ЛЕКЦІЯ 8. ОРГАНІЗАЦІЯ ФАЙЛОВИХ СИСТЕМ

Логічна організація файлових систем

- Структури файлів і файлових систем
- Операції над файлами і каталогами
- Міжпроцесова взаємодія у файловій системі
- Поіменовані канали

Фізична організація і характеристики файлових систем

- Розміщення файлів і каталогів на жорсткому диску
- Продуктивність файлових систем
- Надійність файлових систем

ЛОГІЧНА ОРГАНІЗАЦІЯ ФАЙЛОВИХ СИСТЕМ

Файлові системи можна розглядати на двох рівнях: логічному і фізичному. Логічний визначає відображення файлової системи, призначене для прикладних програм і користувачів, фізичний – особливості розташування структур даних системи на диску й алгоритми, які використовують під час доступу до інформації.

8.1. Поняття файла і файлової системи

Файл – це набір даних, до якого можна звертатися за іменем. Файли організовані у *файлові системи. З* погляду користувача файл є мінімальним обсягом даних файлової системи, з яким можна працювати незалежно.

Типи файлів. Раніше ОС підтримували файли різної спеціалізованої структури. Сьогодні є тенденція взагалі не контролювати на рівні ОС структуру файла, відображаючи кожен файл простою послідовністю байтів. У цьому разі застосування, які працюють із файлами, самі визначають їхній формат.

Будь-який тип файлів вимагає спеціальної програми для роботи з ним. Частина файлів має вбудовану підтримку в самій операційній системі, причому чим новіша версія ОС, тим з більшою кількістю типів файлів вона може працювати без установки додаткового програмного забезпечення. Інша частина файлів вимагає для роботи з ними установки програмного забезпечення.

Якщо задати невірне розширення файлу, то при спробі запуску цього файлу при "не тій" програмі може виникнути серйозний збій в роботі операційної системи.

Найпоширеніший тип файлів, який не вимагає установки програмного забезпечення для запуску, — це файли, що запускаються, з розширеннями COM і EXE.

Командний файл — це простий текстовий файл з розширенням BAT або CMD, вміст якого дотримується визначеного синтаксису. Щоб виконати командний файл, досить ввести в командний рядок його ім'я.

Існує ще цілий ряд "стандартних" розширень:

SYS — системний файл, що містить драйвер якого-небудь пристрою;

TXT — текстовий файл, створений будь-яким текстовим редактором;

DOC — текстовий файл, створений у Microsoft Word або інших сумісних програмах;

BAK, OLD — старі копії системних файлів;

ARJ, RAR, ZIP — файли, створені найбільш поширеними архіваторами;

BMP, JPG, GIF, PNG, PDS та ін — графічні файли;

DBF — база даних;

XLS — електронна таблиця у форматі Microsoft Excel ;

DLL — системний файл, що містить бібліотеки підпрограм;

INI — файл ініціалізації деякої програми (інсталятора або навпаки деінсталятора);
HLP — файл допомоги;
PIF — ярлик на який-небудь файл;
WAV, MP3, WMA — звуковий файл;
AVI, MPG — файл, що містить відеоінформацію, і так далі
HTM, HTML — Web-сторінки
BAS, PAS, CPP та ін — код (текст) програми на мовах програмування

Ще одним варіантом класифікації є поділ на *файли із прямим і послідовним доступом.* Файли із прямим доступом дають змогу вільно переходити до будь-якої позиції у файлі, використовуючи для цього поняття *покажчика поточної позиції файла* (seek pointer), що може переміщатися у будь-якому напрямку за допомогою відповідних системних викликів. Файли із послідовним доступом можуть бути зчитані тільки послідовно, із початку в кінець. Сучасні ОС звичайно розглядають усі файли як файли із прямим доступом.

Кілька останніх символів імені (звичайно відокремлені від інших символів крапкою) у деяких системах називають ***розширенням файла,*** яке може характеризувати його тип. Важливою характеристикою файлової системи є ***максимальна довжина імені файла.*** У минулому багато ОС різним чином обмежували довжину імен файлів. Широко відоме було обмеження на 8 символів у імені файла і 3 – у розширенні, присутнє у файловій системі FAT до появи Windows 95. Сьогодні стандартним значенням максимальної довжини імені файла є 255 символів.

Файлова система – це підсистема ОС, що підтримує організований набір файлів, здебільшого у конкретній ділянці дискового простору (логічну структуру); низькорівневі структури даних, використовувані для організації цього простору у вигляді набору файлів (фізичну структуру); програмний інтерфейс файлової системи (набір системних викликів, що реалізують операції над файлами).

Файлова система надає прикладним програмам абстракцію файла. Прикладні програми не мають інформації про те, як організовані дані файла, як знаходять відповідність між ім'ям файла і його даними, як пересилають дані із диска у пам'ять тощо – усі ці операції забезпечує файлова система.

До головних задач файлової системи можна віднести: організацію її логічної структури та її відображення на фізичну організацію розміщення даних на диску; підтримку програмного інтерфейсу файлової системи; забезпечення стійкості проти збоїв; забезпечення розподілу файлових ресурсів за умов багатозадачності та захисту даних від несанкціонованого доступу.

Зупинимося на тому, як організовують дисковий простір для розміщення на ньому файлової системи, і введемо поняття *розділу.* Розділи реалізують логічне відображення фізичного диска.

Розділ (partition) – частина фізичного дискового простору, що призначена для розміщення на ній структури однієї файлової системи і з логічної точки зору розглядається як єдине ціле. Розділ – це логічний пристрій, що з погляду ОС функціонує як окремий диск. Такий пристрій може відповідати всьому фізичному диску (у цьому разі кажуть, що диск містить один розділ); найчастіше він відповідає частині диска (таку частину називають ще фізичним розділом); буває й так, що подібні логічні пристрої поєднують кілька фізичних розділів, що перебувають, можливо, на різних дисках (такі пристрої ще називають *логічними томами* – logical volumes).

Кожний розділ може мати свою файлову систему (і, можливо, використовуватися різними ОС). Для поділу дискового простору на розділи використовують спеціальну

утиліту, яку часто називають **fdisk.** Для генерації файлової системи на розділі потрібно використати операцію високорівневого форматування диска. У деяких ОС під *томом* (volume) розуміють розділ із встановленою на ньому файловою системою.

Реалізація розділів дає змогу відокремити логічне відображення дискового простору від фізичного і підвищує гнучкість використання файлових систем.

Розділи є основою організації великих обсягів дискового простору для розгортання файлових систем. Для організації файлів у рамках розділу зі встановленою файловою системою було запропоновано поняття файлового *каталогу* (file directory).

Каталог – це об'єкт (найчастіше реалізований як спеціальний файл), що містить інформацію про набір файлів. Про такі файли кажуть, що вони містяться в каталозі. Файли заносяться в каталоги користувачами на підставі їхніх власних критеріїв.

Базовою ідеєю організації даних за допомогою каталогів є те, що вони можуть містити інші каталоги. Вкладені каталоги називають підкаталогами (subdirectories). Таким чином формують ***дерево каталогів***. Перший каталог, створений у файловій системі, встановленій у розділі (корінь дерева каталогів), називають кореневим каталогом (root directory).

Для визначення місцезнаходження файла потрібно додавати до його імені список каталогів, де він перебуває. Такий список називають ***шляхом*** (path). Каталоги у шляху перераховують зліва направо – від меншої глибини вкладеності до більшої. Роздільник каталогів у шляху відрізняється для різних систем: в UNIX прийнято використовувати прямий слеш «/», а у Windows-системах – зворотний «\». O:\OC\LEC\lec08.doc

Залишилося з'ясувати важливе питання про взаємозв'язок розділів і структури каталогів файлових систем. Розрізняють ***два основні підходи*** до реалізації такого взаємозв'язку, які істотно відрізняються один від одного.

Перший підхід в основному використовується у файловій системі UNIX і полягає в тому, що розділи зі встановленими на них файловими системами об'єднуються в єдиному дереві каталогів ОС. Стандартну організацію каталогів UNIX зображують у вигляді дерева з одним коренем – кореневим каталогом, який позначають «/». Файлову систему, на якій перебуває кореневий каталог, називають завантажувальною або кореневою. У більшості реалізацій вона має містити файл із ядром ОС.

Другий підхід, що в основному поширений в лініях Windows, припускає, що кожний розділ зі встановленою файловою системою є видимим для користувача і позначений буквою латинського алфавіту. Такий розділ звичайно називають томом. Позначення томів нам знайомі – це C:, D: тощо.

Структура каталогів файлової системи не завжди є деревом. Багато файлових систем дає змогу задавати кілька імен для одного й того самого файла. Такі імена називають ***зв'язками*** (`links`). Розрізняють *жорсткі та символічні зв'язки*.

Кожний файл має набір характеристик – ***атрибутів***. Набір атрибутів змінюється залежно від файлової системи. Найпоширеніші атрибути файлів:

- *Ім'я файла*,
- *Тип файла*, який звичайно задають для спеціальних файлів (каталогів, зв'язків тощо),
- *Розмір файла* (зазвичай для файла можна визначити його поточний, а іноді й максимальний розмір).
- *Атрибути безпеки*, що визначають права доступу до цього файла,
- *Часові атрибути*, до яких належать час створення останньої модифікації та останнього використання файла.

8.2. Операції над файлами і каталогами

Підходи до використання файлів із процесу бувають такі: *зі збереженням* (`stateful`) і *без збереження стану* (`stateless`).

У разі збереження стану є спеціальні операції, які готують файл до використання у процесі *(відкривають його)* і скасовують цю готовність *(закривають його)*. Інші операції використовують структури даних, підготовлені під час відкриття файла, і можуть виконуватися тільки доти, поки файл не буде закритий. Перевагою такого підходу є висока продуктивність, оскільки під час відкриття файла потрібні структури даних завантажуються у пам'ять.

Якщо стан не зберігають, кожна операція роботи із файлом (читання, записування тощо) супроводжується повною підготовкою файла до роботи (кожна операція починається відкриттям файла і завершується закриттям). Хоча такий підхід програє у продуктивності, його можна використати для підвищення надійності роботи системи (наприклад, коли файлову систему використовують через мережу, тому що у будь-який момент може статися розрив мережного з'єднання).

Назвемо основні файлові операції, які звичайно надає операційна система для використання у прикладних програмах.

- ***Створення файла.*** Ця операція спричиняє створення на диску нового файла нульової довжини. Після створення файл автоматично відкривають.
- ***Відкриття файла.*** Після відкриття файла процес може із ним працювати (наприклад, робити читання і записування). Відкриття файла зазвичай передбачає завантаження в оперативну пам'ять спеціальної структури даних – *дескриптора файла,* який визначає його атрибути та місце розташування на диску. Наступні виклики використовуватимуть цю структуру для доступу до файла.
- ***Закриття файла.*** Після завершення роботи із файлом його треба закрити. При цьому структуру даних, створену під час його відкриття, вилучають із пам'яті. Усі дотепер не збережені зміни записують на диск.
- ***Вилучення файла.*** Ця операція спричиняє вилучення файла і вивільнення зайнятого ним дискового простору. Вона зазвичай недопустима для відкритих файлів.
- ***Читання з файла.*** Ця операція звичайно зводиться до пересилання певної кількості байтів із файла, починаючи із поточної позиції, у заздалегідь виділений для цього буфер пам'яті режиму користувача.
- ***Запис у файл.*** Здійснюють із поточної позиції, дані записують у файл із заздалегідь виділеного буфера. Якщо на цій позиції вже є дані, вони будуть перезаписані. Ця операція може змінити розмір файла.
- ***Переміщення покажчика поточної позиції.*** Перед операціями читання і записування слід визначити, де у файлі перебувають потрібні дані або куди треба їх записати, задавши за допомогою цієї операції поточну позицію у файлі. Зазначимо, що якщо перемістити покажчик файла за його кінець, а потім виконати операцію записування, довжина файла збільшиться.
- ***Отримання і задавання атрибутів файла.*** Ці дві операції дають змогу зчитувати поточні значення всіх або деяких атрибутів файла або задавати для них нові значення.

8.2.1. Файлові операції POSIX

Усі UNIX-системи реалізують доступ до файлів за допомогою компактного набору системних викликів, визначеного стандартом POSIX, який відповідає набору файлових

операцій.

Для ***відкриття файла*** використовують системний виклик `open()`, першим параметром якого є шлях до файла.

```
#include <fcntl.h>
int open(const char *pathname, int flags[, mode_t mode]);
```

Виклик `open()` повертає цілочислове значення – *файловий дескриптор.* Його слід використовувати в усіх викликах, яким потрібен відкритий файл. У разі помилки цей виклик поверне -1, а значення змінної errno відповідатиме коду помилки.

Розглянемо деякі значення, яких може набувати параметр `flags` (їх можна об'єднувати за допомогою побітового «або»):

- 0_RDONLY, 0_WRONLY, 0_RDWR – відкриття файла, відповідно, тільки для читання, тільки для записування або для читання і записування (має бути задане одне із цих трьох значень, наведені нижче не обов'язкові);
- 0_CREAT – якщо файл із таким ім'ям відсутній, його буде створено, якщо файл є і увімкнуто прапорець 0_EXCL, буде повернено помилку;
- 0_TRUNC – якщо файл відкривають для записування, його довжину покладають рівною нулю;
- 0_NONBLOCK – задає *неблокувальне введення-виведення;* особливості його використання розглянемо разом із викликом `read()`.

Параметр `mode` потрібно задавати тільки тоді, коли задано прапорець 0_CREAT. Значенням у цьому випадку буде вісімкове число, що задає права доступу до файла. Докладно ці права буде розглянуто надалі, а поки що задаватимемо як аргумент значення 0644. Ось приклад використання цього системного виклику:

```
// відкриття файла для записування
int fd1=open("./myfile.txt", 0_WRONLY|0_CREAT|0_TRUNC, 0644);
// відкриття файла для читання, помилка, якщо файла немає
int fd1 = open("./myfile.txt", 0_RDONLY);
```

Файл закривають за допомогою системного виклику `close()`, що приймає файловий дескриптор:

```
close(fdl);
```

Для ***читання*** даних із відкритого файла використовують системний виклик `read()`:

```
ssize_t read(int fdl, void *buf, size_t count);
```

Внаслідок цього виклику буде прочитано `count` байтів із файла, заданого відкритим дескриптором `fdl`, у пам'ять, на яку вказує `buf` (ця пам'ять виділяється заздалегідь). Виклик `read()` повертає реальний обсяг прочитаних даних (тип `ssize_t` є цілочисловим). Покажчик позиції у файлі пересувають за зчитані дані.

```
char buf[100];
// читають 100 байт з файлу в buf
int bytes_read = read(fdl, buf, sizeof(buf));
```

Коли потрібна кількість даних у конкретний момент відсутня (наприклад, `fdl` пов'язаний із мережним з'єднанням, яким ще не прийшли дані), поведінка цього виклику залежить від значення прапорця 0_NONBLOCK під час виклику `open()`.

У разі блокувального виклику (0_NONBLOCK не увімкнуто) він призупинить поточний потік до того часу, поки дані не з'являться, а в разі неблокувального (прапорець 0_NONBLOCK увімкнуто) – зчитає всі доступні дані й завершиться, призупинення потоку не відбудеться.

Для ***записування*** даних у відкритий файл через файловий дескриптор використовують системний виклик `write()`:

```
ssize_t write(int fdl, const void *buf, size_t count);
```

Внаслідок цього виклику буде записано `count` байтів у файл через дескриптор `fdl` із пам'яті, на яку вказує `buf`. Виклик `write()` повертає обсяг записаних даних.

```
int fdl, bytes_written;
fdl = open("./myfile.txt", 0_RDWR|0_CREAT, 0644);
bytes_written = write(fdl, "hello", sizeof("hello");
```

Наведемо приклад реалізації ***копіювання*** файлів за допомогою засобів POSIX.

```
char buf[1024];
int bytes_read, infile, outfile;
// відкриття вихідного файла для читання
infile = open("infile.txt", 0_RDONLY);
if (infile == -1) {
    printf ("помилка під час відкриття файла\n"); exit(-1);
}
// створення результуючого файла, перевірку помилок пропущено
outfile=open("outfile.txt", 0_WRONLY|0_CREAT|0_TRUNC, 0644);
do {
  // читання даних із вихідного файла у буфер
  bytes_read = read(infile, buf, sizeof(buf));
  // записування даних із буфера в результуючий файл
  if (bytes_read > 0) write(outfile, buf, bytes_read);
} while (bytes_read > 0);
// закриття файлів
close(infile);
close(outfile);
```

Кожному відкритому файлу відповідає ***покажчик позиції*** (зсув) усередині файла. Його можна пересувати за допомогою системного виклику `lseek()`:

```
off_t lseek(int fdl, off_t offset, int whence);
```

Параметр `offset` задає величину переміщення покажчика. Режим переміщення задають параметром `whence`, який може набувати значень SEEK_SET (абсолютне переміщення від початку файла), SEEK_CUR (відносне переміщення від поточного місця покажчика позиції) і SEEK_END (переміщення від кінця файла).

```
//переміщення покажчика позиції на 100 байт від поточного місця
lseek(outfile, 100, SEEK_CUR);
// записування у файл
write (outfile, "hello", sizeof("hello"));
```

Коли покажчик поточної позиції перед операцією записування опиняється за кінцем файла, він внаслідок записування автоматично розширюється до потрібної довжини. На цьому ґрунтується ефективний спосіб створення файлів необхідного розміру:

```
int fdl = open("file", 0_RDWR|0_CREAT|0_TRUNC, 0644);
// створення файла
lseek(fdl, needed_size, SEEK_SET);
// розширення до потрібного розміру
write(fdl, "", 1);
// записування нульового байта
```

Для отримання ***інформації про атрибути*** файла (тобто про вміст його індексного дескриптора) використовують системний виклик `stat()`.

```
#include <sys/stat.h>
int stat(const char *path, struct stat *attrs);
```

Першим параметром є шлях до файла, другим – структура, у яку записуватимуться атрибути внаслідок виклику. Деякі поля цієї структури (всі цілочислові) наведено нижче:

- `st_mode` – тип і режим файла (бітова маска прапорців, зокрема прапорець S_IFDIR встановлюють для каталогів);
- `st_nlink` – кількість жорстких зв'язків;
- `st_size` – розмір файла у байтах;
- `st_atime, st_mtime, st_ctime` – час останнього доступу, модифікації та зміни атрибутів (у секундах з 1 січня 1970 року).

Ось приклад відображення інформації про атрибути файла:

```
struct stat attrs;
stat("myfile", &attrs);
if (attrs.st_imode & S_IFDIR)
     printf("myfile є каталогом\n");
else printf("розмір файла: %d\n", attrs.st_size);
```

Для отримання такої самої інформації з дескриптора відкритого файла використовують виклик `fstat()`.

```
int fstat(int fdl, struct stat *attrs);
```

8.2.2. Файлові операції Win32 API

Win32 API містить набір функцій для роботи з файлами, багато в чому аналогічних до системних викликів POSIX. Зупинимося на цьому наборі.

Аналогом системного виклику `open()` у Win32 API є функція `CreateFile()`:

```
HANDLE CreateFile (LPCTSTR fname, DWORD amode, DWORD smode,
   LPSECURITY_ATTRIBUTES attrs, DWORD cmode,
   DWORD flags, HANDLE tfile),
```

Першим параметром є ім'я файла. Параметр `amode` задає режим відкриття файла і може набувати значень GENERIC_READ (читання) і GENERIC_WRITE (записування). Параметр `smode` задає можливість одночасного доступу до файла: 0 означає, що доступ неможливий. Параметр `cmode` може набувати таких значень:

- CREATE_NEW – якщо файл є, повернути помилку, у протилежному випадку створити новий;
- CREATE_ALWAYS – створити новий файл, навіть якщо такий уже є;
- OPEN_EXISTING – якщо файл є, відкрити його, якщо немає, повернути помилку;
- OPEN_ALWAYS – якщо файл є, відкрити його, інакше створити новий.

Під час створення файла значенням параметра `flags` може бути FILE_ATTRIBUTE_NORMAL, що означає створення файла зі стандартними атрибутами.

Ця функція повертає дескриптор відкритого файла. У разі помилки буде повернуто певне значення INVALID_HANDLE_VALUE, рівне -1.

```
// відкриття наявного файла
HANDLE infile = CreateFile("infile.txt", GENERIC_READ,
          0, NULL, OPEN_EXISTING, 0, 0);
// створення нового файла
HANDLE outfile = CreateFile("outfile.txt", GENERIC_WRITE,
```

```
            0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
```

На відміну від POSIX, у Win32 відкриті файлові дескриптори за замовчуванням не успадковуються нащадками цього процесу, навіть якщо під час виклику CreateProcess() параметр `inherit_handles` встановлений у TRUE. Потрібно додатково дозволити таке успадкування для кожного дескриптора при його відкритті. Це забезпечують установкою поля `bInheritHandle` (третього один по одному) структури атрибутів безпеки, покажчик на яку передають у `CreateFile()`. Зауважимо, що першим полем структури є її розмір, а друге описує права доступу для об'єкта.

```
SECURITY_ATTRIBUTES sa={ Sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
HANDLE fh = CreateFile("outfile.txt", GENERIC_WRITE, 0, &sa, ...);
```

Для ***закриття дескриптора*** файла застосовують функцію `CloseHandle()`:

```
CloseHandle(infile);
```

Для ***читання з файла*** використовують функцію `ReadFile()`:

```
BOOL ReadFile( HANDLE fh, LPCVOID buf, DWORD len,
          LPDWORD pbytes_read, LPOVERLAPPED over );
```

Параметр `buf` задає буфер для розміщення прочитаних даних, `len` – кількість байтів, які потрібно прочитати, за адресою `pbytes_read` буде збережена кількість прочитаних байтів (коли під час спроби читання трапився кінець файла, `*pbytes_read` не дорівнюватиме `len`). Виклик `ReadFile()` поверне TRUE у разі успішного завершення читання.

```
char buf[100]; DWORD bytes_read;
ReadFile(outfile, buf, sizeof(buf), &bytes_read, 0);
if (bytes_read != sizeof(buf))
   printf("Досягнуто кінця файла\n");
```

Для ***записування у файл*** використовують функцію `WriteFile()`:

```
BOOL WriteFile( HANDLE fh, LPCVOID buf, DWORD len,
   LPDWORD pbytes_written, LPOVERLAPPED over );
```

Параметр `buf` задає буфер, з якого йтиме записування, `len` – обсяг записуваних даних, за адресою `pbytes_written` буде збережена кількість записаних байтів. Виклик `WriteFile()` поверне TRUE, якщо записування завершене успішно.

```
DWORD bytes_written;
WriteFile(outfile, "hello", sizeof("hello"), &bytes_written, 0);
```

Наведемо приклад реалізації ***копіювання файлів*** за допомогою засобів Win32 API.

```
char buf[1024]; DWORD bytes_read, bytes_written;
HANDLE infile = CreateFile("infile.txt", GENERIC_READ, 0, 0,
   OPEN_EXISTING, 0, 0);
if (infile == INVALID_HANDLE_VALUE) {
   printf("Помилка під час відкриття файла\n"); exit(-1);
}
HANDLE outfile = CreateFile("outfile.txt", GENERIC_WRITE, 0, 0,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
do {
   ReadFile(infile, buf, sizeof(buf), &bytes_read, 0);
   if (bytes_read > 0)
     WriteFile(outfile, buf, bytes_read, &bytes_written, 0);
} while (bytes_read > 0);
CloseHandle(infile);
CloseHandle(outfile);
```

Прямий доступ до файла реалізований функцією SetFilePointer():

```
DWORD SetFilePointer(HANDLE fh, LONG offset,
    PLONG offset_high, DWORD whence);
```

Параметри `offset` і `whence` аналогічні до однойменних параметрів `lseek()`. Константи режиму переміщення визначені як FILE_BEGIN (аналогічно до SEEK_SET), FILE_CURRENT (аналогічно до SEEK_CUR) і FILE_END (аналогічно до SEEK_END).

```
// переміщення покажчика позиції на 100 байт від початку файла
SetFilePointer(outfile, 100, NULL, FILE_BEGIN);
// записування у файл
WriteFile(outfile, "hello", sizeof("hello"), &bytes_written, 0);
```

Створення файла заданої довжини виконують аналогічно до прикладу для системного виклику `lseek()`.

Атрибути файла із заданим ім'ям можуть бути отримані за допомогою функції `GetFileAttributesEx()`. Вона заповнює структуру `WIN32_FILE_ATTRIBUTE_DATA` з полями, аналогічними `stat()`:

- `dwFileAttributes` – маска прапорців (зокрема, для каталогів задають прапорець FILE_ATTRIBUTE_DIRECTORY);
- `ftCreationTime`, `ftLastAccessTlme`, `ftLastWriteTlme` – час створення, доступу і модифікації файла (структури типу FILETIME);
- `nFileSizeLow` – розмір файла (якщо для його відображення не достатньо 4 байт, додатково використовують поле `nFileSizeHigh`).

```
WIN32_FILE_ATTRIBUTE_DATA attrs;
// другий параметр завжди задають однаково
GetFUeAttrlbutesEx("myfile", GetFileExInfoStandard, &attrs);
if (attrs.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
    printf("myfile є каталогом\n");
else printf("розмір файла: %d\n", attrs.nFileSizeLow);
```

Для отримання доступу до атрибутів відкритого файла за його дескриптором використовують функцію GetFileInformationByHandle(). Вона дає змогу отримати повнішу інформацію із файла (зокрема, кількість його жорстких зв'язків).

Є також функції, що повертають значення конкретних атрибутів:

```
long size = FlleSize(fh); //розмір файла за його дескриптором
```

8.2.3. Операції над каталогами

Розглянемо базові операції над каталогами.

- ***Створення нового каталогу.*** Ця операція створює новий каталог. Він звичайно порожній, деякі реалізації автоматично додають у нього елементи «.» і «..».
- ***Вилучення каталогу.*** На рівні системного виклику ця операція дозволена тільки для порожніх каталогів.
- ***Відкриття і закриття каталогу.*** Каталог, подібно до звичайного файла, має бути відкритий перед використанням і закритий після використання. Деякі операції, пов'язані із доступом до елементів, допустимі тільки для відкритих каталогів.
- ***Читання елемента каталогу.*** Ця операція зчитує один елемент каталогу і переміщує поточну позицію на наступний елемент. Використовуючи читання елемента каталогу в циклі, можна обійти весь каталог.
- ***Перехід у початок каталогу.*** Ця операція переміщує поточну позицію до першого

елемента каталогу.

Для ***створення каталогу в POSIX*** використовують виклик `mkdir()`, що приймає як параметр шлях до каталогу і режим.

```
if (mkdir("./newdir", 0644) == -1)
    printf("помилка під час створення каталогу\n");
```

Вилучення **порожнього** каталогу за його іменем відбувається за допомогою виклику `rmdir()`:

```
if (rmdir("./dir") == -1)
    printf("помилка у разі вилучення каталогу\n"):
```

Відкривають каталог викликом `opendir()`, що приймає як параметр ім'я каталогу:

```
DIR *opendir(const char *dirname);
```

Під час виконання `opendir()` ініціалізується внутрішній покажчик поточного елемента каталогу. Цей виклик повертає *дескриптор каталогу* – покажчик на структуру типу DIR, що буде використана під час обходу каталогу. У разі помилки повертають NULL.

Для читання елемента каталогу і переміщення внутрішнього покажчика поточного елемента використовують виклик `readdir()`:

```
struct dirent *readdir(DIR *dirp);
```

Цей виклик повертає покажчик на структуру `dirent`, що описує елемент каталогу (із полем `d_name`, яке містить ім'я елемента) або NULL, якщо елементів більше немає.

Після закінчення пошуку потрібно закрити каталог за допомогою виклику `closedir()`. Якщо необхідно перейти до першого елемента каталогу без його закриття, використовують виклик `rewinddir()`. Обидва ці виклики приймають як параметр дескриптор каталогу.

Наведемо приклад коду обходу каталогу в POSIX.

```
DIR *dirp; struct dirent *dp;
dirp = opendir("./dir");
if (! dirp) {printf("помилка під час відкриття каталогу\n"); exit(-1);}
while (dp = readdir(dirp)) {
    printf ("&s\n", dp->d_name); //відображення імені елемента
}
closedir (dirp);
```

Для ***створення каталогу у Win32 API*** використовують функцію `CreateDirectory()`, що приймає як параметри шлях до каталогу та атрибути безпеки.

```
if (! CreateDirectory("c:\\newdir", 0))
    printf("помилка під час створення каталогу\n");
```

Вилучення порожнього каталогу за його іменем відбувається за допомогою функції `RemoveDirectory()`. Якщо каталог непорожній, ця функція не вилучає його і повертає FALSE.

```
if (! RemoveDirectory("c:\\dir"))
    printf("помилка у разі вилучення каталогу\n");
```

Відкривають каталог функцією `FindFirstFile()`:

```
HANDLE FindFirstFile(LPCSTR path, LPWIN32_FIND_DATA fattrs);
```

Параметр `path` задає набір файлів. Він може бути ім'ям каталогу (в набір входитимуть усі файли цього каталогу), крім того, у ньому допустимі символи шаблону «*» і «?». Параметр `fattrs` – це покажчик на структуру, що буде заповнена інформацією про знайдений файл. Структура подібна до WIN32_FILE_ATTRIBUTE_DATA, але в ній додатково зберігають ім'я файла (поле cFileName).

Ця функція повертає *дескриптор пошуку,* який можна використати для подальшого обходу каталогу. Для доступу до такого файла в каталозі використовують функцію `FindNextFile()`, у яку передають дескриптор пошуку і таку саму структуру, як у `FindFirstFile()`:

```
BOOL FindNextFile( HANDLE findh, LPWIN32_FIND_DATA fattrs);
```

Якщо файлів більше немає, ця функція повертає FALSE.

Після закінчення пошуку потрібно ***закрити дескриптор пошуку*** за допомогою `FindClose().CloseHandle()` для цього використати не можна.

Наведемо приклад коду ***обходу каталогу*** в Win32 API:

```
WIN32_FIND_DATA fattrs;
HANDLE findh = FindFirstFile("c:\\mydir\\*", &fattrs);
do {
    printf ("%s\n", fattrs.cFileName);
    // відображення імені елемента
} while (FindNextFile(findh, &fattrs));
FindClose(findh);
```

8.3. Міжпроцесова взаємодія на основі інтерфейсу файлової системи

Файлові блокування (file locks) є засобом синхронізації процесів, які намагаються здійснити доступ до одного й того самого файла. Процес може заблокувати файл повністю або будь-який його діапазон (аж до одного байта), після чого інші процеси не зможуть отримати доступу до цього файла або діапазону доти, поки з нього не буде зняте блокування.

Розрізняють *консультативне*, або *кооперативне* (advisory lock), і *обов'язкове блокування* (mandatory lock) файлів.

Консультативне блокування є основним, найбезпечнішим видом блокування. Його підтримують на рівні процесів режиму користувача. Для коректної синхронізації всі процеси перед доступом до файла мають перевіряти наявність такого блокування (якщо блокування відсутнє, процес запроваджує своє блокування, виконує дії із файлом і знімає блокування). Якщо процес виконає операцію читання із файла або записування у файл без попередньої перевірки консультативного блокування, система дозволить виконання цього виклику.

Обов'язкове блокування підтримують на рівні ядра. Коли процес запровадив обов'язкове блокування, жодні операції над файлом або його діапазоном не будуть можливими доти, поки це блокування не буде зняте. Насправді таке блокування може бути небезпечним, оскільки навіть користувач із правами адміністратора не може його зняти (так, випадкове блокування системного файла може зробити систему непрацездатною).

Підтримка файлових ***блокувань у POSIX*** ґрунтується на системному виклику `fcntl()`, що дає змогу запровадити або перевірити блокування на файл або на діапазон даних усередині файла.

```
#include <fcntl ,h>
int fcntl(int fdl, int and, struct flock *lock);
```

Значеннями параметра `cmd` може бути F_GETLK – перевірити, чи є блокування, F_SETLK – запровадити блокування, якщо воно вже є, повернути помилку, F_SETLKW – запровадити блокування, перейти до очікування, якщо воно вже є.

Структура `flock` має бути задана перед викликом. У ній можна вказати:

- діапазон байтів у файлі (поля `l_start` і `l_end`); якщо ці поля дорівнюють нулю, блокується весь файл;

- тип блокування (поле `l_type` із можливими значеннями F_RDLCK – для читання, F_WRLCK – для записування, FJJNLCK – зняти блокування).

Якщо `cmd` дорівнює F_GETLK, цю структуру заповнюють усередині виклику, її поле `l_type` міститиме тип блокування, якщо воно є.

```
fdl = open("lockfile", 0_WRONLY|0_CREAT);
// задати структуру flock
struct flock lock = {0}:
// задати блокування для записування
lock.l_type = F_WRLCK;
fcntl (fdl, F_SETLKW, &lock);
// зняти блокування
lock.l_type = F_UNLCK;
fcntl (fdl, F_SETLKW, &lock):
close(fdl);
```

За замовчуванням таке блокування є консультативним, для запровадження обов'язкового блокування потрібно спочатку задати спеціальні права доступу до файла (задати `setgid` біт і очистити дозвіл виконання для групи), після чого застосування `fcntl()` до цього файла спричиняє обов'язкове блокування.

Файлові блокування використовують у UNIX-системах як простий і надійний засіб синхронізації процесів. Зазвичай для цієї мети створюють окремий файл блокування, який блокується процесами у разі необхідності доступу до спільно використовуваних ресурсів і вивільняється разом із цими ресурсами. Наприклад, так можна заборонити повторний запуск уже запущеного застосування.

У ***Windows XP*** (***у Win32***) для запровадження обов'язкових файлових ***блокувань*** використовують функцію `LockFileEx()`, а для консультативних – `LockFile()`. Зупинимось на особливостях використання `LockFileEx()`.

```
BOOL LockFileEx (HANDLE fh, DWORD flags, DWORD dummy,
    DWORD hcount, DWORD hcount, LPOVERLAPPED ov);
```

де: `fh` – дескриптор відкритого файла;

`flags` – прапорці режиму блокування, зокрема LOCKFILE_EXCLUSIVE_LOCK – блокування для записування (якщо це значення не задане, встановлюють блокування для читання), LOCKFILE_FAIL_IMMEDIATELY – повернути нуль негайно, якщо файл уже заблокований;

`lcount` – кількість заблокованих байтів;

`ov` – покажчик на структуру типу OVERLAPPED, поле `Offset` якої визначає зсув заблокованої ділянки від початку файла.

`LockFileEx()` повертає нуль, якщо блокування запровадити не вдалося.

Для розблокування використовують функцію `UnLockFileEx()` з тими самими параметрами, за винятком `flags`.

```
HANDLE fh = CreateFile( "lockfile", GENERIC_WRITE, ... );
OVERLAPPED ov = { 0 };
long size = FileSize(fh);
// задати блокування всього файла для записування
LockFileEx(fh, LOCKFILE_EXCLUSIVE_LOCK, 0, size, 0, &ov);
// зняти блокування
UnLockFileEx(fh, 0, size, 0, &ov);
CloseHandle (fh);
```

Принципи дії відображуваної пам'яті. Інтерфейс відображуваної пам'яті POSIX

Відображення файлів у пам'ять відбувається за допомогою спеціального системного виклику; у POSIX такий виклик визначений як `mmap()`. Він призводить до того, що у визначену частину адресного простору процесу відображають заданий файл або його частину. Після виконання цього системного виклику доступ до такої пам'яті спричинятиме прямий доступ до вмісту цього файла (читання пам'яті аналогічне до читання із файла, її зміна аналогічна до зміни файла). Перед використанням цього виклику файл має бути відкритий.

Системний виклик `mmap()` відповідно до POSIX має такий синтаксис:

```
#include <sys/mman.h>
void *mmap (void *start, size_t len, int prot, int flags,
       int fdl, off_t offset);
```

де: `start` – адреса, з якої почнеться відображувана ділянка (найчастіше цей параметр покладають рівним NULL, надаючи ОС самій визначити, за якою адресою почати відображення);

`len` – розмір відображуваної ділянки у байтах;

`prot` – режим доступу до відображуваної пам'яті (PROT_READ – дозволене читання, PROT_WRITE – дозволене записування);

`flags` – прапорці відображення (MAP_SHARED – усі зміни мають бути негайно записані у відображуваний файл без буферизації, MAP_PRIVATE – під час записування в ділянку буде створено копію відображуваного файла, і подальші зміни відбуватимуться з нею);

`fdl` – дескриптор відображуваного файла;

`offset` – зсув у файлі, із якого почнеться відображення.

Результатом виклику `mmap()` буде адреса, з якої почате відображення.

Відображення в цьому разі працює так. Коли файл відобразився на ділянку пам'яті, що починається з адреси *p,* то:

- у разі доступу до байта пам'яті за адресою *p* буде отримано нульовий (початковий) байт цього файла;
- у разі зміни байта пам'яті за адресою *p+(N* байт) буде змінено *N*-й байт файла (починаючи від 0);
- доступ до пам'яті за кінцем файла неможливий і призводить до помилки.

У разі закриття файла або завершення роботи процесу модифіковану інформацію зберігають у файлі на диску.

Загальний принцип відображення файла в адресний простір процесу показано на рис. 8.1.

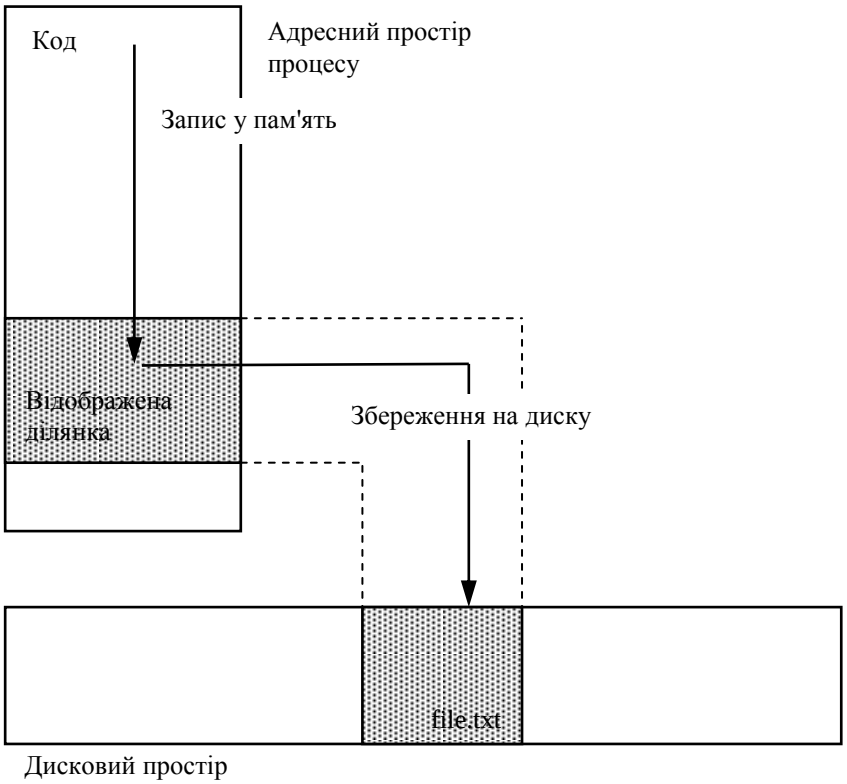

Рис. 8.1. Файл, що відображається у пам’ять

Розрив відображення трапляється у разі використання системного виклику, що у POSIX визначений як `munmap()`, після чого відповідна ділянка пам'яті більше не буде пов'язана із файлом, і спроба доступу до неї спричинить помилку. Розрив відображення теж призводить до збереження інформації у відповідному файлі.

```
int munmap(void *start, size_t len);
```

При завершенні програми необхідності явного виклику `munmap()` немає – записування змін на диск виконуватиметься автоматично.

У разі використання відображуваної пам'яті необхідно виконати такі дії.

1. Відкрити файл із режимом, відповідним до того, як потрібно цей файл використовувати (найкраще – для читання і записування):

```
fdl = open ("myfile", O_RDWR | O_CREAT, 0644):
```

2. Якщо це новий файл, задати його довжину за допомогою `lseek()` і `write()`:

```
lseek(fdl, len, SEEK_SET); write(fdl, "", 1);
```

3. Відобразити файл у пам'ять:

```
int *fmap=(int*)mmap(0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fdl, 0);
```

4. Закрити дескриптор файла:

```
close(fdl);
```

5. Організувати доступ до файла через відображувану пам'ять:

```
fmap[0] = 100; fmap[l] = fmap[0] * 2;
```

6. Якщо відображення більше не потрібне – скасувати його:

```
munmap(fmap, len);
```

Під час роботи із відображуваною пам'яттю застосування може отримувати сигнали SIGSEGV (у разі спроби записати у пам'ять, відкриту тільки для читання) і SIGBUS (у разі спроби доступу до ділянки за межами відображення, наприклад, за кінцем файла).

Особливості інтерфейсу відображуваної пам'яті Win32 API

Як параметр у функцію відображення файла в пам'ять (у Win32 API її називають `MapViewOfFile()`) передають не дескриптор відкритого файла, як у `mmap()`, а дескриптор спеціального *об'єкта відображення,* що створюється заздалегідь за допомогою функції `CreateFileMapping()`. Під час створення об'єкта відображення вказують дескриптор відкритого файла; далі цей об'єкт можна використовувати для відображення кількох регіонів одного файла.

Розглянемо особливості функції `CreateFileMapping()`:

```
HANDLE CreateFileMapping(HANDLE fh, LPSECURITY_ATTRIBUTES psa,
   DWORD prot, DWORD size_high, DWORD size_low, LPCTSTR mapname);
```

де: `hf` – дескриптор файла, відкритого для читання (а можливо, і для записування);

`prot` – режим доступу до пам'яті (PAGE_READWRITE – розподілений доступ для читання і записування, аналогічний до MAP_SHARED, PAGE_WRITECOPY – копіювання під час записування, аналогічне до MAP_PRIVATE);

`size_low`, `size_high` – розмір файла (може дорівнювати нулю, у цьому разі розмір відображення буде рівним поточному розміру файла, `size_high` використовують, коли файл за розміром більший за 232 байти);

`mapname` – ім'я відображення.

`CreateFileMapping()` повертає дескриптор об'єкта відображення. Якщо файл займає менше місця, ніж задано за допомогою параметрів `size_low` і `size_high`, його автоматично змінюють до потрібного розміру без необхідності виклику аналогів `lseek()` і `write()`.

Після створення об'єкта відображення необхідно на його основі задати відображення у пам'ять за допомогою `MapViewOfFile()`:

```
PVOID MapViewOfFile (HANDLE fmap, DWORD prot,
     DWORD off_high, DWORD off_low, SIZE_T len);
```

де: `fmap` – дескриптор об'єкта відображення;

`prot` – режим відображення (FILE_MAP_WRITE – доступ для записування, потребує задавання PAGE_READWRITE для об'єкта відображення, FILE_MAP_COPY – копіювання під час записування, потребує задавання PAGE_WRITECOPY);

`off_low`, `off_high` – зсув відображуваного регіону від початку файла (має бути кратним 64 Кбайт);

`len` – довжина відображуваного регіону у байтах (якщо `len`=0, відображають ділянку до кінця файла).

Ця функція повертає покажчик на початок відображеної ділянки пам'яті. Аналогом `munmap()` є функція `UnmapViewOfFile()`:

```
BOOL UnmapViewOfFile(PVOID start);
```

де `start` – початкова адреса ділянки відображення.

Після використання дескриптор файла і дескриптор об'єкта відображення потрібно закрити за допомогою `CloseHandle()`. Дескриптор файла можна закрити і до виклику `MapViewOfFile()`.

Ґрунтуючись на підтримці відображуваної пам'яті у Win32 API, легко реалізувати ***розподіл пам'яті між процесами***. Це зумовлено низкою можливостей, пов'язаних із наявністю окремих об'єктів відображення.

- Можна задавати об'єкти відображення, які використовують не конкретний дисковий файл, а файл підкачування; для цього достатньо передати у функцію `CreateFile-`

`Mapping()` замість дескриптора файла значення `INVALID_HANDLE_VALUE (-1)`. Це позбавляє програміста необхідності створювати файл тільки для обміну даними з іншим процесом (як потрібно у POSIX).

- Об'єктам відображення під час їхнього створення можна давати імена, після чого вони можуть використовуватися кількома процесами: перший створює об'єкт і задає його ім'я (як параметр `CreateFileMapping()`), а інші відкривають його за допомогою функції `OpenFileMapping()`, передавши як параметр те саме ім'я. Функцію `OpenFileMapping()` використовують для отримання доступу до наявного об'єкта відображення:

```
HANDLE OpenFileMapping(DWORD prot, BOOL isinherit, LPCSTR name);
```

де: `prot` – аналогічний відповідному параметру `MapViewOfFile()`;

`name` – ім'я об'єкта відображення.

Приклад обміну даними між клієнтом і сервером, аналогічний до прикладу для відображуваної пам'яті POSIX, наведено нижче.

```
// сервер
HANDLE mh = CreateFileMapping (INVALID_HANDLE_VALUE, 0,
     PAGE_READWRITE, 0, sizeof(int), "mymap");
int *fmap=(int*)MapViewOfFile(mh, FILE_MAP_WRITE, 0, 0, sizeof(int));
fmap[0] = 1;
for (; ;) {
   printf("%d\n", fmap[0]);
   Sleep(1000);
}
UnmapViewOfFile (fmap);
CloseHandle (mh);
// клієнт
HANDLE mh = OpenFileMapping (FILE_MAP_WRITE, 0, "mymap");
int *fmap=(int*)MapViewOfFile(mh, FILE_MAP_WRITE, 0, 0, sizeof(int));
fmap[0] *= 2;
UnmapViewOfFile (fmap);
CloseHandle (mh);
```

Наведемо ***переваги використання файлів, що відображаються у пам'ять***.

- Робота із такими файлами зводиться до прямого звертання до пам'яті через покажчики, ніби файл був частиною адресного простору процесу. Під час роботи зі звичайними файлами для виконання одних і тих самих дій потрібно відкривати файл, переміщувати покажчик поточної позиції, виконувати читання і записування та закривати файл.
- Робота із відображуваними файлами може бути реалізована ефективніше. Це пов'язано з тим, що:
- для роботи з ними достатньо виконати один системний виклик (`mmap()` або `MapViewOfFile()`), а далі працювати із ділянкою пам'яті в адресному просторі процесу; водночас для відкриття файла, читання, записування тощо потрібно виконувати окремі системні виклики;
- не потрібно копіювати дані між системною пам'яттю і буфером режиму користувача, що необхідно для звичайних операцій читання і записування файла – у разі використання цієї технології файл безпосередньо відображається на адресний простір процесу.

- За допомогою відображуваних файлів можна легко реалізувати розподіл пам'яті між процесами, якщо відобразити один і той самий файл на адресний простір кількох із них.

Головним недоліком цієї технології є те, що відображуваний файл не може бути розширений позиціюванням покажчика поточної позиції за його кінець і виконанням операції записування (через те, що підсистема віртуальної пам'яті у цьому випадку не може визначити точну довжину файла).

8.4. Поіменовані канали

Поіменовані канали POSIX є однобічним засобом передавання даних. Це означає, що якщо один процес записує дані в канал, то інший із цього каналу може тільки читати, причому читання даних відбувається в порядку їхнього записування за принципом FIFO.

Поіменовані канали відображені спеціальними FIFO-файлами у файловій системі UNIX. За іменем такого файла до поіменованого каналу може підключитися будь-який процес у системі, у якого є права читання з цього файла. Після цього всі дані, передані в канал, надходитимуть цьому процесу, поки канал не буде закрито.

Для створення FIFO-файла у POSIX передбачена функція `mkfifo()`:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

Першим параметром `mkfifo()` приймає повний шлях до створюваного файла, другим – режим (аналогічно до першого і третього параметрів `open()`). Ось приклад створення поіменованого каналу:

```
mkfifo ("./myfifo", 0644);
```

Після того як канал (FIFO-файл) створено, будь-який процес може підключитися до нього. Для цього він має спочатку створити з'єднання із каналом за допомогою виклику `open()`. При цьому процес буде заблокований для очікування даних від іншого процесу (у разі читання із каналу очікування триватиме доти, поки інший процес не запише у канал, у разі записування – доки не буде виконане читання). Після закінчення очікування процес може записувати дані в канал за допомогою `write()` або зчитувати їх із каналу за допомогою `read()`:

```
int fdl = open("./myfifo", 0_RDONLY);  // очікування даних
read(fdl, read_buf, sizeof(read_buf)); //дані надійшли - читаємо їх
```

Розглянемо найпростіший підхід до обміну даними через поіменований канал (коли клієнти посилають на сервер повідомлення). Для цього на сервері потрібно виконати такий код:

```
int fdl, bytes_read; char read_buf[100];
mkfifo ("/dir/myfifo", 0644);  // створити файл канал
for (; ;) {           //нескінченний цикл (вихід за сигналом)
    fdl = open("/dir/myfifo", 0_RDONLY);
    bytes_read = read(fdl, read_buf, sizeof(read_buf));
    if (bytes_read > 0) {
      read_buf[bytes_read] = '\0';
      printf("сервер: отримано повідомлення: %s\n". read_buf);
    }
    close(fdl);  // закрити канал
}
```

У разі закриття сервера (наприклад, за сигналом) необхідно закрити канал і вилучити файл каналу за допомогою `unlink()`. Наведемо код клієнта:

```
fdl = open("/dir/myfifo", 0_WRONLY);
                    // відкрити канал для записування
write(fdl, "hello", sizeof("hello"));
                    // записати в нього дані
close(fdl);         // закрити канал
```

На відміну від каналів POSIX, ***поіменовані канали Win32*** реалізують двобічний обмін повідомленнями і не використовують прямо файли на файловій системі. Однак їхнє використання ґрунтується на схожих принципах.

Імена таких каналів створюють за такою схемою: \\ім'я_машини\pipe\ім'я_каналу. Для доступу до каналу на локальному комп'ютері замість імені машини можна використовувати символ «.»: \\.\pipe\ім'я_каналу. Такі імена називають UNC-іменами (Universal Naming Convention – універсальна угода з імен);

Для створення екземпляра каналу на сервері використовують функцію `CreateNamedPipe()`:

```
HANDLE pipe = CreateNaroedPipe(LPCTSTR pname,
      DWORD openmode, DWORD pipemode, DWORD max_conn,
      DWORD obufsize, DWORD ibufsize, DWORD timeout,
      LPSECURITY_ATTRIBUTES psa);
```

де: `pname` – ім'я каналу на локальному комп'ютері (на віддалених комп'ютерах створювати канали не можна);

`openmode` – режим відкриття каналу (наприклад, вмикання прапорця PIPE_ACCESS_DUPLEX означає двобічний зв'язок);

`pipemode` – режим обміну даними із каналом (наприклад, для побайтового читання із каналу потрібно увімкнути прапорець PIPE_READMODE_BYTE, для побайтового записування – PIPE_TYPE_BYTE);

`max_conn` – максимальна кількість клієнтських з'єднань із цим каналом (необмежену кількість задають як PIPE_UNLIMITED_INSTANCES);

`timeout` – максимальний час очікування клієнтом доступу до цього каналу (необмежене очікування задають як NMPWAIT_WAIT_FOREVER).

`CreateNamedPipe()` повертає дескриптор створеного каналу:

```
HANDLE ph=CreateNamedPipe("\\\\.\\pipe\\mypipe", PIPE_ACCESS_DUPLEX,
   PIPE_READMODE_BYTE, 5, 0, 0, NMPWAIT_WAIT_FOREVER, NULL);
```

Для очікування клієнтського з'єднання необхідно виконати функцію `ConnectNamedPipe()`:

```
ConnectNamedPipe(ph, NULL);
```

Як тільки клієнт підключиться до цього каналу, очікування буде завершене, і можна читати з каналу або записувати в канал за допомогою `ReadFile()` і `WriteFile()`. Після завершення обміну даними сервер закриває з'єднання, викликавши функцію `DisconnectNamedPipe()`.

Після завершення роботи із каналом (наприклад, у разі виходу із програми) його дескриптор потрібно закрити за допомогою `CloseHandle()`.

Наведемо код сервера, аналогічний за функціональністю до прикладу для поіменованих каналів POSIX.

```
DWORD bytes read; char buf[100];
HANDLE ph = CreateNamedPipe("\\\\.\\pipe\\mypipe",
```

```
  PIPE_ACCESS_DUPLEX, PIPE_READMODE_BYTE, 1, 0, 0,
  NMPWAIT_WAIT_FOREVER, NULL);
for (; ;) {
ConnectNamedPipe (ph, NULL); //почати прослуховування каналу
// прочитати дані з каналу (від клієнта)
ReadFile(ph, buf, sizeof(buf), &bytes_read, 0);
DisconnectNamedPipe (ph);   // закрити з'єднання
}
CloseHandle(ph); // закрити дескриптор каналу
```

При розробці коду клієнта, необхідно:

- викликати функцію `WaitNamedPipe()`, що очікуватиме виконання сервером `ConnectNamedPipe()`;
- після того, як очікування завершене, встановити з'єднання із каналом за допомогою `CreateFile()`;
- виконати обмін даними за допомогою `ReadFile()` і `WriteFile()`, використовуючи дескриптор з'єднання, повернений `CreateFile()`;
- закрити дескриптор каналу за допомогою `CloseHandle()`:

```
DWORD bytes_written;
WaitNamedPipe("\\\\.\\pipe\\mypipe". NMPWAIT_WAIT_FOREVER);
HANDLE pipeh = CreateFile("\\\\.\\pipe\\mypipe",
     GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
     FILE_ATTRIBUTE_NORMAL, 0);
WriteFile(pipeh, "hello", sizeof("hello"), &bytes_written, 0);
CloseHandle(pipeh);
```

Зауважимо, що у функції `WaitNamedPipe()` і `CreateFile()` можна передавати імена каналів, розміщених на віддалених комп'ютерах.

Фізична організація і характеристики файлових систем

8.5. Розміщення файлів і каталогів на жорсткому диску

Накопичувані на жорстких магнітних дисках (НЖМД) (рис. 8.2, далі – диски) складаються з набору дискових пластин (platters), які покриті магнітним матеріалом і обертаються двигуном із високою швидкістю. Кожній пластині відповідають дві головки (heads), одна зчитує інформацію зверху, інша – знизу. Головки прикріплені до спеціального дискового маніпулятора (disk arm). Маніпулятор може переміщатися по радіусу диска – від центра до зовнішнього краю і назад, таким чином відбувається позиціювання головок.

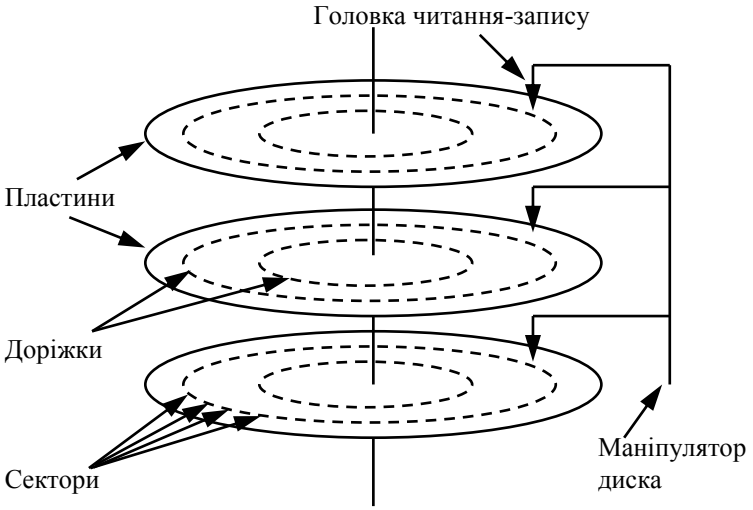

Рис. 8.2. Базовий устрій накопичувача на жорстких магнітних дисках

Головки зчитують інформацію із *доріжок* (tracks), які мають вигляд концентричних кіл. Мінімальна кількість доріжок на поверхні пластини в сучасних дисках – 700, максимальна – більше 20 000. Сукупність усіх доріжок одного радіуса на всіх поверхнях пластин називають *циліндром.*

Кожну доріжку під час низькорівневого форматування розбивають на *сектори* (sectors), обсяг даних сектора для більшості архітектур становить 512 байт (він обов'язково має дорівнювати степені числа 2). Кількість секторів для всіх доріжок однакова (у діапазоні від 16 до 1600).

Основними ***характеристиками доступу до диска*** є:

- *час пошуку* (seek time) – час переміщення маніпулятора для позиціювання головки на потрібній доріжці (у середньому становить від 10 до 20 мс);
- *ротаційна затримка* (rotational delay) – час очікування, поки пластина повернеться так, що потрібний сектор опиниться під доріжкою (у середньому становить 8 мс);
- *пропускна здатність передавання даних* (transfer bandwidth) – обсяг даних, що передаються від пристрою в пам'ять за одиницю часу; для сучасних дисків ця характеристика порівнянна із пропускною здатністю оперативної пам'яті (200 Мбайт/с), час передавання одного сектора вимірюють у наносекундах.

Час, необхідний для читання сектора, одержують додаванням часу пошуку, ротаційної затримки і часу передавання (при цьому час передавання можна вважати дуже малим). Очевидно, що час читання одного сектора практично не відрізняється від часу читання кількох розташованих поряд секторів, а час читання цілої доріжки за одну операцію буде менший, ніж час читання одного сектора через відсутність ротаційної затримки.

Швидкість доступу до диска, порівняно із доступом до пам'яті, надзвичайно мала, зараз диски є основним «вузьким місцем» з погляду продуктивності комп'ютерної системи. При цьому реальне поліпшення останніми роками наявне лише для пропускної

здатності передавання даних. Час пошуку і ротаційна затримка майже не змінюються, тому що вони пов'язані з керуванням механічними пристроями (дисковим маніпулятором і двигуном, що обертає пластини) і обмежені їхніми фізичними характеристиками.

У результаті час читання великих обсягів неперервних даних усе менше відрізняється від часу читання малих. Як наслідок, першорядне значення для розробників файлових систем набуває розв'язання двох задач:

- організації даних таким чином, щоб ті з них, які будуть потрібні одночасно,
- перебували на диску поруч (і їх можна було зчитати за одну операцію);
- підвищення якості кешування даних (оскільки пам'яті стає все більше, зростає ймовірність того, що всі потрібні дані міститимуться в кеші, і доступ до диска стане взагалі не потрібний).

Файлова система звичайно будує базове відображення даних поверх того, яке їй надають драйвери дискових пристроїв. Насамперед, ОС розподіляє дисковий простір не секторами, а спеціальними одиницями розміщення – *кластерами* (clusters) або *дисковими блоками* (disk blocks, термін «дисковий блок» більш розповсюджений в UNIX-системах). Визначення розміру кластера і розміщення інформації, необхідної для функціонування файлової системи, відбувається під час високорівневого форматування розділу. Саме таке форматування створює файлову систему в розділі.

Розмір кластера визначає особливості розподілу дискового простору в системі. Використання кластерів великого розміру може спричинити значну внутрішню фрагментацію через файли, які за розміром менші, ніж кластер.

Деякі застосування (насамперед, сервери баз даних) можуть реалізовувати свою власну фізичну організацію даних на диску. Для них файлова система може виявитися зайвим рівнем доступу, що тільки сповільнюватиме роботу. Багато ОС надають таким застосуванням можливість працювати із розділами, поданими у вигляді простого набору дискових секторів, який не містить структур даних файлової системи. Про такі розділи кажуть, що вони містять неорганізовану файлову систему (raw file system). Для них не виконують операцію високорівневого форматування.

Перед тим як перейти до розгляду особливостей фізичної організації файлової системи в рамках розділу, коротко ознайомимося з організацією розділів на диску. Початковий (нульовий) сектор диска називають *головним завантажувальним записом* (Master Boot Record, MBR). Наприкінці цього запису міститься *таблиця розділів* цього диска, де для кожного розділу зберігається початкова і кінцева адреси.

Один із розділів диска може бути позначений як *завантажувальний* (bootable) або *активний* (active). Після завантаження комп'ютера апаратне забезпечення звертається до MBR одного з дисків, визначає з його таблиці розділів завантажувальний розділ і намагається знайти в першому кластері цього розділу спеціальну невелику програму – *завантажувач ОС* (OS boot loader). Саме завантажувач ОС відповідає за пошук на диску і початкове завантаження у пам'ять ядра операційної системи.

Усередині розділу розташовані структури даних файлової системи.

З погляду користувача файл із заданим іменем (далі вважатимемо, що інформацію про шлях включено в ім'я) – це неструктурована послідовність байтів, а з погляду фізичної структури файлової системи, файл – це набір дискових блоків, що містять його дані. Завдання файлової системи полягає у забезпеченні перетворення сукупності імені файла і логічного зсуву в ньому на фізичну адресу всередині відповідного дискового блоку.

Необхідність такого перетворення визначає основне завдання файлової системи – відстежувати розміщення вмісту файлів на диску. Інформація про розміщення даних

файла на диску зберігається у структурі даних, що називають *заголовком файла.* Такі заголовки звичайно зберігають на диску разом із файлами. Під час розробки структури даних для такого заголовка потрібно враховувати, що більшість файлів мають малий розмір, а основну частину дискового простору розподіляють, навпаки, під файли великого розміру, із якими переважно і виконують операції введення-виведення.

Оскільки продуктивність файлової системи залежить від кількості операцій доступу до диска, важливо максимально її обмежити. Кілька сотень таких операцій можуть додатково зайняти кілька секунд часу. На практиці слід враховувати, що всі імена файлів (і самі файли) каталогу і всі блоки у файлі зазвичай використовують разом, послідовно.

Принципи, що лежать в основі фізичної організації файлової системи, визначають різні способи розміщення файлів на диску. Крім обліку розміщення даних, фізичне розміщення потребує також обліку вільних кластерів.

Найпростіший ***підхід до фізичної організації файлових систем*** – це неперервне розміщення файлів. При цьому кожному файлові відповідає набір неперервно розташованих кластерів на диску (рис. 8.3). Для кожного файла мають зберігатися адреса початкового кластера і розмір файла.

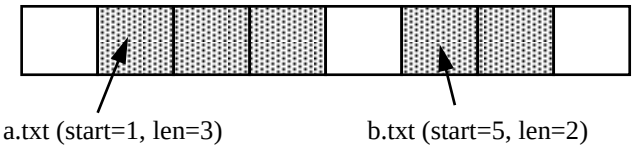

Рис. 8.3. Неперервне розміщення файлів

Зазначимо, що розподіл дискового простору в цьому разі подібний до динамічного розподілу пам'яті. Для пошуку вільного блоку на диску можна використати алгоритми першого підходящого або найкращого підходящого блоку.

Неперервне розміщення файлів вирізняється простотою в реалізації та ефективністю (наприклад, весь файл може бути зчитаний за одну операцію), але має істотні недоліки.

- Під час створення файла користувач має заздалегідь задати його максимальну довжину і виділити весь простір на диску за один раз. Збільшувати розміри файлів під час роботи не можна. У багатьох ситуаціях це абсолютно неприйнятне (наприклад, неможливо вимагати від користувача текстового редактора щоб він вказував остаточну довжину файла перед його редагуванням).
- Вилучення файлів згодом може спричинити велику зовнішню фрагментацію дискового простору з тих самих причин, що й за динамічного розподілу пам'яті. У сучасних ОС для організації даних на жорстких дисках неперервне розміщення майже не використовують, проте його застосовують у таких файлових системах, де можна заздалегідь передбачити, якого розміру буде файл. Прикладом є файлові системи для компакт-дисків. Вони мають кілька властивостей, що роблять неперервне розміщення файлів найкращим рішенням:
- записування такої файлової системи здійснюють повністю за один раз, під час записування для кожного файла заздалегідь відомий його розмір;
- доступ до файлових систем на компакт-диску здійснюють лише для читання, файли в них ніколи не розширюють і не вилучають, тому відсутні причини появи зовнішньої фрагментації.

Іншим підходом є організація кластерів, що належать файлу, у ***зв'язний список.*** Кожен кластер файла містить інформацію про те, де перебуває наступний кластер цього файла (наприклад, його номер). Найпростіший приклад такого розміщення бачимо на рис. 8.4. Заголовок файла в цьому разі має містити посилання на його перший кластер, вільні кластери можуть бути організовані в аналогічний список.

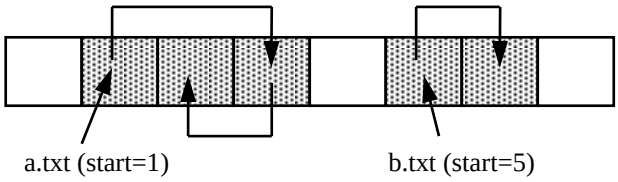

Рис. 8.4. Найпростіший приклад зв’язаного розміщення файлів

Розміщення файлів з використанням зв'язних списків надає такі ***переваги***:

- відсутність зовнішньої фрагментації (є тільки невелика внутрішня фрагментація, пов'язана з тим, що розмір файла може не ділитися націло на розмір кластера);
- мінімум інформації, яка потрібна для зберігання у заголовку файла (тільки посилання на перший кластер);
- можливість динамічної зміни розміру файла;
- простота реалізації керування вільними блоками, яке принципово не відрізняється від керування розміщенням файлів.

Цей підхід, однак, не позбавлений і серйозних ***недоліків***:

- відсутність ефективної реалізації випадкового доступу до файла: для того щоб одержати доступ до кластера з номером *n*, потрібно прочитати всі кластери файла з номерами від 1 до *n*-1;
- зниження продуктивності тих застосувань, які зчитують дані блоками, за розміром рівними степеню числа 2 (а таких застосувань досить багато): частина будь-якого кластера повинна містити номер наступного, тому корисна інформація в кластері займає обсяг, не кратний його розміру (цей обсяг навіть не є степенем числа 2);
- можливість втрати інформації у послідовності кластерів: якщо внаслідок збою буде втрачено кластер на початку файла, вся інформація в кластерах, що йдуть за ним, також буде втрачена. Є модифікації цієї схеми, які зберегли своє значення дотепер, ***найважливішою з них є використання таблиці розміщення файлів***.

Цей підхід (рис. 8.5) полягає в тому, що всі посилання, які формують списки кластерів файла, зберігаються в окремій ділянці файлової системи фіксованого розміру, формуючи *таблицю розміщення файлів* (File Allocation Table, FAT). Елемент такої таблиці відповідає кластеру на диску і може містити:

- номер наступного кластера, якщо цей кластер належить файлу і не є його останнім кластером;
- індикатор кінця файла, якщо цей кластер є останнім кластером файла;
- індикатор, який показує, що цей кластер вільний.

Для організації файла достатньо помістити у відповідний йому елемент каталога номер першого кластера файла. За необхідності прочитати файл система знаходить за цим номером кластера відповідний елемент FAT, зчитує із нього інформацію про наступний кластер і т. д. Цей процес триває доти, поки не трапиться індикатор кінця файла.

Використання цього підходу дає змогу підвищити ефективність і надійність розміщення файлів зв'язними списками. Це досягається завдяки тому, що розміри FAT дозволяють кешувати її в пам'яті. Через це доступ до диска під час відстеження посилань заміняють зверненнями до оперативної пам'яті. Зазначимо, що навіть якщо таке кешування не реалізоване, випадковий доступ до файла не призводитиме до читання всіх попередніх його кластерів – зчитані будуть тільки попередні елементи FAT.

Крім того, спрощується захист від збоїв. Для цього, наприклад, можна зберігати на диску додаткову копію FAT, що автоматично синхронізуватиметься з основною. У разі ушкодження однієї з копій інформація може бути відновлена з іншої.

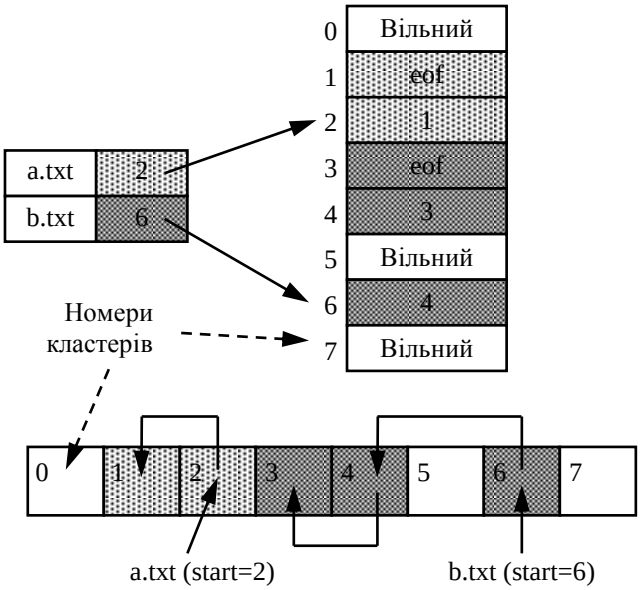

Рис. 8.5. Використання таблиці розміщення файлів

І нарешті, службову інформацію більше не зберігають безпосередньо у кластерах файла, вивільняючи в них місце для даних. Тепер обсяг корисних даних всередині кластера майже завжди (за винятком, можливо, останнього кластера файла) дорівнюватиме степеню числа 2.

Однак, у разі такого способу розміщення файлів для розділів великого розміру обсяг FAT може стати доволі великим і її кешування може потребувати значних витрат пам'яті. Скоротити розмір таблиці можна, збільшивши розмір кластера, але це, в свою чергу, призводить до збільшення внутрішньої фрагментації для малих файлів.

Також руйнування обох копій FAT (внаслідок апаратного збою або дії програми-зловмисника, наприклад, комп'ютерного вірусу) робить відновлення даних дуже складною задачею, яку не завжди можна розв'язати.

Базовою ідеєю ще одного підходу до розміщення файлів ***є перелік адрес всіх кластерів файла в його заголовку***. Такий заголовок файла дістав назву індексного дескриптора, або і-вузла (inode), а сам підхід – індексованого розміщення файлів.

За індексованого розміщення із кожним файлом пов'язують його індексний дескриптор. Він містить масив із адресами (або номерами) усіх кластерів цього файла, при цьому *n-й* елемент масиву відповідає *n-му* кластеру. Індексні дескриптори зберігають окремо від даних файла, для цього звичайно виділяють на початку розділу спеціальну ділянку індексних дескрипторів. В елементі каталогу розміщують номер індексного дескриптора відповідного файла (рис. 8.6).

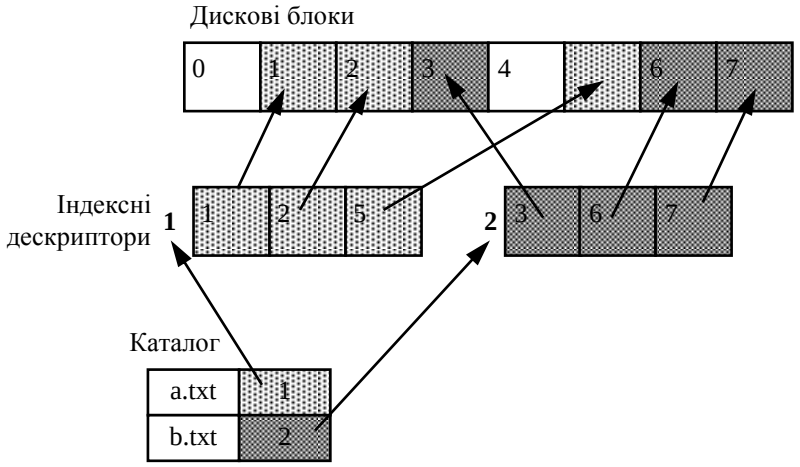

Рис. 8.6. Індексоване розміщення файлів

Під час створення файла на диску розміщують його індексний дескриптор, у якому всі покажчики на кластери спочатку є порожніми. Під час першого записування в *n-й* кластер файла менеджер вільного простору виділяє вільний кластер і його номер або адресу заносять у відповідний елемент масиву.

Цей підхід стійкий до зовнішньої фрагментації й ефективно підтримує як послідовний, так і випадковий доступ (інформація про всі кластери зберігається компактно і може бути зчитана за одну операцію). Для підвищення ефективності індексний дескриптор повністю завантажують у пам'ять, коли процес починає працювати з файлом, і залишають у пам'яті доти, поки ця робота триває.

Основною проблемою є підбір розміру і задавання оптимальної структури індексного дескриптора, оскільки:

- з одного боку, зменшення розміру дескриптора може значно зекономити дисковий простір і пам'ять (дескрипторів потрібно створювати значну кількість – по одному на кожний файл, разом вони можуть займати досить багато місця на диску; крім того, для кожного відкритого файла дескриптор буде розташовано в оперативній пам'яті).
- з іншого боку, дескриптора надто малого розміру може не вистачити для розміщення інформації про всі кластери великого файла.

Одне з компромісних розв'язань цієї задачі, яке застосовується вже багато років у UNIX-системах, зображене на рис. 8.7. Під час його опису замість терміна «кластер» вживатимемо його синонім «дисковий блок».

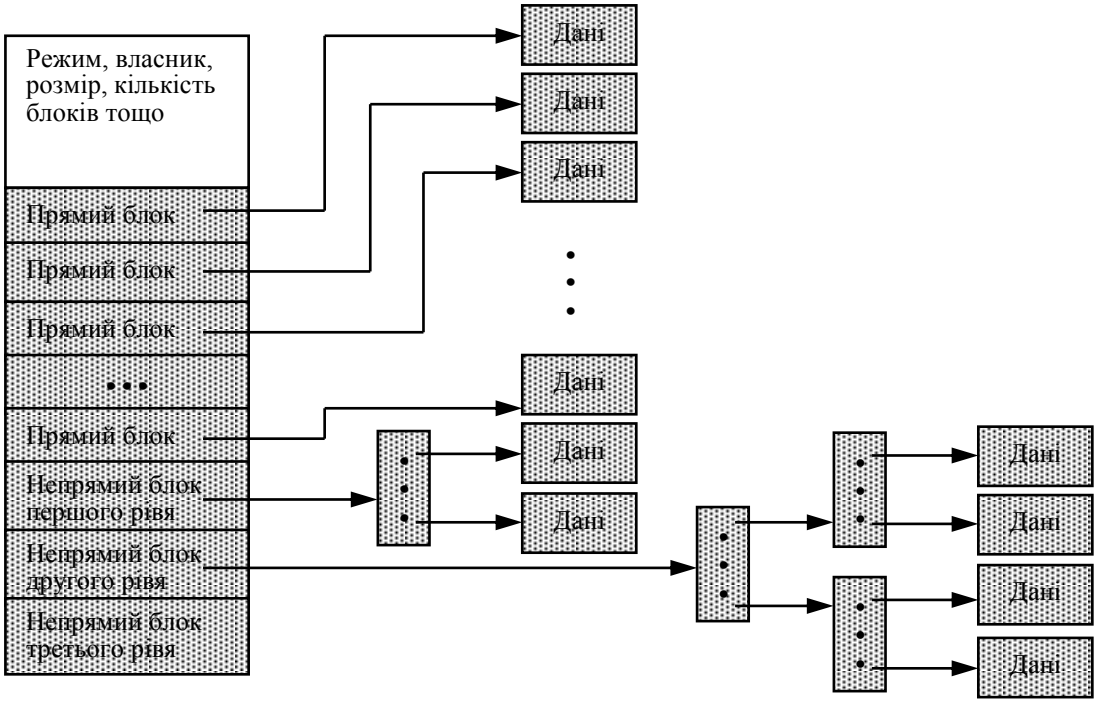

Рис. 8.7. структура індексного дескриптора в UNIX

У цьому разі індексний дескриптор містить елементи різного призначення.

- Частина елементів (зазвичай перші 12) безпосередньо вказує на дискові блоки, які називають *прямими* (direct blocks). Отже, якщо файл може вміститися у 12 дискових блоках (за розміру блоку 4 Кбайт максимальний розмір такого файла становитиме 4096 x 12 = 49 152 байти), усі ці блоки будуть прямо адресовані його індексним дескриптором і жодних додаткових структур даних не буде потрібно.
- Якщо файлу необхідно для розміщення даних більше, ніж 12 дискових блоків, використовують непряму адресацію першого рівня. У цьому разі 13-й елемент

індексного дескриптора вказує не на блок із даними, а на спеціальний непрямий блок першого рівня (single indirect block). Він містить масив адрес наступних блоків файла (за розміру блоку 4 Кбайт, а адреси – 4 байти в ньому міститимуться адреси 1024 блоків, при цьому максимальний розмір файла буде 4096 x (12 + 1024) = 4 234 456 байт).

- Якщо файлу потрібно для розміщення більше ніж 1024 + 12 = 1036 дискових блоків, використовують непряму адресацію другого рівня. 14-й елемент індексного дескриптора в цьому разі вказуватиме на непрямий блок другого рівня (double indirect block). Такий блок містить масив з 1024 адрес непрямих блоків першого рівня, кожен із них, як зазначалося, містить масив адрес дискових блоків файла. Тому за допомогою такого блоку можна адресувати 1024^2 додаткових блоків.
- Нарешті, якщо файлу потрібно більше ніж 1036 + 1024^2 дискових блоків, використовують непряму адресацію третього рівня. Останній 15-й елемент індексного дескриптора вказуватиме на непрямий блок третього рівня (triple indirect block), що містить масив з 1024 адрес непрямих блоків другого рівня, даючи змогу адресувати додатково 1024^3 дискових блоків.

Розмір блоку може відрізнятися від 4 Кбайт. Чим більший блок, тим більшим є розмір, що може бути досягнутий файлом, поки не виникне необхідності у непрямій адресації вищого рівня. З іншого боку, більший розмір блоку спричиняє більшу внутрішню фрагментацію.

Можна розподіляти дисковий простір не блоками (кластерами), а їхніми групами (неперервними ділянками із кількох дискових блоків). Такі групи ще називають *екстентами* (extents). Кожен екстент характеризується довжиною (у блоках) і номером початкового дискового блоку. Коли виникає необхідність виділити кілька неперервно розташованих блоків одночасно, замість цього виділяють лише один екстент потрібної довжини. У результаті обсяг службової інформації, яку потрібно зберігати, може бути скорочений.

Характер перетворення адреси в номер кластера робить цей підхід аналогом сторінкової організації пам'яті, причому індексний дескриптор відповідає таблиці сторінок.

Багато операційних систем не зберігають покажчики на дискові блоки файлів у їхніх індексних дескрипторах, поки до них не було доступу для записування. Фрагменти, до яких цього доступу не було з моменту створення файла, називають *«дірами»* (holes), дисковий простір під них не виділяють, але під час розрахунку довжини файла їх враховують. У разі читання вмісту «діри» повертають блоки, заповнені нулями, звертання до диска не відбувається.

На практиці «діри» найчастіше виникають, коли покажчик поточної позиції файла перемішують далеко за його кінець, після чого виконують операцію записування. У результаті розмір файла збільшується без додаткового виділення дискового простору. Подібні файли називають ***розрідженими файлами*** (sparse files). Вони реально займають на диску місця набагато менше, ніж їхня довжина, фактично довжина розрідженого файла може перевищувати розмір розділу, на якому він перебуває.

Каталоги звичайно організують як спеціальні файли, що містять набір елементів каталогу (directory entries), кожен з яких відповідає одному файлові.

Елемент каталогу обов'язково містить ім'я файла та інформацію, що дає змогу за іменем файла знайти на диску адреси його кластерів. Структуру такої інформації визначають підходи до розміщення файлів: для неперервного розміщення в елементі

каталогу зберігатиметься адреса початкового кластера і довжина файла, для розміщення зв'язними списками – тільки адреса або номер початкового кластера, для індексованого розміщення достатньо зберігати номер індексного дескриптора файла.

Крім обов'язкових даних, елемент каталогу може зберігати додаткову інформацію, характер якої залежить від реалізації. Це може бути, наприклад, набір атрибутів файла (так найчастіше роблять при неперервному розміщенні або розміщенні зв'язаними списками). З іншого боку, за індексованого розміщення всі атрибути файла та іншу службову інформацію зберігають в індексному дескрипторі, а в елемент каталогу додаткову інформацію не заносять (там є тільки ім'я файла і номер дескриптора).

Реалізація каталогу включає також організацію списку його елементів. Найчастіше елементи об'єднують у лінійний список, але якщо очікують, що в каталогах буде багато елементів, для підвищення ефективності пошуку файла можна використати складніші структури даних, такі як бінарне дерево пошуку або хеш-таблиця. Для прискорення пошуку можна також кешувати елементи каталогу, при цьому під час кожного пошуку файла спочатку перевіряється його наявність у кеші, у разі влучення пошук буде зроблено дуже швидко.

Розглянемо, яким чином у каталозі зберігають довгі імена файлів. Є ряд підходів до вирішення цієї проблеми.

- Найпростіше зарезервувати простір у кожному елементі каталогу для максимально допустимої кількості символів у імені. Такий підхід можна використати, якщо максимальна кількість символів невелика, у протилежному випадку місце на диску витрачатиметься даремно, оскільки більша частина імен не займатиме весь зарезервований простір.
- Можна зберігати довгі імена в елементах каталогу повністю, у цьому випадку довжина такого елемента не буде фіксованою. Перед кожним елементом каталогу зберігають його довжину, а кінець імені файла позначають спеціальним (зазвичай нульовим) символом. Недоліки цього підходу пов'язані з тим, що через різну довжину елементів виникає зовнішня фрагментація і каталог надто великого розміру може зайняти кілька сторінок у пам'яті, тому під час перегляду такого каталогу є ризик виникнення сторінкових переривань.
- Нарешті, можна зробити всі елементи каталогу однієї довжини, при цьому кожен із них міститиме покажчик на довге ім'я. Усі довгі імена зберігатимуться окремо (наприклад, наприкінці каталогу). Це вирішує проблему зовнішньої фрагментації для елементів каталогу, але питання про керування ділянкою зберігання довгих імен залишається відкритим.

Поряд з урахуванням кластерів, виділених для розміщення даних файла, файлові системи мають ***вести облік вільних кластерів***. Це насамперед необхідно для того щоб розв'язати задачу виділення нових кластерів для даних. Для організації керування вільним дисковим простором найчастіше використовують два підходи.

- *Бітовий масив* (бітова карта кластерів), у якій кожен біт відповідає одному кластеру на диску. Якщо відповідний кластер вільний, біт дорівнює одиниці, якщо зайнятий – нулю. Головна перевага такого підходу полягає в тому, що пошук першого ненульового біта можна легко реалізувати, спираючись на апаратну підтримку.
- *Зв'язний список вільних кластерів.* Такий підхід, як зазначалося, найзручніше використати, коли зв'язні списки використовують і для організації розміщення файлів. Звичайно в цьому разі організують список, елементами якого є кластери з адресами (номерами) вільних кластерів на диску.

Достатньо зберігати в пам'яті один елемент списку вільних кластерів або один кластер із бітовою картою. Коли вільні блоки в ньому закінчаться, зчитують наступний елемент. У разі вилучення файла номери його кластерів додають у поточний елемент списку або в поточну бітову карту. Коли місця там більше немає, поточний елемент (карту) записують на диск, а в пам'яті створюють новий елемент або нову карту, куди заносять номери кластерів, для яких забракло місця.

8.6. Продуктивність файлових систем

Розглянемо різні підходи, які використовують для ***підвищення продуктивності*** файлових систем. Є дві категорії таких підходів.

- До першої належать різні підходи, які можуть бути впроваджені на етапі проектування і розробки файлової системи: керування розміром блоку, оптимізація розміщення даних, застосування продуктивніших алгоритмів і структур даних.
- До другої належать низькорівневі універсальні підходи, прозорі для файлової системи (їхнє впровадження може підвищити продуктивність файлової системи без її модифікації). До них належать реалізація дискового кеша і планування переміщення головок диска.

Розглянемо, яким чином можна оптимізувати продуктивність файлової системи зміною структур даних і алгоритмів, які в ній застосовують. У викладі використовуватимемо класичний приклад оптимізації традиційної файлової системи вихідної версії UNIX під час розроблення системи Fast File System (FFS) для BSD UNIX (у наш час ця файлова система також відома як ufs).

Традиційна файлова система UNIX складається із суперблока (що містить номери блоків файлової системи, поточну кількість файлів, покажчик на список вільних блоків), ділянки індексних дескрипторів і блоків даних (рис. 8.8). Розмір блока фіксований і становить 512 байт. Вільні блоки об'єднані у список.

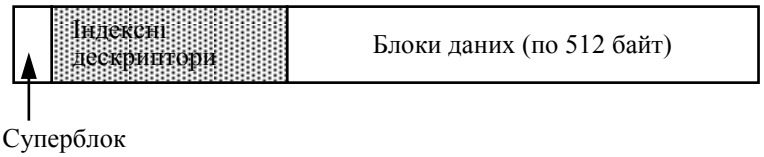

Рис. 8.8. Традиційна файлова система UNIX

Така система є прикладом простого і витонченого вирішення, яке виявилось неприйнятним із погляду продуктивності. На практиці ця файлова система могла досягти на пересиланні даних пропускної здатності, що становить усього 2 % можливостей диска. Назвемо деякі причини такої низької продуктивності.

- Розмір дискового блоку виявився недостатнім, внаслідок чого для розміщення даних файла була потрібна велика кількість блоків; індексні дескриптори навіть для невеликих файлів потребували кількох рівнів непрямої адресації, перехід між якими сповільнював доступ; пересилання даних одним блоком призводила до зниження пропускної здатності.
- Пов'язані об'єкти часто виявлялися віддаленими один від одного і не могли бути зчитані разом, зокрема, індексні дескриптори були розташовані далеко від блоків даних і для каталогу не перебували разом; послідовні блоки для файла також не містилися разом (це траплялося тому, що протягом експлуатації системи через вилучення файлів список вільних блоків ставав «розкиданим» по диску, внаслідок чого файли під час створення отримували блоки, віддалені один від одного).

До розв'язання цих проблем під час розробки файлової системи FFS були запропоновані декілька підходів.

Насамперед, у цій системі було збільшено розмір дискового блока (у FFS звичайно використовували два розміри блока: 4 і 8 Кбайт). Для того щоб уникнути внутрішньої фрагментації (яка завжди зростає зі збільшенням розміру блока), було запропоновано в разі необхідності розбивати невикористані блоки на частини меншого розміру – фрагменти, які можна використати для розміщення невеликих файлів. Мінімальний розмір фрагмента дорівнює розміру сектора диска, звичайно було використано фрагменти на 1 Кбайт.

Крім того, велику увагу було приділено групуванню взаємозалежних даних. З огляду на те, що найбільші втрати часу трапляються під час переміщення головки, було запропоновано розміщувати такі дані в рамках групи циліндрів, яка об'єднує один або кілька суміжних циліндрів. Під час доступу до даних однієї такої групи головку переміщувати було не потрібно, або її переміщення виявлялося мінімальним. Кожна така група за своєю структурою повторювала файлову систему: у ній був суперблок, ділянка індексних дескрипторів і ділянка дискових блоків, виділених для файлів. Тому індексний дескриптор кожного файла розміщувався в тому самому циліндрі, що і його дані, в одній групі циліндрів розміщувалися також всі індексні дескриптори одного каталогу. Послідовні блоки файла прагнули розміщувати в суміжних секторах.

Нарешті, ще одна важлива зміна була зроблена у форматі зберігання інформації про вільні блоки – список вільних блоків було замінено бітовою картою, яка могла бути повністю завантажена у пам'ять. Пошук суміжних блоків у такій карті міг бути реалізований ефективніше. Для ще більшої ефективності цього процесу в системі постійно підтримували деякий вільний простір на диску (коли є вільні дискові блоки, ймовірність знайти суміжні блоки зростає).

Внаслідок реалізації цих і деяких інших рішень пропускна здатність файлової системи зросла в 10-20 разів (до 40 % можливостей диска).

На підставі цього прикладу можна зробити такі висновки:

- розмір блоку впливає на продуктивність файлової системи, при цьому потрібно враховувати можливість внутрішньої фрагментації;
- програмні зусилля, витрачені на скорочення часу пошуку і ротаційної затримки, окупаються (насамперед вони мають спрямовуватися на забезпечення суміжного розміщення взаємозалежної інформації);
- використання бітової карти вільних блоків теж спричиняє підвищення продуктивності.

Ідеї, що лежать в основі FFS, вплинули на особливості проектування файлової системи ext2fs – основної файлової системи Linux.

Найважливішим засобом підвищення продуктивності файлових систем є організація ***дискового кеша*** (disk cache). Дисковим кешем називають спеціальну ділянку в основній пам'яті, яку використовують для кешування дискових блоків. У разі спроби доступу до дискового блока його поміщають в кеш, розраховуючи на те, що він незабаром буде використаний знову. Під час наступного повторного використання можна обійтися без звертання до диска.

Керування дисковим кешем відбувається на нижчому рівні порівняно з реалізацією файлової системи. Кеш має бути прозорий для файлової системи: блоки, що перебувають у ньому з погляду файлової системи розташовані на диску, відмінності є тільки у швидкості доступу.

Загальний принцип організації дискового кеша показано на рис. 8.9. Для прискорення пошуку потрібного блоку звичайно використовують хеш-функцію, що переводить номер блока у хеш-код. У пам'яті зберігають хеш-таблицю, елементи якої відповідають окремим значенням хеш-коду. Усі блоки з однаковим значенням хеш-коду об'єднують у зв'язні списки. Для пошуку в кеші блока із конкретним номером досить обчислити значення хеш-функції та переглянути відповідний список.

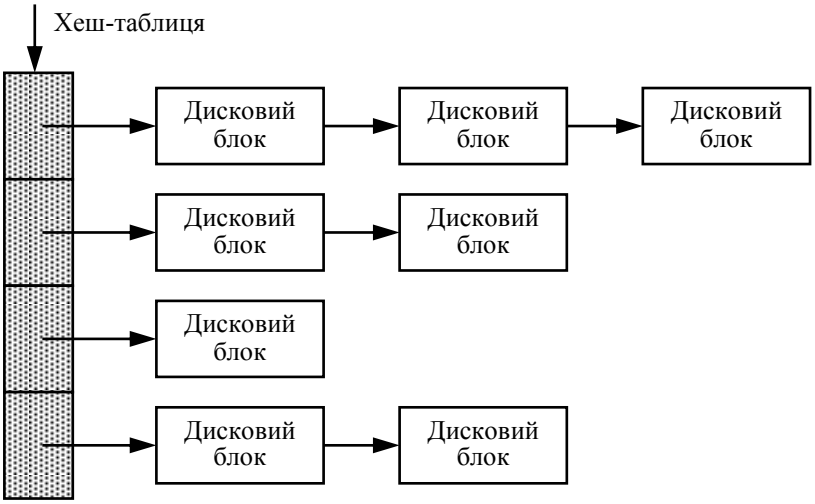

Рис. 8.9. Дисковий кеш

Під час реалізації керування дисковим кешем необхідно отримати відповіді на такі запитання.

- Якого розміру має бути кеш?
- Як має бути організоване заміщення блоків у кеші?
- Як організувати збереження модифікованої інформації із кеша на диск?
- Яким чином оптимізувати завантаження блоків у кеш?

Визначаючи ***розмір дискового кеша***, важливо пам'ятати, що він разом із менеджером віртуальної пам'яті з погляду використання основної пам'яті є двома головними підсистемами, що конкурують між собою.

Є два різновиди кеша з погляду керування його розміром: *кеш фіксованого розміру* і *кеш змінного розміру.*

Для кеша фіксованого розміру межі дискового і сторінкового кешів задають жорстко і змінюватися під час виконання вони не можуть. Недоліком такого кеша є те, що він не може підлаштовуватися під зміну навантаження в системі (наприклад, коли відкрито багато файлів, може знадобитися розширити дисковий кеш, коли завантажено багато процесів – сторінковий).

Стандартна реалізація кеша змінного розміру виділяє спільну пам'ять і під сторінковий, і під дисковий кеш. Під час виконання кожна із підсистем працює з цією пам'яттю, розміщуючи там і сторінки, і дискові блоки (зручно, якщо розмір сторінки дорівнює розміру блока). Недоліком такої реалізації є те, що один файл великого розміру, відкритий процесом, може зайняти блоки, призначені не тільки для відкритих файлів інших процесів, але й для сторінкового кеша віртуальної пам'яті. Для вирішення цієї проблеми можна перейти до змішаного керування кешем, коли межі задають, але їх можна змінити під час виконання.

Організація ***заміщення блоків у дисковому кеші*** багато в чому подібна до реалізації заміщення віртуальної пам'яті. Фактично для визначення заміщуваного блока можна використати більшість алгоритмів заміщення сторінок, такі як FIFO, LRU, годинниковий алгоритм. Зазначимо, що в цьому разі LRU-алгоритм реалізувати простіше, оскільки

звертання до дискового кеша відбувається значно рідше, ніж звертання до пам'яті, і за тривалістю інтервал між такими звертаннями значно перевищує той час, який потрібно затратити на фіксування моменту звертання. Для реалізації LRU-алгоритму можна підтримувати список усіх блоків кеша, упорядкований за часом використання (найстаріший блок – на початку, найновіший – наприкінці), переміщуючи блоки у кінець списку під час звертання до них.

Необхідно враховувати, що за такої ситуації LRU-алгоритм не завжди є кращим або навіть просто прийнятним вирішенням. Наприклад, якщо деякий критично важливий блок не буде записаний на диск після його модифікації, подальший збій може бути фатальним для всієї файлової системи. У той же час у разі використання LRU-алгоритму цей блок буде поміщено в кінець списку, і його записування відкладеться на якийсь час.

Не рекомендують також використовувати алгоритм LRU у численних ситуаціях, коли розмір файла, який зчитують послідовно, перевищує розмір блока. У цьому разі найоптимальнішою є саме зворотна стратегія – MRU (Most Recently Used), коли витісняють блок, що був використаний останнім.

Розглянемо ***приклад***. Нехай файл містить 4 дискові блоки, а в кеші є місце для трьох. Файл зчитують послідовно, при цьому отримують рядок посилань 1234. Якщо використати LRU-алгоритм, то при зчитуванні блоку 4 він заміщує в пам'яті блок 1. Якщо тепер прочитати файл іще раз (отримавши рядок посилань 12341234), побачимо, що першим потрібним блоком буде саме блок 1, який щойно витиснули. Доведеться його зчитувати ще раз, при цьому буде заміщено блок 2, який відразу ж виявиться потрібним знову і т. д. У MRU-алгоритмі блок 4 під час першого читання заміщує блок 3, блоки 1, 2 і 4 під час другого читання перебуватимуть у кеші.

З погляду реалізації ***записування модифікованих даних на диск*** розрізняють два основні типи дискових кешів: *із наскрізним* (write through cache) і *з відкладеним записом* (write back cache).

Для кеша із наскрізним записом у разі будь-якої модифікації блоку, що перебуває в кеші, його негайно зберігають на диску. Основною перевагою такого підходу (широко розповсюдженого в часи персональних ОС і реалізованого, наприклад, в MS-DOS) є те, що файлова система завжди перебуватиме в несуперечливому стані. Головний недолік цього підходу полягає у значному зниженні продуктивності під час записування даних.

Для кеша із відкладеним записом (такі кеші застосовують у більшості сучасних ОС) під час модифікації блоку його позначають відповідним чином, але на диску не зберігають. Записування модифікованих блоків на диск здійснюється окремо за необхідності (зазвичай у фоновому режимі). Цей підхід значно виграє у продуктивності, але вимагає додаткових зусиль із забезпечення надійності та несуперечності файлової системи. Якщо система зазнає краху в той момент, коли модифіковані блоки ще не були записані на диск, без додаткових заходів всі зміни втратяться.

Основна проблема, пов'язана із реалізацією відкладеного записування, полягає у виборі ефективного компромісу між продуктивністю і надійністю. Що більший інтервал між збереженням модифікованих даних на диску, то вища продуктивність кеша, але більше роботи потрібно виконати для поновлення даних у разі збою.

Відокремлюють чотири випадки, коли зміни в кеші зберігаються на диску:

- модифікований блок витісняють з кеша відповідно до алгоритму заміщення (аналогічно до того, як це реалізовано для віртуальної пам'яті);
- закривають файл;
- явно видають команду зберегти всі зміни із кеша на диск; в UNIX-системах таку

команду звичайно називають sync, вона зводиться до виконання системного виклику із тим самим ім'ям:

```
#include <unistd.h>
sync();   // скидання всіх даних з кеша на диск
```

- проходить заданий проміжок часу (в UNIX-системах за цим стежить спеціальний фоновий процес, який називають update; за замовчуванням такий інтервал для нього становить 30 хвилин).

Системний виклик `sync()` зберігає всі зміни із кеша на диск. Гнучкішим підходом є збереження змін для конкретних файлів. Для цього у POSIX передбачено системний виклик `fsync()`. Він блокує виконання процесу до повного збереження всіх буферів відкритого файла із пам'яті на диск.

```
#include <unistd.h>
#include <fcntl.h>
int fdl = open("myfile.txt", 0_RDWR | 0_CREAT, 0644);
write(fdl, "hello", sizeof("hello"));
fsync(fdl);     // скидання всіх даних файла з кеша на диск
```

Аналогом `fsync()` у Win32 API є функція `FlushFileBuffers()`:

```
HANDLE fh = CreateFile("myfile.txt", GENERIC_WRITE, ...);
WriteFile(fh, "hello", sizeof("hello"), ...);
FlushFileBuffers(fh); // скидання всіх даних файла з кеша на диск
```

Найважливіші блоки (наприклад, блоки з довідковою інформацією, каталогами, атрибутами файлів) варто зберігати якнайчастіше, можливо, навіть після кожної модифікації; блоки із даними можна зберігати рідше.

Для застосувань, які працюють із важливими даними, рекомендовано час від часу робити явне збереження змін на диску (наприклад, виконувати системні виклики sync(), fsync() або їхні аналоги). Для таких застосувань, як СУБД, найкращим вирішенням в основному є повна відмова від дискового кеша (використання *прямого режиму доступу до диска)* і реалізація свого власного кешування. Уже згадувалося про те, що часто такі застосування зовсім відмовляються від послуг файлової системи.

Для реалізації прямого доступу до диска в Linux необхідно під час відкриття файла увімкнути спеціальний прапорець 0_DIRECT:

```
int fdl = open ("myfile.txt", 0_RDWR | 0_DIRECT, 0644);
```

Аналогом `0_DIRECT` у Win32 API є `FILE_FLAG_WRITE_THROUGH`:

```
HANDLE fh = CreateFile("myfile.txt", ...,
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_WRITE_THROUGH, NULL);
```

Під час роботи із диском оптимальною можна вважати ситуацію, коли блоки даних опиняються в пам'яті саме в той час, коли вони мають використовуватись. В ідеальній перспективі (за наявності необмеженої пропускної здатності диска і можливості пророкувати майбутнє) необхідність у кеші може взагалі відпасти – достатньо перед використанням будь-якого блоку просто зчитувати його у пам'ять заздалегідь (або один раз зчитати всі потрібні блоки і потім працювати із ними в пам'яті).

У реальності, як неодноразово було наголошено, знання про майбутнє немає, але можна спробувати його передбачити на підставі аналізу минулого. Таке пророкування може задати стратегію *випереджувального читання даних* (data prefetching, read-ahead).

Під час реалізації випереджувального читання намагаються визначити дискові блоки, які, швидше за все, використовуватимуться разом. Найчастіше при цьому застосовують принцип *просторової локальності* (spatial locality), за яким імовірність

спільного використання вища для кластерів, що належать до одного файла і розташовані на диску поруч. При цьому під час випереджувального читання файлова система після читання кластера із номером *n* перевіряє, чи є в кеші кластер із номером *n*+1, і, якщо він там відсутній, його зчитують у пам'ять заздалегідь, до використання.

Такий підхід спрацьовує для послідовного доступу до файлів, для випадкового доступу він не дає жодного виграшу у продуктивності. За цієї ситуації має сенс «допомагати» системі, організовуючи спільно використовувані дані так, щоб вони на диску перебували поруч. Можна також відслідковувати характер використання файлів і, поки він залишається послідовним, використовувати випереджувальне читання.

Опишемо ***ще один підхід до оптимізації доступу до диска***, що реалізують на низькому рівні, найчастіше у драйвері диска. Він зводиться до вибору алгоритму, який використовують для планування порядку виконання запитів на переміщення головки диска. Такі алгоритми називають алгоритмами планування переміщення головок або алгоритмами дискового планування.

Алгоритми дискового планування необхідні, бо на один фізичний ресурс (у даному разі маніпулятор диска) припадає багато запитів на використання (у цьому є подібність із алгоритмами планування процесорного часу, де фізичним ресурсом є процесор). Основною метою алгоритмів дискового планування є оптимізація механічних характеристик доступу до диска, насамперед мінімізація часу пошуку (переміщення маніпулятора для позиціювання головки на потрібній доріжці).

Розглянемо три алгоритми дискового планування. Для ілюстрації роботи кожного алгоритму припускатимемо, що спочатку головка перебуває на доріжці 50, після чого їй потрібно виконати запити на звертання до доріжок 90, 190, 40, 120, 0, 130, 70 і 80.

Найпростішим ***алгоритмом дискового планування є «першим прийшов – першим обслужений»*** (First Come – First Served, FCFS, FIFO), коли кожний запит виконують негайно після його надходження. Цей алгоритм простий у реалізації, справедливий, але недостатньо ефективний. На рис. 8.10 видно, як він спричиняє зайві переміщення головки диска.

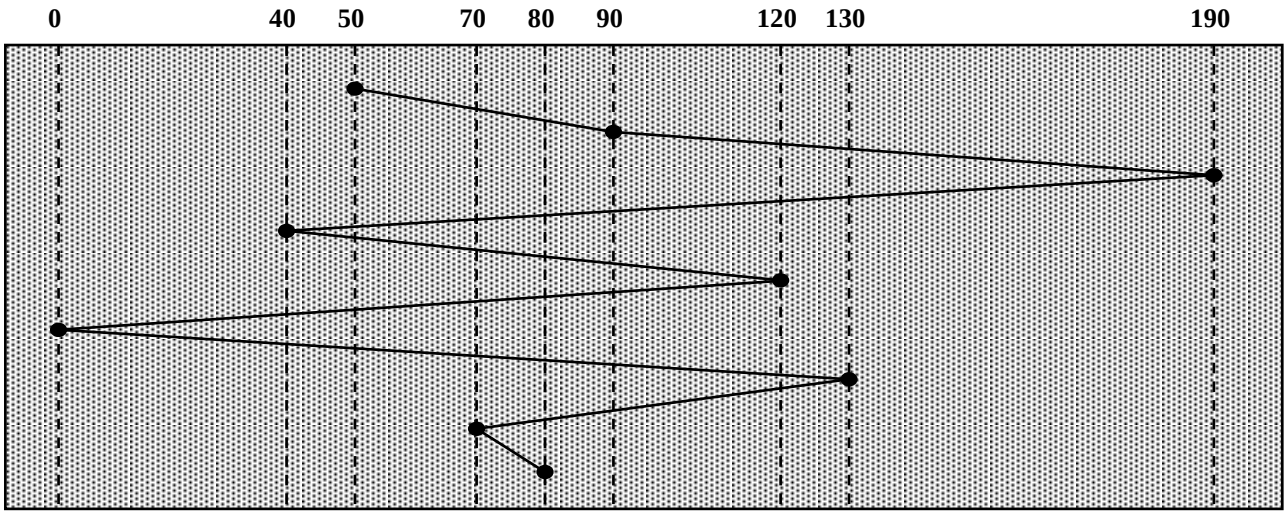

Рис. 8.10. Переміщення головки диска у разі використання алгоритму «першим прийшов – першим обслужений»

Наступний алгоритм (Shortest Seek Time First, SSTF) планує запити так, щоб першим виконувався той із них, що призводить до мінімального переміщення головки щодо її поточного положення (рис. 8.11). Цей алгоритм набагато ефективніший за FCFS-алгоритм, але не зовсім справедливий – для запитів на переміщення до крайніх доріжок диска він може спричиняти голодування.

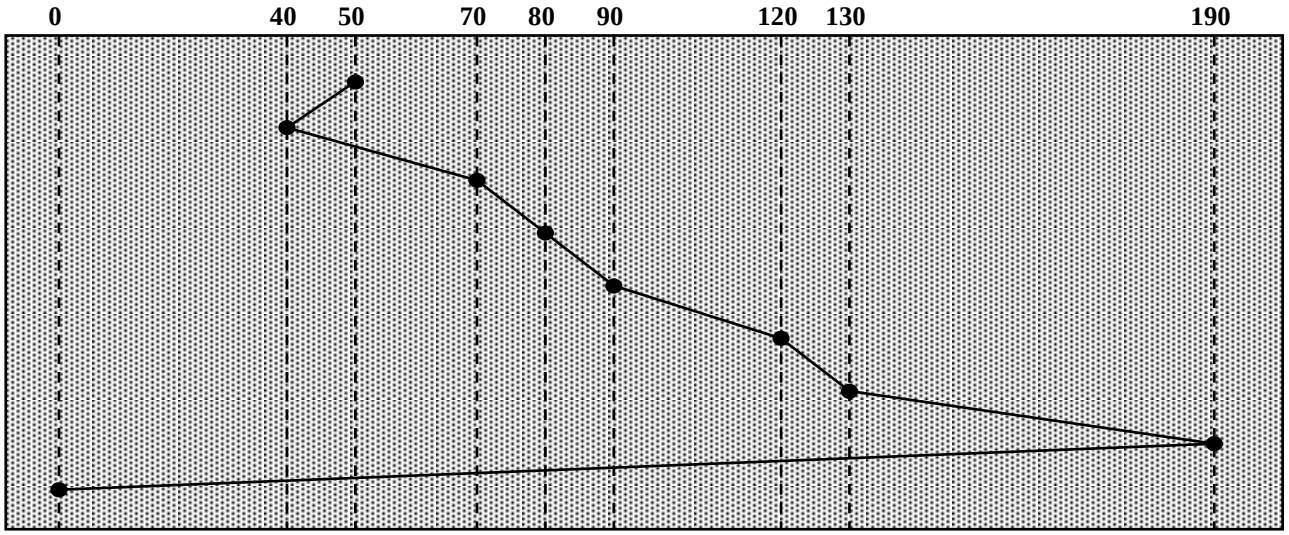

Рис. 8.11. Переміщення головки диска у разі використання алгоритму «найкоротший пошук – першим»

Припустимо, що головка перебуває на доріжці 15 і є запити на переміщення - до доріжок 20 і 120. Відповідно до цього алгоритму першим виконують запит на переміщення до доріжки 20. Якщо за цей час у систему надійде новий запит на переміщення до доріжки 25, наступним буде виконано його, а запит на переміщення до доріжки 120 залишиться чекати далі. Якщо далі постійно надходитимуть запити на переміщення до ближніх доріжок, цей запит може і зовсім не виконатися. У цьому разі отримаємо голодування.

Алгоритм «ліфта» (elevator algorithm) використовує той самий принцип, що і ліфт під час переміщення між поверхами. Відомо, коли в кабіні ліфта, що рухається, натиснути кілька кнопок поверхів (як нижче, так і вище від поточного поверху), то вона спочатку відповідатиме на ті з них, що вимагають переміщення в тому напрямку, в якому вона рухалася в момент натискання.

Алгоритм планування переміщає головку диска аналогічно до кабіни ліфта. Спочатку головка рухається у якомусь одному напрямку і виконує ті запити, які вимагають переміщення в цей самий бік. Після того, коли вона доходить до крайньої доріжки, повертає назад і починає виконувати запити на переміщення у зворотному напрямку (рис. 8.12). Далі процес повторюється. Цей алгоритм дещо програє SSTF з погляду середнього часу пошуку, але є справедливішим (голодування тут бути не може).

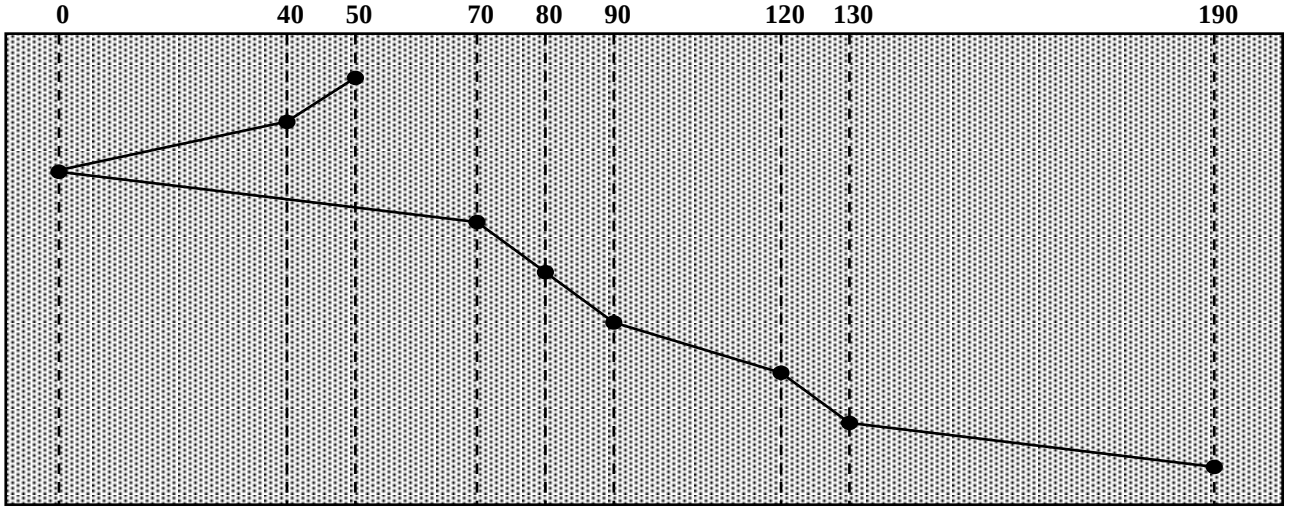

Рис. 8.12. Переміщення головки диска у разі використання алгоритму «ліфта»

8.7. Надійність файлових систем

Що відбудеться із даними файлової системи, коли комп'ютерна система зазнає краху? Розглянемо, які тут можуть виникнути проблеми.

- У разі використання дискового кеша дані, що перебувають у ньому, після відключення живлення зникнуть. Якщо до цього моменту вони не були записані на диск, зміни в них будуть втрачені. Більш того, коли такі зміни були частково записані на диск, файлова система може опинитися у суперечливому стані.
- Файлова система може опинитися у суперечливому стані внаслідок часткового виконання операцій із файлами. Так, якщо операція вилучення файла видалить відповідний запис із каталогу та індексний дескриптор, але через збій не встигне перемістити відповідні дискові блоки у список вільних блоків, вони виявляться «втраченими» для файлової системи, оскільки розмістити в них нові дані система не зможе. Щоб забезпечити відновлення після системного збою, можна заздалегідь створити резервну копію всіх (або найважливіших) даних файлової системи, аби відновити їх з цієї копії;
- під час першого завантаження після збою дослідити файлову систему і виправити суперечності, викликані цим збоєм (при цьому файлова система має прагнути робити зміни так, щоб обсяг такого дослідження був мінімальним);
- зберегти інформацію про останні виконані операції і під час першого завантаження після збою повторити ці операції. Як можуть бути реалізовані такі дії, побачимо нижче.

Найвідомішим способом підвищення надійності системи є ***резервне копіювання*** даних (data backup). *Резервним копіюванням* або *архівуванням* називають процес створення на зовнішньому носії копії всієї файлової системи або її частини з метою відновлення даних у разі аварії або помилки користувача. Аваріями є вихід жорсткого диска з ладу, фізичне ушкодження комп'ютера, вірусна атака тощо, помилки користувача звичайно зводяться до вилучення важливих файлів. Резервні копії зазвичай створюють на дешевих носіях великого обсягу.

Одним із головних завдань резервного копіювання є визначення підмножини даних файлової системи, які необхідно архівувати.

- Звичайно створюють резервні копії не всієї системи, а тільки певної підмножини її каталогів. Наприклад, каталоги із тимчасовими файлами архівувати не потрібно. Часто не архівують і системні каталоги ОС, якщо їх можна відновити із дистрибутивного диска.
- Крім того, у разі регулярного створення резервних копій є сенс організувати *інкрементне архівування* (increment backup), коли зберігаються тільки ті дані, які змінились із часу створення останньої копії. Є різні підходи до організації інкрементних архівів. Можна робити повну резервну копію через більший проміжок часу (наприклад, через тиждень), а інкрементні копії – додатково із меншим інтервалом (наприклад, через добу); можна зробити повну копію один раз, а далі обмежуватися тільки інкрементними копіями. Основною проблемою тут є ускладнення процедури відновлення даних.

Виділяють ***два базові підходи до створення резервних копій***: *фізичне і логічне архівування.* Під час фізичного архівування створюють повну копію всієї фізичної структури файлової системи, усі дані диска копіюють на резервний носій кластер за кластером. Переваги цього підходу полягають у його простоті, надійності та високій

швидкості, до недоліків можна віднести неефективне використання простору (копіюють і всі вільні кластери), неможливість архівувати задану частину файлової системи, створювати інкрементні архіви і відновлювати окремі файли. Логічне архівування працює на рівні логічного відображення файлової системи (файлів і каталогів). За його допомогою можна створити копію заданого каталогу або інкрементну копію (при цьому відслідковують час модифікації файлів), на основі такого архіву можна відновити каталог або конкретний файл. Зазначимо, що для реалізації коректного відновлення окремих файлів у разі інкрементного архівування необхідно архівувати весь ланцюжок каталогів, що становлять шлях до зміненого файла, навіть якщо жоден із цих каталогів сам не модифікувався із моменту останнього архівування. Стандартну утиліту створення логічних резервних копій в UNIX-системах називають `tar`.

Розглянемо приклад організації резервного копіювання на рівні ОС. Служба *системного відновлення* (System Restore) Windows XP дає змогу повертати систему в *точку відновлення* (restore point) – заздалегідь відомий стан, у якому вона перебувала в минулому. За замовчуванням точку відновлення створюють кожні 24 години роботи системи, крім того, можна задавати такі точки явно (наприклад, під час встановлення програмного забезпечення).

Коли служба відновлення створює точку відновлення, формують каталог точки відновлення, куди записують поточні копії системних файлів, після чого спеціальний драйвер системного відновлення починає відслідковувати зміни у файловій системі. Вилучені та змінені файли зберігають у каталозі точки відновлення, інформацію, що описує зміни (назви операцій, імена файлів і каталогів), заносять до журналу відновлення для цієї точки.

Під час виконання операції відновлення (Restore) системні файли копіюють із каталогу точки відновлення у системний каталог Windows, після чого відновлюють змінені файли користувача на підставі інформації із журналу відновлення.

Резервне копіювання – це важливий засіб підвищення надійності файлових систем. Однак для файлової системи воно є зовнішнім інструментом. Зупинимося на ***внутрішніх засобах підвищення надійності***. Вони переважно пов'язані зі зміною алгоритмів виконання файлових операцій і виконанням низькорівневих операцій відновлення.

Методи підвищення надійності файлових систем можуть бути розділені на дві основні групи.

- *Песимістичні* передбачають, що кожна операція роботи із файловою системою може потенційно потерпіти крах, залишивши систему у суперечливому стані. Пропонують модифікувати операції так, щоб цього не допустити. Продуктивність при цьому може знизитися, але надійність зросте. Час відновлення після глобальної аварії буде невеликим.
- *Оптимістичні* передбачають, що операції коли й зазнають невдачі, то рідко, і не варто жертвувати продуктивністю файлової системи для забезпечення надійності кожної операції. Помилки, якщо вони є, залишаються у файловій системі. У разі потреби відновлення після збою запускають процедуру перевірки і відновлення цілісності всієї файлової системи, яка може тривати досить довго. Більшість ОС пропонують утиліти, що здійснюють цю перевірку; в UNIX-системах таку утиліту найчастіше називають `fsck`, у Windows XP – `chkdsk` або `chkntfs`. Необхідно, однак, мати на увазі, що не всі помилки можуть бути виправлені в такий спосіб.

На практиці зазвичай використовують деяку комбінацію цих методів: для деяких помилок (наприклад, тих, які не можуть бути виправлені пізніше) використовують політику запобігання, для всіх інших – оптимістичний підхід із процедурою перевірки.

Найпростіше запобігти ***суперечливостям файлової системи***, забезпечивши синхронні операції записування. Ідея проста: виконуючи записування блоку на диск, необхідно дочекатися від диска підтвердження перед тим, як записувати наступний блок.

Насправді такий підхід не може бути застосований для всіх операцій записування (інакше продуктивність знизилася б неприпустимо, наприклад, перестало б працювати кешування). Треба зрозуміти, для якої підмножини операцій його застосування не призводить до неприйнятних результатів. Для цього розглянемо, які суперечливості можуть бути у файловій системі. Як приклад візьмемо індексоване розміщення файлів.

Наведемо кілька правил (інваріантів), які треба виконувати в несуперечливій файловій системі.

- Усі вільні блоки мають перебувати в списку вільних блоків (і навпаки, всі елементи списку вільних блоків мають справді бути вільними).
- Дисковий блок має бути використаний тільки одним файлом (два індексних дескриптори не можуть вказувати на один і той самий блок).
- Лічильник жорстких зв'язків файлового дескриптора має збігатися із числом жорстких зв'язків, що справді посилаються на нього.

Розглянемо проблеми, що виникають у разі порушення цих інваріантів. Для ілюстрації цієї проблеми почнемо з послідовності кроків під час виконання операції створення файла.

1. Шукають у поточному робочому каталозі файл із тим самим ім'ям. Якщо він є, повертають помилку, у протилежному випадку відшукують місце для елемента каталогу.
2. Шукають вільний індексний дескриптор. Знайдений дескриптор позначають як виділений.
3. Ім'я та номер дескриптора додають в елемент каталогу.

У разі збою між кроками 2 і 3 дескриптор залишиться позначений як виділений, але фактично таким не буде. У результаті ***з'являються втрачені дані.***

Проблема втрачених даних виникає під час ситуації, коли невикористані ресурси позначають як виділені, звичайно це відбувається за наявності списку або карти вільних блоків. Для запобігання її появі потрібно дотримуватися такого правила: не можна постійно зберігати покажчик на об'єкт, що перебуває у списку вільних блоків (як це зробили на кроці 2).

Подібна проблема виникає і під час вивільнення блоків. Як приклад можна навести операцію скорочення файла, коли довжину файла скорочують, а зайві блоки вивільняють, її можна виконувати так.

1.Покласти покажчик на блок в індексному дескрипторі рівним нулю.

2.Помістити блок у список вільних блоків.

Якщо поміняти місцями кроки 1 і 2 (спочатку помістити блок у список вільних блоків, а потім вилучити із дескриптора), то збій між цими діями спричиняє невірне припущення, що блок вивільнений, хоча фактично вільним він не є (на нього вказує індексний дескриптор).

Тому можна сформулювати друге правило, протилежне до першого: повторно використати ресурс можна тільки після того, як були обнулені всі покажчики на нього.

Для відновлення списку вільних блоків після збою можна застосовувати й оптимістичний підхід, реалізований утилітою типу fsck. Для цього використовують збирання сміття із маркуванням і очищенням. Спочатку припускають, що список вільних блоків включає всі блоки диска (якщо він реалізований як бітова карта, це зробити

легко); це буде етап маркування. Після цього потрібно почати із кореневого каталогу і рекурсивне обійти всі підкаталоги, вилучаючи всі їхні блоки зі списку вільних блоків – це буде етап очищення. Цей алгоритм одночасно відновлює виділені блоки, позначені як вільні, та втрачені блоки. Його недоліком є те, що потрібен обхід усього дерева каталогів (для великих файлових систем це може займати багато часу).

Для ілюстрації цієї проблеми розглянемо послідовність кроків під час виконання операції вилучення жорсткого зв'язку для файла (unlink).

1. Здійснюють обхід каталогу в пошуку цього імені. Якщо його не знайдено, повертають помилку.
2. Очищають елемент каталогу.
3. Зменшують лічильник жорстких зв'язків індексного дескриптора.
4. Якщо лічильник жорстких зв'язків дорівнює нулю, очищають індексний дескриптор і всі блоки, на які він вказує.

У разі збою між кроками 2 і 3 буде отримано надто велике значення лічильника зв'язків (зв'язок вилучений, а лічильник не змінився). Блоки, що належать до цього файла, ніколи не вивільняться (оскільки лічильник зв'язків тепер ніколи не досягне нуля). Для великих файлів це може спричинити істотні втрати дискового простору.

Припустимо, що очищення елемента каталогу і зменшення лічильника поміняли місцями (тобто спочатку зменшують лічильник, а потім вилучають зв'язок). Тепер, якщо збій відбудеться між цими двома операціями, буде надто мале значення лічильника зв'язків (коли зв'язків стане більше, ніж значення лічильника). Ця проблема ще серйозніша, ніж попередня, оскільки використовувані блоки будуть позначені як вільні. Є багато різних проявів цієї проблеми: від «зниклого» вмісту файлів (що перейшов до іншого файла) до переміщення у вільні блоки вмісту системних файлів, наприклад, файла паролів.

Є два способи боротьби з цією проблемою.

1. Оптимістичний – обходити дерево каталогів і коригувати значення лічильника зв'язків для кожного індексного дескриптора в рамках утиліти типу fsck.
2. Песимістичний – ніколи не зменшувати лічильник зв'язків до обнулення покажчика на об'єкт (наприклад, завжди синхронно зберігати на диску список вільних блоків під час кожного розміщення та вивільнення блока). При цьому використовують 1-2 додаткові операції записування на диск.

Великий обсяг дисків робить виконання програми перевірки і відновлення під час завантаження після збою досить тривалим процесом (для диска розміром у десятки Гбайтів така перевірка може тривати кілька годин). У деяких ситуаціях (наприклад, на серверах баз даних з оперативною інформацією) подібні затримки із відновленням працездатності системи після кожного збою можуть бути недопустимими. Необхідно організувати збереження інформації таким чином, щоб відновлення після збою не вимагало перевірки всіх структур даних на диску. Спроби розв'язати цю проблему привели до виникнення *журнальних файлових систем* (logging file systems).

Основна мета журнальної файлової системи – надати можливість після збою, замість глобальної перевірки всього розділу, робити відновлення на підставі інформації *журналу* (log) – спеціальної ділянки на диску, що зберігає опис останніх змін. Використання журналу засноване на важливому спостереженні: під час відновлення після збою потрібно виправляти тільки інформацію, яка перебувала у процесі зміни у момент цього збою. Це та інформація, що не встигла повністю зберегтися на диску (зазвичай вона займає тільки малу його частину).

Основна ідея таких файлових систем – виконання будь-якої операції зміни даних на диску у два етапи.

1. Спочатку інформацію зберігають у журналі (у ньому створюють новий запис). Таку операцію називають *випереджувальним записуванням* (write-ahead) або *веденням журналу* (journaling).
2. Коли ця операція повністю завершена (було підтверджено зміну журналу), інформацію записують у файлову систему (можливо, не відразу). Після того, як зміну журналу було підтверджено, усі записи в журналі, створені на етапі 1, стають непотрібними і можуть бути вилучені. Зауважимо, що синхронізація журналу і реальних даних на диску може відбуватися і явно; виконання такої операції називають *точкою перевірки* (checkpoint). Дані із журналу після цієї перевірки теж можуть бути вилучені.

Читання даних завжди здійснюють із файлової системи, журнал у цій операції не бере участі ніколи.

Після того, як інформація про зміни потрапила в журнал, самі ці зміни можуть бути записані у файлову систему не відразу. Часто об'єднують кілька операцій зміни даних у файловій системі, якщо вони належать до одного кластера, для того щоб виконати їх усі разом. У результаті кількість операцій звертання до диска істотно знижується. Ще однією важливою обставиною підвищення продуктивності, є той факт, що операції записування в журнал виконують послідовно і без пропусків. Найкращої продуктивності можна домогтися, помістивши журнал на окремий диск.

Розмір журналу має бути достатній для того, щоб у ньому помістилися ті зміни, які на момент збою можуть перебувати у пам'яті.

Є різні підходи до того, яка інформація має зберігатися в журналі.

- Тільки описи змін у метаданих (до метаданих належить вся службова інформація: індексні дескриптори, каталоги, імена тощо). Такий журнал забезпечує несуперечливість файлової системи після відновлення, але не гарантує відновлення даних у файлах. З погляду продуктивності це найшвидший спосіб.
- Змінені кластери повністю. Такий підхід не вирізняється високою продуктивністю, натомість з'являється можливість відновити дані повністю.

Файлові системи звичайно дають змогу вибрати варіант збереження інформації в журналі (це може бути зроблено під час монтування системи). На практиці вибір підходу залежить від конкретної ситуації.

Програма відновлення файлів має розрізняти дві ситуації.

- Збій відбувся до підтвердження зміни журналу. У цьому разі здійснюють *відкат* (rollback): цю зміну ігнорують, і файлова система залишається в несуперечливому стані, у якому вона була до операції. Зазначимо, що такі атомарні операції мають багато спільного із *транзакціями* – атомарними операціями у базі даних (відомо, що сервери баз даних для підтримки транзакцій також реалізують роботу із журналом).
- Збій відбувся після підтвердження зміни журналу. За цієї ситуації потрібно відновити дані на підставі інформації журналу (такий процес ще називають *відкатом уперед* – rolling forward).

Сучасні операційні системи все більше переходять до використання журнальних файлових систем; наприклад, для Linux є декілька їх реалізацій (ext3fs, ReiserFS, XFS). Файлова система NTFS також підтримує ведення журналу.

Висновки

- Для універсального доступу до інформації із прикладних програм ОС надають інтерфейс файлових систем. Файлову систему можна розглядати на двох рівнях: логічному і фізичному. На логічному рівні файлові системи абстрагують доступ до дискового простору або до інших пристроїв у вигляді концепції файлів, розташованих у каталогах. На фізичному рівні, вони реалізують набір структур даних, що зберігаються на пристроях, які забезпечують логічне відображення.
- Файлом називають набір даних на файловій системі, до якого можна звертатися за іменем. Файли є основним засобом реалізації зберігання даних в енергонезалежній пам'яті і забезпечують їхнє спільне використання різними процесами. Сучасні ОС звичайно не виділяють структуру файла, зображуючи його як послідовність байтів. Винятками є спеціальні файли. Операції доступу до файлів (відкриття, закриття, читання, записування тощо) – це універсальний інтерфейс доступу до даних в операційних системах. Важливим підходом до використання файлів у сучасних ОС стала реалізація на їхній основі відображуваної та розподілюваної пам'яті.
- Сучасні файлові системи мають структуру дерева або графа каталогів. Підходи до організації структури каталогів різні для різних ОС: деякі системи реалізують єдине дерево або граф каталогів, куди можуть підключатися (монтуватися) окремі файлові системи, розміщені на розділах диска; інші системи позначають окремі розділи диска буквами алфавіту і працюють із відповідними файловими системами окремо. Дерево каталогів стає графом за наявності зв'язків. Зв'язки можуть бути жорсткими (альтернативне ім'я для файла) і символічними (файл, що містить посилання на інший файл або каталог).

- Фізичне розміщення даних у файловій системі має забезпечувати ефективність доступу. Для цього необхідно враховувати механіку сучасних дискових пристроїв, приділяючи основну увагу мінімізації часу пошуку і ротаційної затримки.
- Основними підходами до фізичного розміщення даних є неперервне розміщення, розміщення зв'язними списками (важливою модифікацією якого є використання FAT) та індексоване розміщення. Каталоги звичайно реалізовані як спеціальні файли.
- Для підвищення продуктивності файлової системи в деяких випадках необхідно змінити її структуру (застосувати ефективні алгоритми розміщення, структури даних тощо). Такі зміни вимагають доступу до вихідного коду цієї системи і можуть бути вироблені тільки її розробниками.
- До підвищення продуктивності може також привести реалізація таких вирішень, як дисковий кеш або дискове планування. Ці вирішення є прозорими для файлової системи. Вони звичайно реалізовуються на нижчому рівні, наприклад, на рівні дискових драйверів.
- Для підвищення надійності файлових систем можна використати високорів-неві підходи, такі як організація резервного копіювання, і підходи низького рівня (запобігання суперечностям внаслідок додаткових дій під час кожної файлової операції, відновлення системи після збоїв).
- Журнальні файлові системи дають змогу відновлювати дані на підставі інформації журналу, в якому зберігають відомості про операції, виконані з файловою системою. Цю інформацію зберігають перед остаточним записом даних на диск.

Контрольні запитання та завдання

1.Перелічіть переваги і недоліки відображення файла у вигляді неструктурованого потоку байтів.

2.Які проблеми виникають, якщо дозволено одночасне монтування файлової системи в кілька точок монтування?

3.У деякій ОС файли автоматично відкриваються під час першого звертання до них і закриваються – у разі завершення процесу. Опишіть переваги і недоліки цього підходу порівняно з традиційним, коли файли відкривають і закривають явно.

4.Яка помилка може виникати під час виконання наступного коду? У яких умовах вона проявляється?

```
    void fileCopydnt fsrc. int fdest){ char buf[100]:
whileCreadCfsrc.buf.lOO) > 0)
    write(fdest.buf.lOO); }
```

5.Напишіть функцію, внаслідок виконання якої вміст масиву цілих чисел (типу Int) записується у файл, починаючи з позиції, що відповідає його середині. Файл завдяки цьому збільшується: елементи його другої половини зміщуються до кінця файла, розташовуючись після даних масиву. Параметрами функції є дескриптор файла, покажчик на початок масиву і кількість елементів у масиві.

6.Наведіть приклади взаємних блокувань, що можуть виникати у разі використання файлових блокувань.

7.Реалізуйте набір функцій обробки файлів, що містять записи фіксованої довжини. У набір повинні входити функції створення, відкриття і закриття файла, читання і відновлення значення запису за його номером, перегляду всіх записів. У разі відновлення файла всі спроби інших процесів оновити той самий запис повинні бути блоковані. Якщо під час спроби відновлення запис із таким номером не було знайдено, має бути створено новий запис.

8.У чому полягають основні відмінності реалізації відображуваної пам'яті в Linux і Windows XP? Який із підходів видається більш гнучким?

9.Реалізуйте операцію копіювання файлів у Linux і Windows XP на базі інтерфейсу файлів, відображуваних у пам'ять.

10.Розробіть систему обміну даними про поточну температуру повітря для Linux і Windows XP з використанням відображуваної пам'яті. Інформація про температуру є цілим числом. Є N процесів-клієнтів, які повинні зчитувати і відображати цю інформацію, і один процес-менеджер, якому дозволено її змінювати. Зміни, зроблені менеджером, мають відображати всі клієнти. 12. Розробіть просту клієнт-серверну систему для Linux і Windows XP з використанням поіменованих каналів. Клієнт приймає від користувача шлях файла та передає його на сервер. Сервер повинен знаходити на диску відповідний файл і направляти його вміст клієнту, котрий після отримання цих даних має відобразити їх. Якщо файл не знайдено, сервер повертає рядок з повідомленням про помилку. У разі одержання рядка «exit» сервер повинен завершити свою роботу.

11.Порівняйте продуктивність файлових систем на основі FAT і з розміщенням зв'язними списками для послідовного і випадкового доступу до файла.

12.Припустімо, що замість використання індексних дескрипторів розробник файлової системи реалізував збереження усієї відповідної інформації в елементі каталогу. Які проблеми при цьому виникають?

13.Припустимо, що у файловій системі підтримується збереження кількох версій файла. Де і як потрібно зберігати інформацію про ці версії? Як повинні виглядати базові

системні виклики для роботи з файлами в такій системі?

14.Індексний дескриптор містить інформацію про атрибути файла (16 байт), 13 прямих блоків і по одному непрямому блоку трьох рівнів. Вільне місце у дескрипторі також може бути зайняте даними файла. Обчисліть максимальний розмір файла, якщо розмір дискового блоку дорівнює 512 байт, а розмір адреси блоку – 4 байти. Скільки операцій звертання до диска необхідно виконати, щоб прочитати дисковий блок у позиції 300 000 з використанням такого індексного дескриптора?

15.На скільки збільшиться максимальний розмір файла для індексованого розміщення у разі:

а) збільшення розміру блоку вдвічі;

б) введення четвертого рівня дотичності?

16.Чи є обов'язковою наявність у системі списку вільних блоків? Як створити заново такий список у випадку його втрати?

17.Під час виконання тестового набору процесів у системі з дисковим кешем фіксованого розміру було виявлено, що звертань до диска досить багато. Після збільшення обсягу кеша продуктивність системи під час виконання цього набору процесів ще більше знизилася. Поясніть, чому це відбулося.

18.Назвіть проблеми, що можуть виникнути у випадку, якщо менеджер кеша і менеджер віртуальної пам'яті спільно використовують одну й ту саму фізичну пам'ять, фактично будучи конкурентами. У яких ситуаціях додавання основної пам'яті для використання менеджером віртуальної пам'яті дасть більший виграш у продуктивності порівняно зі збільшенням кеша? У яких ситуаціях, навпаки, вигідніше збільшувати кеш?

19.Запити до диска на переміщення голівки пристрою приходять у такому порядку: 10, 22, 20, 2, 40, 6, 38. Швидкість переміщення голівки – 6 мс/доріжку. У поточний момент голівка перебуває на доріжці 20. Нумерація доріжок – від 0 до 49. Обчисліть час пошуку у разі застосування алгоритмів:

а) FIFO;

б) SSTF;

в) алгоритму ліфта (голівка починає рух у напрямку від доріжки 0).

20.Перелічіть можливі варіанти поведінки системи резервного копіювання у випадку виявлення нею символічного зв'язку. Який варіант найкращий?

21.Поясніть, чому в системі FFS індексний дескриптор записують на диск, перед тим як зберегти на диску іншу інформацію з нового файла.

22.Утиліта fsck виконує повний обхід дерева каталогів системи з індексованим розміщенням файлів і формує списки вільних і використовуваних дискових блоків. Яка проблема наявна у файловій системі, якщо:

а) ім'я одного й того самого блоку є в обох списках;

б) імені існуючого блоку немає в жодному з них?

Які дії може виконати утиліта в тому й іншому випадку?

23.У UNIX-системах багаторазово розглядали і щоразу відкидали пропозицію дозволити встановлення жорстких зв'язків для каталогів. Чому?

24.Опишіть фактори, що впливають на вибір розміру журналу в журнальній файловій системі. За яких умов журнальна файлова система може виявитися продуктивнішою порівняно з такою самою системою, але без журналу?

25.Для яких операцій з журнальною файловою системою:

а) у журнал записують більше даних, ніж безпосередньо у файли;

б) з журналу зчитують більше даних, ніж із файлів?