування вільних блоків, структуру директорій та окремі файли. Коротко опишемо такі структури:

- ***Блок управління завантаженням (boot control block** – один на том)* може містити інформацію, потрібну для завантаження ОС з відповідного тому. Якщо диск не містить ОС, даний блок може бути порожнім. Зазвичай це перший блок тому (volume). У UFS він називається ***завантажувальним***

блоком (boot block). В NTFS – ***завантажувальним сектором (partition boot sector)***.

- ***Блок управління томом (volume control block*** *– один на том)* містить характеристики тому, такі як кількість блоків у томі, розмір блоків, кількість вільних блоків (free-block count), вказівники на вільні блоки (free-block pointers) та кількість вільних блоків управління файлами (free-FCB count) і вказівники на FCB. У UFS подібний блок називають ***суперблоком***. В NTFS – ***головна таблиця файлів*** ***(master file table)***.
- ***Структура директорій*** *(одна для файлової системи)* використовується для організації файлів. У UFS вона включає назви файлів та відповідні inode numbers. В NTFS вона зберігається в головній таблиці файлів.
- ***FCB на файл*** містить багато інформації про файл, зокрема, й унікальний ідентифікатор для асоціювання з папкою (directory entry). В NTFS дана інформація насправді зберігається в головній таблиці файлів, яка використовує структуру реляційної БД з рядком на файл.

Вбудована (in-memory) інформація використовується як для управління файловою системою, так і для покращення продуктивності за допомогою кешування. Дані завантажуються під час монтування, оновлюються протягом операцій з файловою системою та скасовуються при відключенні (dismount). Може включати кілька типів структур.

- Вбудована таблиця монтувань (***mount table)*** містить інформацію про кожний змонтований том.
- Вбудований кеш структури директорій (directory-structure cache) зберігає інформацію щодо папок, до яких недавно здійснювався доступ (для директорій, в яких відбувається монтування тому, даний кеш може містити вказівник на таблицю тому (volume table)).
- ***Системна таблиця відкритих файлів (system-wide open-file table)*** містить копію FCB кожного відкритого файлу разом з іншою інформацією.
- ***Таблиця відкритих фалів на процес (per-process open-file table)*** зберігає вказівники на відповідні записи в системній таблиці відкритих файлів разом з іншою інформацією для всіх файлів, відкритих процесом.
- Буфери, всередині яких тримаються блоки файлової системи при зчитуванні з / запису в файлову систему.

Для створення нового файлу процес викликає логічну файлову систему, яка знає формат структур директорій. Вона виділяє новий FCB (як альтернатива,

якщо конкретна файлова система створює всі FCB-и під час створення файлової системи, FCB виділяється з набору вільних FCB). Потім система зчитує відповідну папку в пам’ять, оновлює її новими назвою файлу та FCB, а потім записує її назад у файлову систему. Типовий FCB показано на рис. 2.

Дозволи для файлу
Дати для файлу (створення, доступу, запису)
Власник файлу, група, ACL
Розмір файлу
Блоки даних файлу або вказівники на них

Рис. 2. Типова структура блоку управління файлами

Деякі ОС, включаючи UNIX, працюють з папкою так же, як і з файлом, маючи окреме поле “type”, що визначає, чи є сутність папкою. Інші ОС, включаючи Windows, реалізують окремі системні виклики для файлів і папок, розглядаючи їх як окремі сутності. Whatever the larger structural issues, логічна файлова система може викликати модуль файлової організації для відображення directory-вводу-виводу на розташування блоків зберігання даних, які передаються базовій файловій системі та системі керування вводом-виводом.

Тепер після створення файлу його можна використовувати для вводу-виводу. Проте спочатку його потрібно відкрити. Системний виклик open() передає назву файлу в логічну файлову систему. Спочатку відбувається пошук у системній таблиці відкритих файлів (system-wide open-file table), щоб перевірити, чи вже з файлом працює інший процес. Якщо так, створюється per-process запис у таблиці відкритих файлів, який вказує на існуючу системну таблицю відкритих файлів. Цей алгоритм може save substantial overhead. Якщо файл ще не відкритий, у структурі директорій шукається дана назва файлу. Частини структури директорій зазвичай кешуються в пам’ять для прискорення операцій з папками. Як тільки файл знайдено, FCB копіюється в системну таблицю відкритих файлів у пам’яті. Дана таблиця не лише зберігає FCB, але й стежить за кількістю процесів, які мають файл відкритим.

Далі заноситься запис у per-process таблицю відкритих файлів із вказівником на запис у системній таблиці відкритих файлів та деякі інші поля. Ці інші поля можуть включати вказівник на поточне місце в файлі (для наступної операції read() або write()), а також режим доступу, в якому файл відкривається. Виклик open() повертає вказівник на відповідний запис у таблиці файлової системи для кожного процесу. Всі файлові операції потім виконуються за допомогою цього вказівника. Назва файлу може не бути частиною таблиці

відкритих файлів, оскільки система не використовує її, як тільки відповідний FCB знайдено на диску. Проте вона може кешуватись, щоб зберегти час на наступні операції відкривання того ж файлу. Назва запису може бути різна: UNIX-системи називають його ***файловим дескриптором***, а Windows – ***файловим хендлером (file handle)***.

Коли процес закриває файл, запис у таблиці відповідного процесу видаляється, а системний лічильник декрементується. Коли всі користувачі, які відкрили файл, закриють його, усі оновлені метадані копіюються назад у disk-based структуру директорій, а запис у системній таблиці відкритих файлів видаляється.

Не слід упускати аспекти кешування у структурах файлових систем. Більшість систем тримає всю інформацію щодо відкритого файлу, за винятком його блоків даних, у пам'яті. BSD UNIX – типовий приклад використання кешів, wherever дисковий ввід-вивід може зберігатись. Середній відсоток влучань для кешу у 85% показує, що такі підходи варті реалізації.

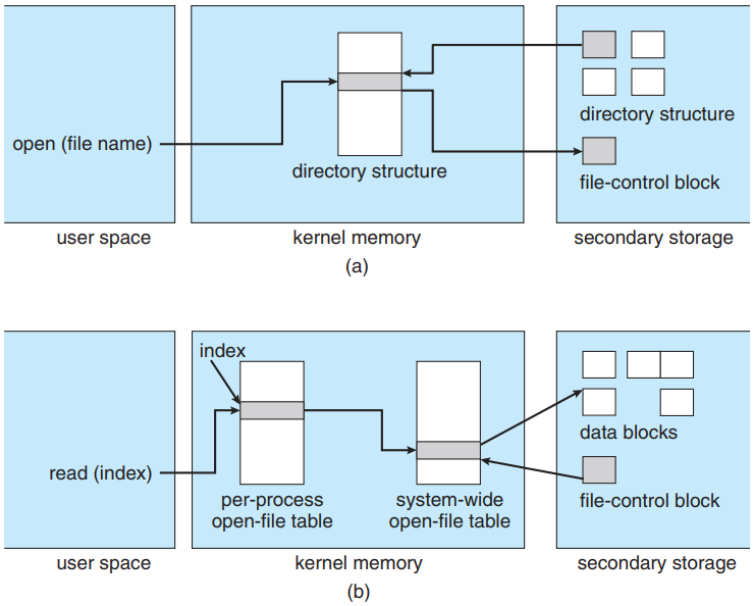

Рис. 3. Структури вбудованої файлової системи. (a) Відкриття файлу, (b) Зчитування файлу.

2. Реалізація директорій

Вибір алгоритмів виділення директорій та управління ними суттєво впливає на ефективність, продуктивність та надійність файлової системи. Найпростіший метод реалізації папки – використання ***лінійного списку*** назв файлів із вказівниками на їх блоки даних. Цей метод легко реалізується, проте повільно виконується. Для створення нового файлу необхідно спочатку знайти

папку, щоб бути впевненим, що ще не існує файлу з такою назвою. Далі додаємо новий запис у кінець директорії. Для видалення файлу шукаємо директорію цього файлу та звільняємо виділений йому простір.

Для повторного використання директорії можна зробити кілька речей. Можемо позначити запис (entry) як «unused» (присвоюючи йому спеціальну назву, наприклад, з усіх пробілів; присвоюючи йому номер невалідного inode (наприклад, 0) або включаючи біт «used–unused» у кожний запис), або прикріпляти його до списку free directory записів.

Третя альтернатива – скопіювати останній запис у директорії у звільнене розташування та зменшити довжину директорії. Зв'язний список теж може використовуватись для зменшення необхідного часу на видалення файлу.

Справжнім недоліком лінійного списку directory-записів є лінійний пошук файлу. Інформація про директорію часто використовується, тому користувачі помітять, якщо доступ до неї буде повільним. Фактично, багато ОС реалізують програмний кеш для збереження інформації щодо нещодавно використаних директорій. Влучання в кеш усуває потребу постійного перечитування інформації з вторинного сховища. Відсортований список дозволяє бінарний пошук та зменшує середній час пошуку. Проте вимога, щоб список підтримувався відсортованим може ускладнити створення та видалення файлів, оскільки може знадобитись переміщення значних обсягів directory-інформації, щоб підтримувати відсортовану директорію. Більш складна деревовидна структура даних, наприклад, збалансоване дерево, може тут допомогти. Перевагою відсортованого списку є те, що відсортований вміст папки можна отримати без окремого кроку сортування.

Інша структура даних, яка використовується для file directory, – це хеш-таблиця, яка отримує значення, обчислене з назви файлу та повертає вказівник на назву файлу в лінійному списку. Тому значно знижується час пошуку по директорії. Вставка та видалення також досить прямолінійні, хоч деяку обробку треба додати для колізій.

Основні складнощі з хеш-таблицею: її загалом фіксований розмір та залежність хеш-функції від цього розміру. Наприклад, створюємо linear-probing хеш-таблицю на 64 записи. Хеш-функція конвертує назви файлів у цілі числа від 0 до 63. Якщо пізніше спробуємо створити 65й файл, необхідно збільшувати хеш-таблицю папки—наприклад, до 128 записів. У результаті потрібна нова хеш-функція, яка відображає назви файлів у діапазон від 0 до 127, тому необхідно

реорганізувати існуючі записи директорії, щоб відобразити нові значення хеш-функції.

У якості альтернативи можемо застосувати вирішення колізій методом ланцюжків. Пошук може дещо сповільнитись, оскільки додається пошук по зв'язному списку. Проте цей метод, скоріше за все, буде набагато швидший за лінійний пошук у всій папці.

3. Методи виділення дискового простору.

У більшості випадків багато файлів зберігаються на одному носії даних, тому основною задачею є спосіб виділення простору для цих файлів так, щоб простір використовувався ефективно, а доступ до файлів був швидкий. Три основні методи виділення простору вторинної пам'яті: ***послідовний (contiguous)***, ***зв'язаний (linked)*** та ***індексований (indexed)***. Хоч деякі системи підтримують всі три методи, частіше зустрічаються системи з використанням одного методу для всіх файлів у межах типу файлової системи.

Послідовне виділення (Contiguous allocation) вимагає, щоб кожний файл займав набір послідовних блоків на пристрої. Адреси визначають лінійне впорядкування даних. Припустимо, що тільки одне завдання (job) отримує доступ до пристрою. Звернення до блоку b+1 після блоку b зазвичай не вимагає переміщення зчитуючої голівки (head). Коли її переміщення потрібне (з останнього сектору одного циліндру на перший сектор наступного циліндру), голівка лише переміщається з однієї доріжки на наступну. Таким чином, для жорстких дисків кількість операцій пошуку (seek) на диску, потрібних для доступу до послідовно виділених файлів мінімальна (припускаємо, що блоки з близькими логічними адресами близькі й фізично), як і тривалість пошуку (seek time), коли такий пошук нарешті буде потрібним.

Послідовне виділення пам'яті під файл визначається адресою першого блоку та довжиною (в блоках) файлу. Якщо файл охоплює *n* блоків та починається з розташування *b*, тоді він займатиме блоки *b, b+1, b+2, ..., b+n-1*. Запис директорії (directory entry) для кожного файлу вказує на адресу початкового блоку та довжину області пам'яті, виділеної для цього файлу (рис. 4). Послідовне виділення просте в реалізації, проте має обмеження, таким чином, не використовуючись у сучасних файлових системах.

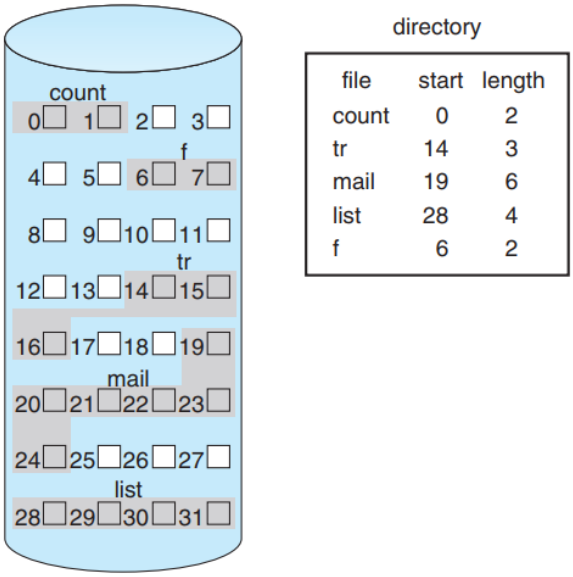

Рис. 4. Послідовне виділення дискового простору

Доступ до файлу, якому послідовно виділено пам’ять, простий. Для послідовного доступу файлова система згадує адресу останнього блоку, на який було посилання, і за необхідності зчитує наступний блок. Для прямого доступу до блоку i файлу, який починається з блоку b, звертаємось до блоку $b+i$. Таким чином, послідовний та прямий доступ може підтримуватись послідовним виділенням пам’яті.

Проте послідовне виділення має деякі проблеми. Одна зі складнощів – знаходження простору для нового файлу. Обрана для управління вільним простором система визначає, як виконати таку задачу. Застосувати можна будь-яку систему управління, проте деякі повільніше працюють за інші.

Проблема послідовного виділення пам’яті може розглядатись як окремий випадок загальної задачі ***динамічного виділення пам’яті (dynamic storage-allocation)***, розглянутої в минулій темі. Вона включає задоволення запиту на простір пам’яті розміром n зі списку вільних блоків. Найбільш поширеними стратегіями виділення є «first-fit» та «best-fit» підходи.

Перелічені алгоритми страждають від проблеми зовнішньої фрагментації. По мірі виділення та звільнення пам’яті простори вільної пам’яті розбиваються на маленькі частини. Це стає проблемою, коли найбільший неперервний шматок недостатньо великий для запиту; сховище фрагментується на багато дірок, кожна недостатнього розміру. Залежно від загального обсягу дискового сховища та середнього розміру файлу, зовнішня фрагментація може бути малою або великою проблемою.

Одна зі стратегій запобігання втрати значних обсягів сховища в зв'язку з зовнішньою фрагментацією – копіювати всю файлову систему на інший пристрій. Початковий носій потім повністю очищається, створюючи єдиний великий неперервний вільний простір. Потім копіюємо файли назад на початковий пристрій, виділяючи безперервний простір з отриманої великої дірки. Дана схема ефективно компактує весь вільний простір, вирішуючи проблему фрагментації.

Ціна такого стиснення – час, і вона може бути особливо високою для великих накопичувачів. Компактування таких пристроїв може тривати годинами, а потребуватись щотижня. Деякі системи вимагають, щоб ця функція виконувалась оффлайн, коли файлова система відмонтована. Протягом цього часу нормальна робота системи зазвичай не може відбуватись, тому подібне компактування уникається. Більшість сучасних ОС, яка потребує дефрагментації, може виконати її протягом нормальної роботи системи, проте втрата продуктивності може бути суттєвою.

Інша проблема послідовного виділення – визначення, скільки місця потрібно для файлу. Коли файл створюється, загальний обсяг необхідного простору потрібно знайти та виділити. Як творець (програма чи людина) дізнається розмір файлу для створення? У деяких випадках це може бути досить просто (копіювання існуючого файлу, наприклад). Проте загалом розмір вихідного файлу може бути важко оцінити.

Якщо виділити надто мало простору для файлу, може виявитись, що розширити файл неможливо. Особливо зі стратегією best-fit, простір з обох сторін файлу може використовуватись. Тому не можемо зробити файл більшим на місці. Існують 2 можливості. Перша – програма може перервати виконання з відповідним повідомленням про помилку. Тоді користувач повинен виділити більше простору та запустити програму знову. Такі повторювані дії можуть бути коштовними. Для їх запобігання користувач зазвичай визначає необхідний обсяг пам'яті з запасом, що призводить до значних даремних витрат місця. Інша можливість – знайти більшу дірку, скопіювати вміст файлу в новий простір та звільнити попередній. Цей ряд дій може повторюватись, поки існуватиме простір, хоч це може бути затратним по часу. Проте користувача ніколи не потрібно явно інформувати про ці дії; система продовжує роботу незалежно від проблеми, хоч і все повільніше.

Навіть якщо загальний обсяг потрібного простору для файлу чудово відомий, попереднє виділення (preallocation) може бути неефективним. Файл, що

повільно розростатиметься протягом тривалого періоду (місяців чи років), повинно виділятись достатньо простору для остаточного розміру файлу, навіть якщо велика частина цього простору буде довго не використовуватись. Таким чином, існує значна внутрішня фрагментація.

Для мінімізації цих недоліків ОС може використовувати модифіковану схему послідовного виділення пам’яті. Тут спочатку виділяється безперервний шматок простору. Потім, якщо цього обсягу недостатньо, додається інший достатньо великий безперервний шматок, який називають ***розширенням (extent)***. Розташування блоків файлу потім записується як розташування + кількість блоків + вказівник на перший блок наступного розширення. На деяких системах власник файлу може задати розмір розширення, проте це налаштування призводить до некоректної роботи, якщо власник некоректно визначає розмір. Внутрішня фрагментація все ще може бути проблемою, якщо розширення надто великі, а зовнішня фрагментація може стати проблемою по мірі зміни розмірів розширень при виділенні та звільненні пам’яті. Комерційна файлова система Symantec Veritas використовує extents для оптимізації продуктивності. Veritas – високопродуктивна заміна стандартної UNIX UFS.

Зв’язне виділення (Linked allocation) вирішує всі проблеми послідовного виділення пам’яті. При зв’язному виділенні кожний файл є зв’язним списком блоків зберігання; блоки можуть бути розпорошеними будь-де на пристрої. Директорія містить вказівник на перший та останній блоки файлу. Наприклад, файл з 5 блоків може починатись у блоці 9 і продовжуватись у блоках 16, 1, 10 і 25 (рис. 5). Кожний наступний блок містить вказівник на наступний блок. Ці вказівники недоступні для користувача, тому якщо кожний блок буде розміром 512 байтів, а адреса блоку (вказівник) – 4 байти, користувач побачить блоки по 508 байтів.

Для створення нового файлу просто створюємо новий запис у директорії. Кожний такий запис має вказівник (ініціалізується null) на перший блок файлу. Розмір поля також встановлюється як 0 (порожній файл). Записування в файл спричиняє знаходження системою керування вільним простором вільного блоку, в який ітиме запис та на який ставиться посилання наприкінці файлу. Для зчитування файлу просто читаються блоки за відповідними посиланнями зв’язного списку. При зв'язному виділенні відсутня зовнішня фрагментація, і кожний вільний блок зі списку вільних блоків може бути використаним для задоволення запиту. Розмір файлу не потрібно оголошувати при створенні файлу.

Файл може рости за умови доступності вільних блоків. Звідси, ніколи не потрібно стискати (компактувати) дисковий простір.

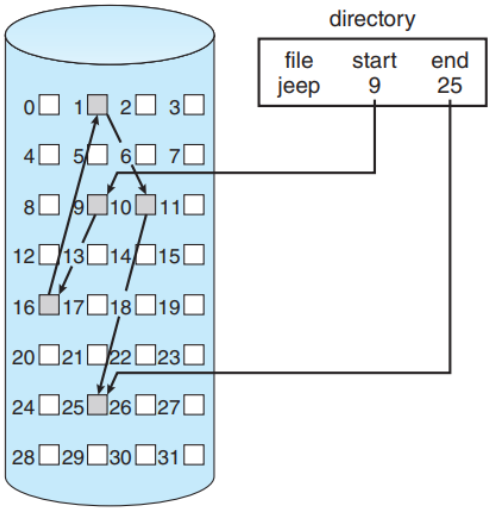

Рис. 5. Зв’язне виділення дискового простору

Основною проблемою зв’язного виділення є його ефективне використання лише для файлів з послідовним доступом. Щоб знайти *i*-тий блок файлу, необхідно стартувати з початку файлу та слідувати за вказівниками, поки не отримається *i*-тий блок. Кожний доступ до вказівника вимагає задіяти операцію зчитування з пристрою, а деякі ще й вимагають пошуку по ЖД. Відповідно, підтримка можливості прямого доступу для зв’язновиділених файлів неефективна.

Ще одним недоліком є необхідна для вказівників пам’ять. Якщо вказівник вимагає 4 байти з 512-байтового блоку, тоді 0.78% диску буде зайняте вказівниками, а не інформацією. Кожний файл вимагатиме трохи більше місця.

Поширене вирішення цієї проблеми – зібрати блоки у кластери та виділяти пам’ять саме кластерам. Наприклад, файлова система може визначити кластер на 4 блоки та працювати із вторинним сховищем тільки за розмірами кластерів. Тоді вказівники займають набагато менше файлового простору. Цей метод дозволяє залишатись простим відображенню логічних блоків у фізичні, покращує пропускну здатність ЖД (оскільки необхідно менше рухів голівок для пошуку) та спрощує керування списком вільних блоків. Плата за такі переваги – зростання внутрішньої фрагментації, оскільки більше простору витрачається, коли кластер частково повний, ніж при частковій заповненості блоку. Також страждає продуктивність довільного вводу-виводу, тому що запит на малий обсяг даних передає великий обсяг. Кластери можуть застосовуватись для покращення часу доступу до диску і для багатьох інших алгоритмів.

Інша проблема зв’язного виділення – надійність. Уявіть, що станеться, коли вказівник з ланцюжка буде втрачений або пошкоджений. Баг у коді ОС або апаратна помилка може призвести до обрання неправильного вказівника. Така помилка, в свою чергу, призводить до зв’язування зі списком вільних блоків або іншим файлом. Часткове вирішення – використовувати двозв’язні списки, ще одне – зберігати назву файлу та relative block number у кожному блоці. Проте ці схеми вимагають ще більше накладних витрат на кожний файл.

Важлива варіація зв’язного розміщення – використання таблиці розміщення файлів ***(file-allocation table, FAT)***. Цей простий, але ефективний метод виділення дискового простору використовувався в ОС MS-DOS. Частина сховища на початку кожного тому віддається під таблицю. У таблиці для кожного блоку є 1 запис, індексований по номеру блоку. FAT-таблиця використовується подібно до зв’язного списку. Directory-запис містить номер першого блоку файлу. Запис таблиці, індексований цим номером блоку, містить номер наступного блоку файлу. Ланцюг продовжується, поки не буде досягнуто останній блок, який має спеціальне end-of-file значення, записане в таблицю. Невикористаний блок позначається в таблиці нулем. Виділення файлу нового блоку відбувається шляхом знаходження першого запису в таблиці з 0 та заміни попереднього end-of-file значення на адресу нового блоку. Нуль потім заміняється на end-of-file значення. Демонстрація зображена на рис. 6.

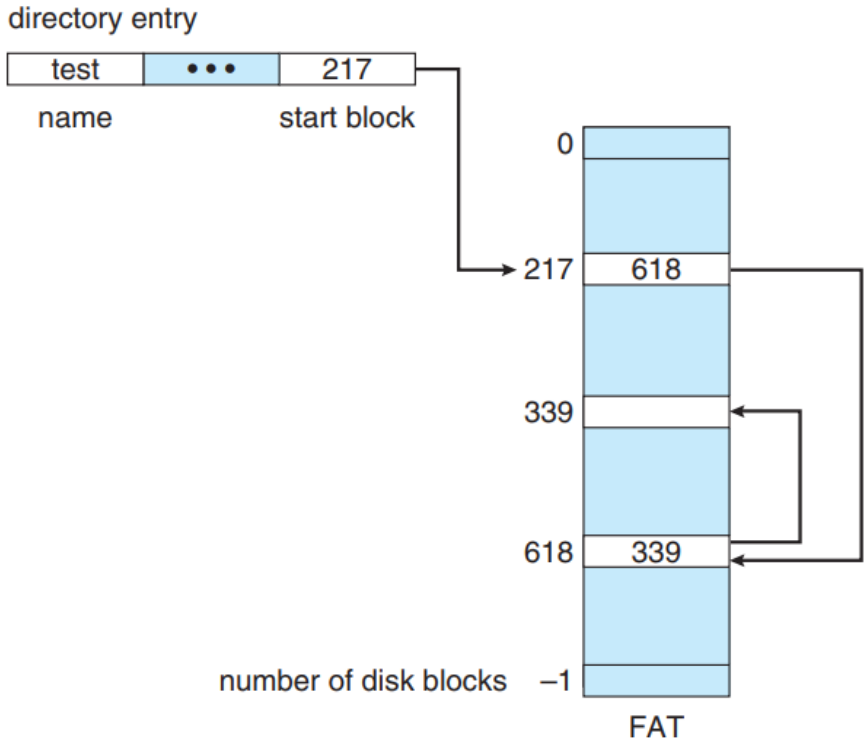

Рис. 6. Таблиця файлової алокації

Схема розміщення FAT може призводити до значної кількості переміщень зчитуючої голівки, якщо FAT не буде кешуватись. Голівка повинна переміщуватись на початок тому, щоб зчитати FAT та знайти розташування шуканого блоку (block in question), потім рухається до розташування самого блоку. У найгіршому випадку обидва рухи відбуваються для кожного з блоків. Перевага полягає в тому, що час довільного доступу покращується, оскільки голівка може знайти розташування будь-якого блоку, читаючи інформацію в FAT.

Зв'язне розміщення вирішує проблеми зовнішньої фрагментації та оголошення розмірів, характерні для послідовного розміщення. Проте за відсутності FAT-таблиці зв'язне розміщення не може підтримувати ефективний прямий доступ, оскільки вказівники на блоки розпорошені разом з власне блоками даних по всьому диску та повинні отримуватись впорядковано. ***Індексоване розміщення (Indexed allocation)*** вирішує цю проблему, вносячи всі вказівники разом в одне розташування – ***блок індексів (index block)***.

Кожний файл має власний блок індексів – масив адрес storage-блоків. *i*-тий запис у блоці індексів вказує на *i*-тий блок файлу.Директорія містить адресу індексного блоку (рис. 7). Для знаходження та зчитування *i*-того блоку використовуємо вказівник на *i*-тий запис блоку індексів. Дана схема подібна до сторінкової організації пам'яті.

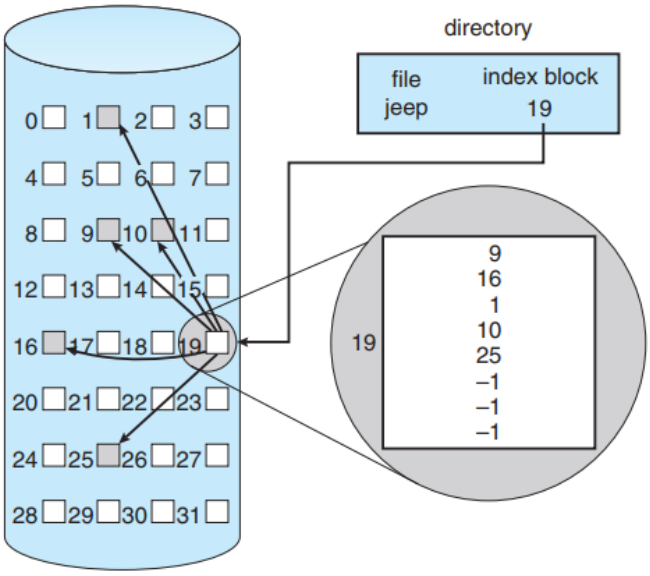

Рис.7. Індексоване виділення дискового простору

Коли файл створено, всі вказівники у блоці індексів встановлені як null. Коли *i*-тий блок вперше записується, сам блок отримується від менеджеру

вільного простору, а адреса цього блоку записується як і-тий запис у блоці індексів.

Індексоване розміщення підтримує прямий доступ, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

- ***Linked scheme***. An index block is normally one storage block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is null (for a small file) or is a pointer to another index block (for a large file).
- ***Багаторівневий індекс***. Варіант зв'язного представлення, який використовує індекс першого рівня, який вказуватиме на набір блоків з індексами другого рівня, які, в свою чергу, вказують на блоки файлу. Для доступу до блоку ОС використовує індекс першого рівня, щоб знайти блок індексів другого рівня, а потім використовує цей блок для знаходження бажаного блоку даних. Цей підхід можна продовжити до 3го чи 4го рівня, залежно від бажаного максимального розміру файлу. З блоками розміром 4Кб можемо зберегти 1024 чотирибайтних вказівники у блоці індексів (index block). Два рівні індексів дозволить розмістити 1048576 блоків даних та файл розміром до 4Гб.
- ***Змішана схема***. Інша альтернатива, яка використовується в файлових системах для ОС на базі UNIX – тримати перших, припустимо, 15 вказівників блоку індексів у структурі inode для файлу. Перші 12 з цих вказівників указують на прямі (direct) блоки (всередині адреси блоків, які містять дані файлу). Таким чином, дані для малих файлів (розміром не

більше 12 блоків) не потребуються окремого блоку індексів. Якщо орзмір блоку буде 4Кб, тоді до 48Кб даних можна отримати напряму. Наступні 3 вказівника указують на непрямі (indirect) блоки. Перший вказує на окремий непрямий блок, який є блоком індексів, який містить не дані, а адреси блоків, що містять дані. Другий вказує на подвійний непрямий блок (double indirect block), який містить адресу непрямого блоку. Останній вказівник містить адресу потрійного непрямого блоку (рис. 8.). Для цього методу кількість можливих для розміщення блоків файлу перевищує обсяг простору, адресованого 4-байтними вказівниками на файл, які використовуються в багатьох ОС. 32-біний вказівник на файл досягає лише 2^{32} байтів, тобто 4Гб. Багато реалізацій UNIX та Linux підтримують 64-бітні вказівники на файл, які дозволяють підтримувати файли та файлові системи розміром у кілька екзабайт. Файлова система ZFS підтримує 128-бітні вказівники на файл.

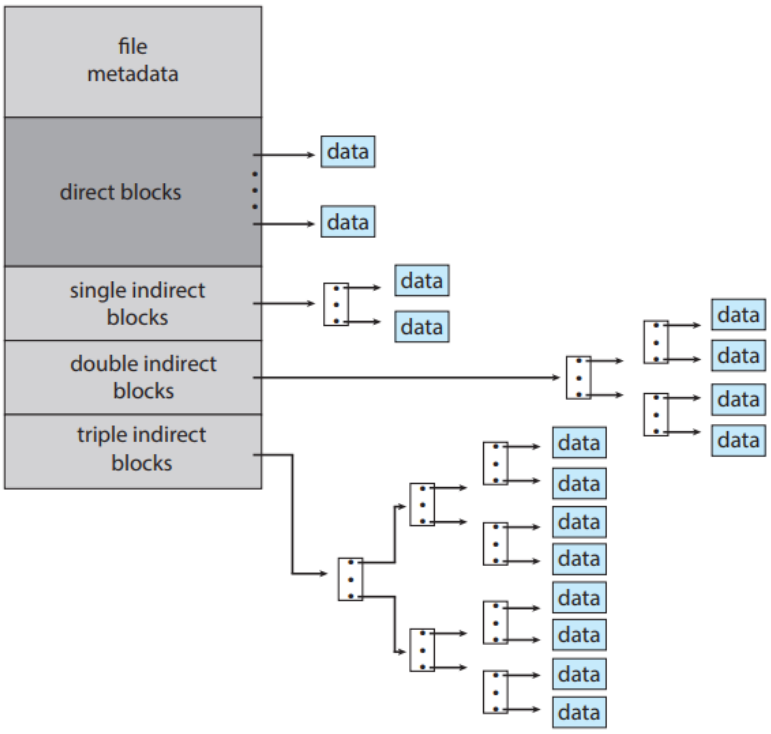

Рис. 8. inode в UNIX

Схеми індексованого виділення пам’яті страждають від тих же проблем продуктивності, що й зв’язне виділення. Зокрема, індексні блоки можуть кешуватись у пам’ять, проте блоки даних розпорошуватись по всьому тому.

Розглянуті методи виділення варіюються в ефективності зберігання та часах доступу до блоку даних. Обидва критерії важливі при виборі доречного методу чи методів для реалізації в ОС. Перед вибором методу алокації необхідно

визначити, як системи будуть використовуватись (переважатиме послідовний чи довільний доступ).

Для будь-якого типу доступу послідовне виділення потребує тільки 1 доступ для отримання блоку. Оскільки ми можемо тримати в пам’яті початкову адресу файлу, можна негайно обчислити адресу *i*-того блоку (або наступний блок) та напряму зчитати його.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the ith block might require i block reads. This problem indicates why linked allocation should not be used for an application requiring direct access. As a result, some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

Використовується багато інших оптимізацій. Given the disparity between CPU speed and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements. Furthermore, this

disparity is increasing over time, to the point where hundreds of thousands of instructions could reasonably be used to optimize head movements.

Для NVM-пристроїв немає процесу пошуку дискової голівки, тому потрібні різні алгоритми та оптимізації. Використовуючи старий алгоритм, який витрачає багато циклів ЦП, намагатись уникнути неіснуючого переміщення голівки буде неефективно. Існуючі та нові файлові системи змінюються для підтримки максимальної продуктивності NVM-накопичувачів. Ці розробки націлені на зниження кількості інструкцій та загального шляху від накопичувача до доступу додатку до даних.

4. Керування вільним дисковим простором

Оскільки простір сховища обмежений, необхідно повторно використовувати простір видалених файлів для виділення новим файлам за можливості (наприклад, ROM-диски записуються лише 1 раз в будь-який з доступних секторів, тому повторне використання неможливе). Для відстеження вільного простору система підтримує ***список вільних блоків (free-space list)***. Цей список стежить за всіма вільними блоками пристрою. Для створення файлу шукаємо в списку вільних блоків потрібний обсяг пам’яті для виділення новому файлу. Далі цей простір видаляється зі списку вільних блоків. Коли файл видаляється, його простір додається назад до списку вільних блоків. Незважаючи на слово «список», реалізація не обов’язково передбачає саме цю структуру даних.

Часто список вільних блоків реалізується як бітова карта або бітовий вектор. Кожний блок представляється 1 бітом (1 – якщо блок вільний, інакше – 0). Наприклад, розглянемо диск, де блоки 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 та 27 – вільні, а решта блоків виділені. Бітова карта виглядатиме так:

00**1111**00**111111**000**11**0000000**11**00000 ...

Основна перевага цього підходу – відносна простота та ефективність знаходження першого вільного блоку або *n* суміжних вільних блоків на диску. Комп'ютери постачають інструкції маніпулювання бітами, які можуть ефективно виконувати цю ціль. Один спосіб знаходження першого вільного блоку в системі, яка використовує бітовий вектор для виділення простору, – послідовно перевіряти кожне слово в бітмапі, щоб побачити, чи буде воно нульовим. Перше ненульове слово сканується на перший 1-біт, який буде розташуванням першого вільного блоку. Обчислення номеру блоку здійснюється так:

(кількість бітів у слові) × (кількість нульових слів) + зсув першого 1-біту.

На жаль, бітові вектори неефективні, поки весь вектор тримається в основній пам’яті (and is written to the device containing the file system occasionally for recovery needs). Тримати вектор в основній пам’яті можливо для менших пристроїв, проте не є необхідним для більших накопичувачів. Диск розміром 1.3Гб з 512-байтними блоками потребуватиме бітмап розміром понад 332Кб, щоб відстежувати вільні блоки, хоч кластеризація блоків в групах по 4 зменшить дане число до 83Кб на диск. Диск розміром 1Тб з блоками розміром 4Кб вже вимагатиме 32 Мб (2^{40} / 2^{12} = 2^{28} бітів = 2^{25} байтів = 32 Мб) для збереження бітового вектору.

Інший підхід до керування вільним простором – з’єднати разом усі вільні блоки, тримаючи вказівник на перший вільний блок у спеціальному розташуванні в файловій системі та кешуючи його в пам’ять. Цей перший блок містить вказівник на наступний вільний блок і т. д. Для попереднього прикладу триматимемо вказівник на блок 2 в якості першого вільного блоку (рис. 9).

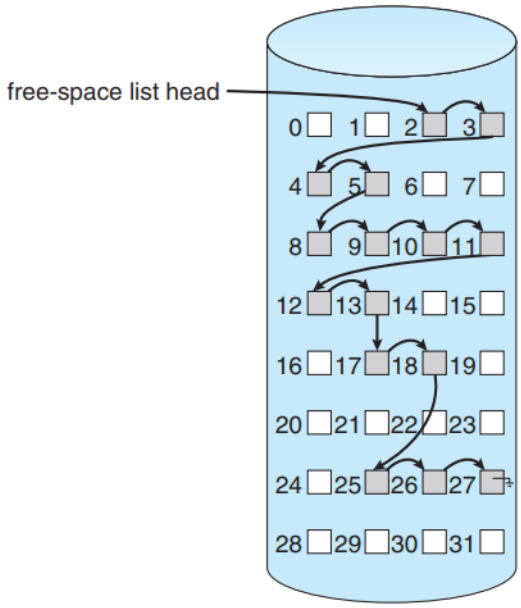

Рис. 9. Зв’язний список вільних блоків на диску

Дана схема неефективна; для обходу списку необхідно зчитати кожний блок, що вимагає суттєвого часу на ввід-вивід для ЖД. На щастя, обхід цього списку – нечаста дія. Зазвичай ОС просто потребує вільний блок, щоб виділити його файлу. Метод FAT збирає інформацію про вільні блоки в алокаційну структуру даних. Окремого методу не потрібно.

Групування. Модифікація підходу зі списком вільних блоків, яка зберігає адреси *n* вільних блоків у першому вільному блоці. Перші *n-1* з цих блоків дійсно вільні. Останній блок містить адреси інших *n* вільних блоків і т. д. Адреси великої

кількості вільних блоків тепер знаходяться швидко, на відміну від ситуації, коли використовується стандартний підхід зі зв'язним списком.

Підрахунок. Інший підхід отримує переваги від того факту, що загалом кілька суміжних блоків можуть виділятись або звільнятись одночасно, особливо коли простір виділений алгоритмом послідовного виділення або за допомогою кластерингу. Таким чином, можемо тримати адресу першого вільного блоку та кількість *n* вільних суміжних блоків, які йдуть після нього. Кожний запис у списку вільних блоків тоді складається з device address та count. Хоч кожний запис вимагає більше місця, ніж проста адреса, загальний список коротший, якщо count зазвичай більше за 1. Ці записи можна зберігати у збалансованому дереві, а не в зв'язному списку, для ефективного пошуку, вставки або видалення.

5. Практичні завдання

Змоделюйте процеси виділення пам'яті на диску та керування вільним простором засобами файлової системи. Оцініть ефективність (швидкість доступу до вмісту файлу) та гнучкість (здатність тримати файли різних розмірів) для відповідних алгоритмів (завдання). Програмна реалізація передбачає наступні блоки:

- ***Блок управління томом (Volume control block)***: даний блок відображає загальну кількість блоків, кількість вільних блоків, розмір блоку, вказівники на вільні блоки або бітову карту вільних блоків.
- ***Структура директорій***: необхідна для відслідковування основної інформації про файли та розташування, в якому зберігається вміст файлів. Можна використовувати лінійний список або хеш-таблицю.
- ***Виділення блоків диску***: програма повинна реалізувати всі 3 стандартні методи виділення дискового простору (безперервне, зв'язане та індексоване виділення).
- ***Керування вільним простором***: для визначення розташувань вільних блоків буде застосовуватись бітовий вектор або зв'язний список.

Носій інформації представлено масивом зі 128 записів із записами <user input> як блок диску. Показана нижче таблиця – простий приклад перших 6 блоків фізичного сховища, яке містить 2 різних файли. Зауважте, що цей приклад передбачає 5 записів на блок. Структура директорії зберігається у першому блоці, показуючи інформацію щодо 2 файлів. У цьому прикладі перше число, 100 – файловий ідентифікатор. Друге число – початковий індекс першого блоку даних (тобто 1), третє число – індекс заключного блоку даних (тут – 4).

Index	Block	File Data
0	0	<v.ctrl B>
1	0	100,1,4
2	0	200,5,5
3	0	
4	0	
5	1	101
6	1	102
7	1	103
8	1	104
9	1	4
10	2	
11	2	
12	2	
13	2	
14	2	

Index	Block	File Data
15	3	
16	3	
17	3	
18	3	
19	3	
20	4	105
21	4	106
22	4	
23	4	
24	4	
25	5	201
26	5	202
27	5	203
28	5	
29	5	

Файл представляється ідентифікатором з діапазону від 100 до 9999 (тобто 100, 200, … 1000, 1100, …., 9800, 9900). Вміст файлу в прикладі представляється наступним цілим числом відносно ідентифікатора (тобто 1101, 1102, 1103, …).

Кожний запис у масиві включає одне число, яким може бути або вміст файлу, або вказівник на інший блок. Цей підхід застосовується до всього масиву записів, крім структури директорії, яка може містити кілька атрибутів, як показано в прикладі. Дана структура змінюватиметься залежно від методу виділення пам’яті. Проте кожний запис у структурі директорії може містити лише інформацію щодо одного файлу.

Ключові кроки програми:

1) Програма повинна починати моделювання з процесу форматування, запитуючи користувача розмір блоку та кількість блоків. Весь дисковий простір форматується відповідно до цих налаштувань.
2) Програма має зчитувати csv-файл на вхід. Перше поле – файлова операція, яка включає додавання, зчитування чи видалення файлу. Друге поле – файловий ідентифікатор, а решта – дані з файлу. Кожний рядок представляє операцію, яку програмі потрібно виконати над одним файлом.
   add, 100, 101, 102, 103, 104, 105, 106
   add, 200, 201, 202, 203
   read, 106
   delete, 200
3) На основі методу виділення дискового блоку програма завершить операцію відповідно до інструкції та виведе статус дії. Операція add призведе до двох

повідомлень. Перше для виводу переліку знайдених вільних блоків для файлу разом з «часом», необхідним для виділення вільного блоку. «Час» визначаємо як кількість кроків, необхідних програмі для обходу array entry. Друге повідомлення показує розташування файлу з ідентифікаторами усіх виділених блоків. Список має бути з правильною послідовністю блоків, у яких зберігається файл. Виводьте помилку, якщо файл неможливо зберегти через нестачу пам'яті.

Число після операції read вказує на вміст файлу (тобто 106), який потрібно зчитати. Вивід включатиме, які блоки та адреси зчитуються, а також access "time".

Для операції delete вказуйте лише файл, який видаляється, та виведіть звільнені блоки. Виведіть відповідне повідомлення про помилку, якщо файл неможливо додати (файл вже існує), зчитати (файлу немає) або видалити (файлу немає). Приклад виводу:

Adding file100 and found free B1,B4
Added file100 at B1(101,102,103,104), B04(105,106)
Adding file200 and found free B5
Added file200 at B5(201,202,203)
Read file100(106) from <you will decide how it can be processed>
Deleted file200 and freed B5

4) Остаточний вивід програми має бути картою диску, відображаючи повний вміст фізичного сховища (всі 128 записів). Можете використовувати таблицю 1 як керівництво. Також слід виводити загальну кількість доданих файлів, загальний розмір та витрачений "час" in the addition. Це ж стосується й операції read.

Вибір методів для організації роботи файлової системи здійснюється відповідно до номеру в списку підгрупи:

№	Структура директорій	Виділення дискових блоків	Керування вільним простором
1.	Список	Послідовне	Бітова карта
2.	Хеш-таблиця	Зв'язне	Зв'язний список
3.	Список	Індексоване	Зв'язний список
4.	Хеш-таблиця	Послідовне	Бітова карта
5.	Список	Зв'язне	Зв'язний список
6.	Хеш-таблиця	Індексоване	Бітова карта

7.	Список	Послідовне	Зв’язний список
8.	Хеш-таблиця	Зв’язне	Бітова карта
9.	Список	Індексоване	Зв’язний список
10.	Хеш-таблиця	Послідовне	Зв’язний список
11.	Список	Зв’язне	Бітова карта
12.	Хеш-таблиця	Індексоване	Зв’язний список
13.	Список	Послідовне	Бітова карта
14.	Хеш-таблиця	Зв’язне	Бітова карта
15.	Список	Індексоване	Зв’язний список

Приблизний вигляд подібного проекту можна знайти тут.