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.

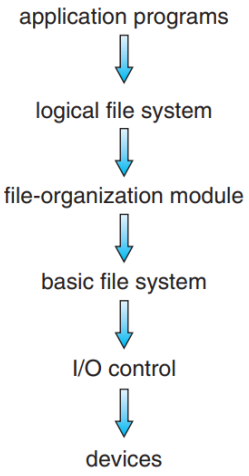

Рис. 1. Шарувата структура файлової системи

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, the logical file system can call the file-organization module to map the directory I/O into storage block locations, which are passed on to the basic file system and I/O control system.

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 an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open. The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It could be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a ***file descriptor***; Windows refers to it as a ***file handle***.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata are copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

The caching aspects of file-system structures should not be overlooked. Most systems keep all information about an open file, except for its actual data blocks, in memory. The BSD UNIX system is typical in its use of caches wherever disk I/O can be saved. Its average cache hit rate of 85% shows that these techniques are well worth implementing.

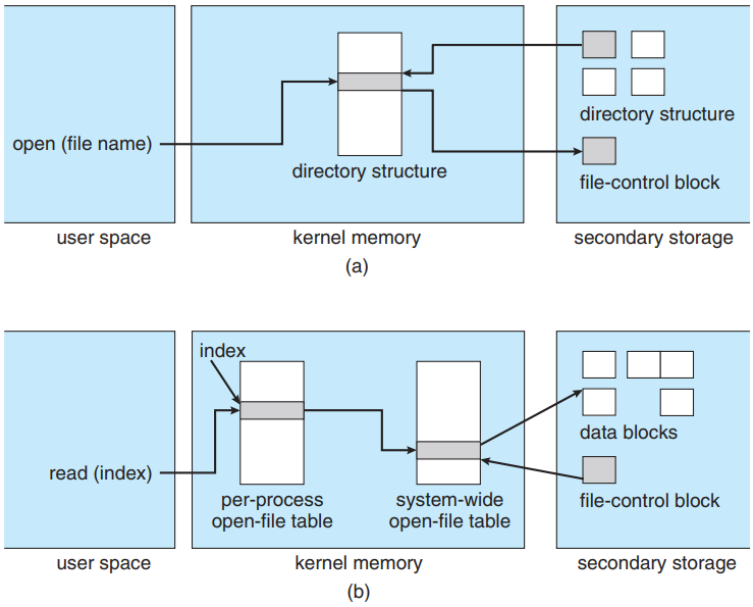

Рис. 3. Структури вбудованої файлової системи. (a) Відкриття файлу, (b) Зчитування файлу.

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. A cache hit avoids the need to constantly reread the information from secondary storage. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.

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.

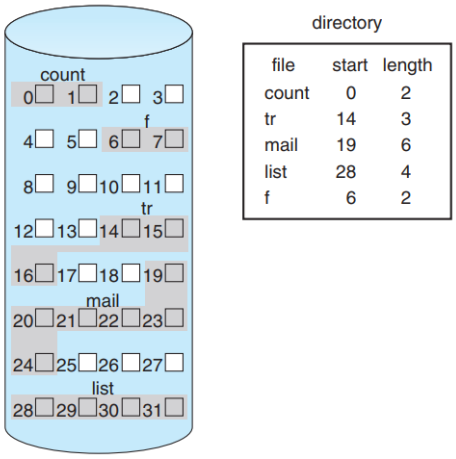

Рис. 4. Послідовне виділення дискового простору

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. We then copy the files back onto the original device by allocating contiguous space from this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem.

The cost of this compaction is time, however, and the cost can be particularly high for large storage devices. Compacting these devices may take hours and may be necessary on a weekly basis. Some systems require that this function be done off-line, with the file system unmounted. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. Most modern systems that need defragmentation can perform it on-line during normal system operations, but the performance penalty can be substantial.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate.

If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening, however; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, an operating system can use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated. The commercial Symantec Veritas file system uses extents to optimize performance. Veritas is a high-performance replacement for the standard 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). A write to the file causes the freespace management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

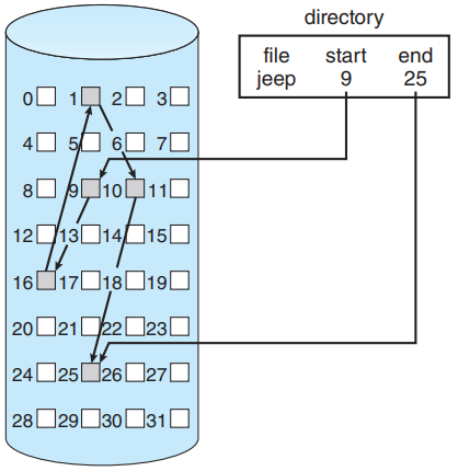

Рис. 5. Зв’язне виділення дискового простору

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block. Each access to a pointer requires a storage device read, and some require an HDD seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the secondary storage device only in cluster units. Pointers then use a much smaller percentage of the file's space. This method allows the logical-to-physical block mapping to remain simple but improves HDD throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Also random I/O performance suffers because a request for a small amount of data transfers a large amount of data. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the device, and consider what would happen if a pointer was lost or damaged. A bug in the operating-system software or a hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file.

An important variation on linked allocation is the use of a ***file-allocation table (FAT)***. This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for each block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 6 for a file consisting of disk blocks 217, 618, and 339.

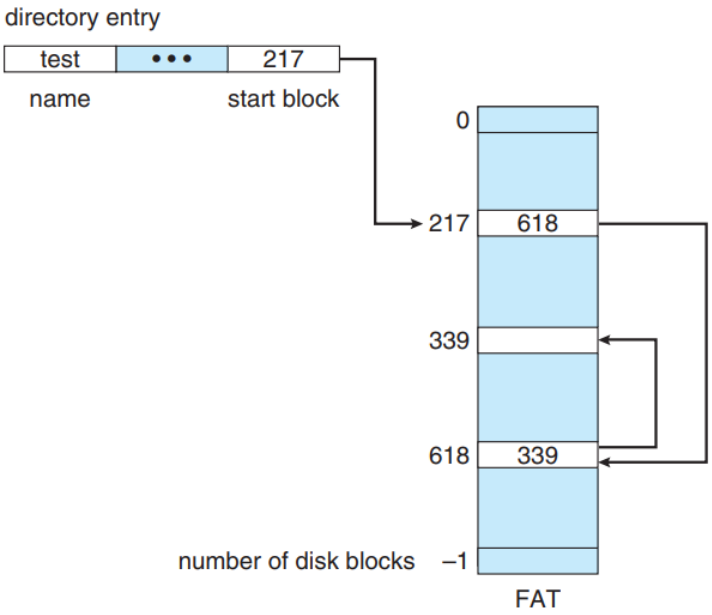

Рис. 6. File-allocation table.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. ***Indexed allocation*** solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of storage-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block (Figure 7). To find and read the ith block, we use the pointer in the ith index-block entry. This scheme is similar to the paging scheme described in Section 9.3.

When the file is created, all pointers in the index block are set to null. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Indexed allocation supports direct access, 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.

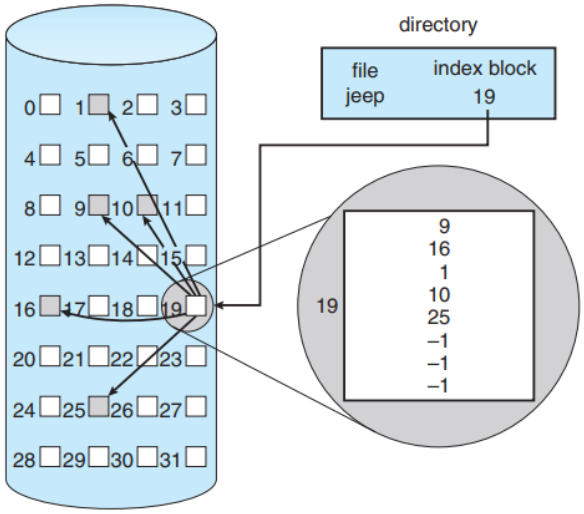

Рис.7 Indexed allocation of disk space

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).
- ***Multilevel index***. Avariant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

– ***Combined scheme***. Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The nextthree pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block. (A UNIX inode is shown in Figure 14.8.). Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 232 bytes, or 4 GB. Many UNIX and Linux implementations now support 64-bit file pointers, which allows files and file systems to be several exbibytes in size. The ZFS file system supports 128-bit file pointers.

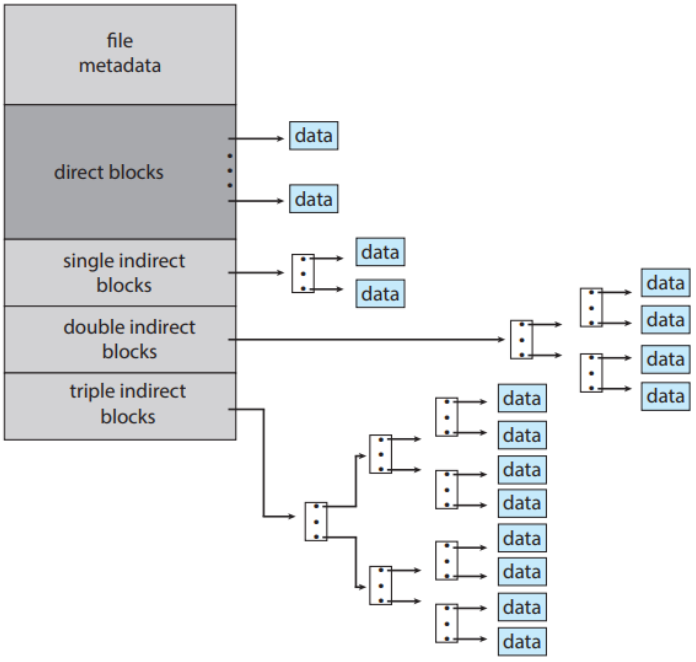

Рис. 8. The UNIX inode

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method

or methods for an operating system to implement. Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should not use the same method as a system with mostly random access.

For any type of access, contiguous allocation requires only one access to get a block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the address of the ith block (or the next block) and read it directly.

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.

Many other optimizations are in use. 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.

For NVM devices, there are no disk head seeks, so different algorithms and optimizations are needed. Using an old algorithm that spends many CPU cycles trying to avoid a nonexistent head movement would be very inefficient. Existing file systems are being modified and new ones being created to attain maximum performance from NVM storage devices. These developments aim to reduce the instruction count and overall path between the storage device and application access to the data.

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.

Frequently, the free-space list is implemented as a bitmap or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

001111001111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One technique for finding the first free block on a system that uses a bit vector to allocate space is to sequentially check each word in the bitmap to see whether that

value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

(кількість бітів у слові) × (кількість 0-value слів) + зсув of first 1 bit.

Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to the device containing the file system occasionally for recovery needs). Keeping it in main memory is possible for smaller devices but not necessarily for larger ones. A 1.3-GB disk with 512-byte blocks would need a bitmap of over 332 KB to track its free blocks, although clustering the blocks in groups of four reduces this number to around 83 KB per disk. A 1-TB disk with 4-KB blocks would require 32 MB ($2^{40} / 2^{12} = 2^{28}$ bits = 2^{25} bytes = 25 MB) to store its bitmap. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on. Recall our earlier example, in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 9).

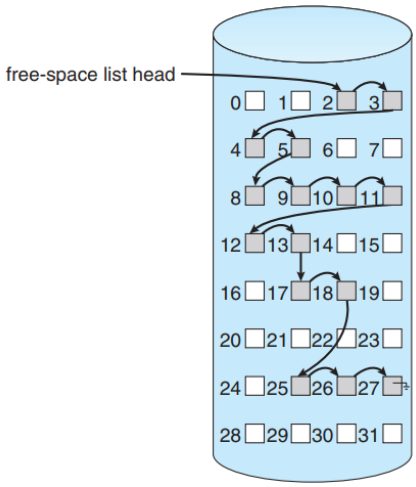

Рис. 9. Linked free-space list on disk

This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time on HDDs. Fortunately, however, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method

incorporates free-block accounting into the allocation data structure. No separate method is needed.

Grouping. A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Counting. Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a device address and a count. Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1. Note that this method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Space Maps. Oracle's ZFS file system (found in Solaris and some other operating systems) was designed to encompass huge numbers of files, directories, and even file systems (in ZFS, we can create file-system hierarchies). On these scales, metadata I/O can have a large performance impact. Consider, for example, that if the free-space list is implemented as a bitmap, bitmaps must be modified both when blocks are allocated and when they are freed. Freeing 1 GB of data on a 1-TB disk could cause thousands of blocks of bitmaps to be updated, because those data blocks could be scattered over the entire disk. Clearly, the data structures for such a system could be large and inefficient.

In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures. First, ZFS creates ***metaslabs*** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks. Rather than write counting structures to disk, it uses log-structured file-system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays

the log into that structure. The in-memory space map is then an accurate representation of the allocated and free space in the metaslab. ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry. Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS. During the collection and sorting phase, block requests can still occur, and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree is the free list.

TRIMing Unused Blocks. HDDs and other storage media that allow blocks to be overwritten for updates need only the free list for managing free space. Blocks do not need to be treated specially when freed. A freed block typically keeps its data (but without any file pointers to the block) until the data are overwritten when the block is next allocated.

Storage devices that do not allow overwrite, such as NVM flash-based storage devices, suffer badly when these same algorithms are applied. Recall from Section 11.1.2 that such devices must be erased before they can again be written to, and that those erases must be made in large chunks (blocks, composed of pages) and take a relatively long time compared with reads or writes.

A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure (once the block containing the page is entirely free). That mechanism varies based on the storage controller. For ATA-attached drives, it is TRIM, while for NVMe-based storage, it is the unallocate command. Whatever the specific controller command, this mechanism keeps storage space available for writing. Without such a capability, the storage device gets full and needs garbage collection and block erasure, leading to decreases in storage I/O write performance (known as “a write cliff”). With the TRIM mechanism and similar capabilities, the garbage collection and erase steps can occur before the device is nearly full, allowing the device to provide more consistent performance.

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 with <user input> entries as a disk block. 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’язане	Зв’язний список
3.	Список	Індексоване	Зв’язний список
4.	Хеш-таблиця	Послідовне	Бітова карта
5.	Список	Зв’язане	Зв’язний список
6.	Хеш-таблиця	Індексоване	Бітова карта
7.	Список	Послідовне	Зв’язний список
8.	Хеш-таблиця	Зв’язане	Бітова карта
9.	Список	Індексоване	Зв’язний список
10.	Хеш-таблиця	Послідовне	Зв’язний список
11.	Список	Зв’язане	Бітова карта
12.	Хеш-таблиця	Індексоване	Зв’язний список
13.	Список	Послідовне	Бітова карта
14.	Хеш-таблиця	Зв’язане	Бітова карта
15.	Список	Індексоване	Зв’язний список

Приблизний вигляд подібного проекту можна знайти тут.