ЛЕКЦІЯ 9. РЕАЛІЗАЦІЯ ФАЙЛОВИХ СИСТЕМ

- Інтерфейс файлової системи VFS
- Файлові системи Linux: `ext2fs`, `ext3fs` і `/proc`
- Файлові системи лінії FAT
- Файлова система NTFS
- Системний реєстр Windows

9.1. Інтерфейс віртуальної файлової системи VFS

Більшість UNIX-систем сьогодні керують різними типами файлових систем із використанням універсального рівня програмного забезпечення, який називають *віртуальною файловою системою* (Virtual File System, VFS). Така система :

- надає прикладним програмам однаковий програмний інтерфейс, що реалізується за допомогою системних викликів роботи з файлами;
- надає розробникам файлових систем набір функцій, які їм треба реалізувати для інтеграції їхньої системи в інфраструктуру VFS (цей набір функцій називають *інтерфейсом файлової системи).*

Основною метою VFS є забезпечення можливості роботи ОС із максимально широким набором файлових систем. Цей рівень програмного забезпечення перетворює стандартні системні виклики UNIX для керування файлами у виклики функцій низького рівня, реалізованих розробником конкретної файлової системи.

Рівень VFS забезпечує доступ через стандартні файлові системні виклики до будь-якого рівня програмного забезпечення, що реалізує інтерфейс файлової системи. Це програмне забезпечення може взагалі не працювати із дисковою файловою системою, а, наприклад, генерувати всю інформацію «на льоту». Програмні модулі, що реалізують інтерфейс файлової системи, називаються модулями підтримки файлових систем.

Файлові системи, які підтримує VFS, можуть бути розділені на три основні категорії.

- *Дискові є* файловими системами в їхньому традиційному розумінні (розташовані на диску). Вони можуть мати будь-яку внутрішню структуру, важливо тільки, щоб відповідне програмне забезпечення реалізовувало інтерфейс файлової системи. Серед дискових файлових систем, які підтримує VFS, можна виділити розроблені спеціально для Linux (`ext2fs`, `ext3fs`, `ReiserFS`); Windows XP (лінія FAT, NTFS); інших версій UNIX (FFS, XFS); CD-ROM (ISO9660).
- *Мережні* реалізують прозорий доступ до файлів на інших комп'ютерах через мережу. До них належать NFS (забезпечує доступ до інших UNIX-серверів) і 8MB (виконує аналогічні функції для серверів мережі Microsoft).
- *Спеціальні* або *віртуальні* відображають у вигляді файлової системи те, що насправді файловою системою не є. Вони не керують дисковим простором ні локально, ні віддалено. Прикладом віртуальної файлової системи є файлова система `/proc`.

Розглянемо, як працює VFS на прикладі копіювання файла із CD диску на жорсткий диск. На CD диску розміщена файлова система ISO9660, на жорсткому диску – стандартна для Linux файлова система `ext2fs`. Обидва ці дискові пристрої монтуються в окремі каталоги дерева каталогів UNIX. Припустимо, що шлях до файла на CD диску буде `/cd/dest.txt`, шлях до файла на диску – `/tmp/src.txt`.

Інфраструктура VFS абстрагує відмінності між файловими системами джерела і місця призначення (рис. 9.1).

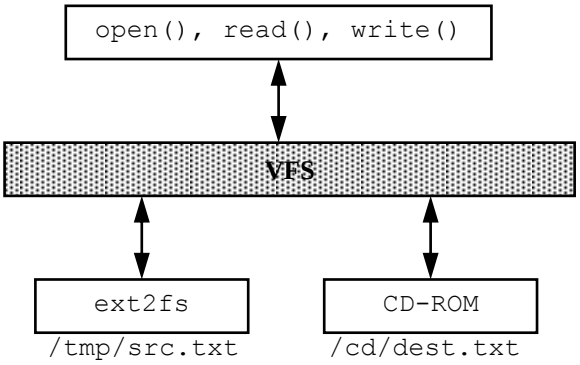

Рис. 9.1. Участь VFS в операції копіювання файлів

Прикладна програма, що здійснює копіювання, має зробити тільки такі універсальні дії:

```
    char buf[1024];
    int bytes_read, infile, outfile;
    // відкрити файл на CD диску
    infile=open ("/cd/src.txt", 0_RDONLY);
    // створити новий файл на жорсткому диску
    outfile=open ("/tmp/dest.txt", 0_WRONLY|0_CREAT|0_TRUNC, 0644);
    // подальший код копіювання
    do {
      // читання даних із вихідного файлу 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);
```

Як видно, жоден із викликів не використовує інформацію, специфічну для тієї чи іншої файлової системи. Для реалізації зворотного копіювання (із вінчестера на CD) досить поміняти місцями імена файлів − увесь інший код залишиться незмінним.

Під час розробки VFS були широко використані принципи об'єктної орієнтації. Ця інфраструктура складається із двох основних груп елементів: набору правил, яким мають підлягати файлові об'єкти, і рівня програмного забезпечення для керування цими об'єктами. Базова архітектура VFS визначає *три основних об'єктних типи*.

- *Об'єкт індексного дескриптора* (об'єкт **inode**, inode object) описує набір атрибутів і методів, за допомогою яких відображують файл на рівні файлової системи. Його назва свідчить про те, що більшість файлових систем в UNIX використовує індексоване розміщення даних; насправді цей об'єкт може інкапсулювати будь-яку структуру фізичного розміщення файла на диску.
- *Об'єкт відкритого файла* (об'єкт **file**, file object) відображає відкритий файл на рівні процесу.
- *Об'єкт файлової системи* (**filesystem** object) відображає всю файлову систему; у Linux його називають *об'єктом суперблока* (superblock object). У реалізації VFS у Linux до цих трьох об'єктів додають ще один: об'єкт елемента каталогу (об'єкт `dentry`, dentry object), що відображає елемент каталогу і його зв'язок із файлом на диску.

Для кожного із цих типів об'єктів VFS визначає набір операцій (методів), які мають бути реалізовані в об'єктах. Набір цих методів становить інтерфейс файлової системи. У кожному об'єкті є покажчик на таблицю набору адрес функцій, що реалізують ці методи для конкретного об'єкта. Програмне забезпечення VFS завжди викликає ці функції через покажчики на них, йому не потрібно знати заздалегідь, із яким конкретним типом об'єкта воно працює (з погляду об'єктного підходу у VFS реалізовано поліморфізм). Так, під час читання даних через об'єкт файлового дескриптора для рівня VFS байдуже, який саме елемент конкретної файлової системи відображає цей об'єкт (такими елементами можуть бути мережні файли, дискові файли різних файлових систем тощо). На цьому рівні відбувається тільки виклик стандартного методу читання даних через покажчик на нього. Реальна файлова система може зовсім не використовувати індексних дескрипторів – відповідний об'єкт генеруватиметься «на ходу».

Об'єкт файлової системи (об'єкт суперблоку в Linux) відображає пов'язаний набір файлів, що міститься в ієрархії каталогів. Ядро підтримує по одному такому об'єкту для кожної змонтованої файлової системи. У Linux об'єкт суперблока відображений структурою `super_block`.

Основне завдання такого об'єкта – надання доступу до об'єктів індексних дескрипторів. Кожний із них у VFS ідентифікують унікальною парою (файлова система, номер індексного дескриптора); якщо виникає потреба отримати доступ до індексного дескриптора за його номером, програмне забезпечення VFS звертається до об'єкта файлової системи, який і повертає відповідний до цього номера об'єкт індексного дескриптора.

Об'єкт файлової системи, крім того, надає операції над файловою системою загалом. Насамперед це стосується системного виклику `mount()`, що використовується для монтування файлової системи в каталог іншої файлової системи:

```
#include <sys/mount.h>
int mount(const char *partition, const char *dir, const char
          *fstype, unsigned long mflags, const void *data );
```

де: `partition` – розділ, який необхідно монтувати (задають у вигляді імені спеціального *файла пристрою* в каталозі `/dev`); `dir` – каталог, у який монтуватиметься розділ; `fstype` – тип файлової системи (задають у вигляді рядка символів, наприклад: `"ext2"` для `ext2fs`, `"msdos"` для FAT, `"iso9660"` для файлової системи CD-ROM); `mflags` – набір прапорців режиму монтування, наприклад MS_RDONLY для монтування системи тільки для читання).

Наведемо приклад виклику для монтування першого розділу жорсткого диска (заданого файлом `/dev/hda1`) із файловою системою `ext2fs` у каталог `/usr`:

```
mount("/dev/hda1", "/usr", "ext2", 0, NULL);
```

Для розмотування файлової системи використовують системний виклик `umount()`:

```
umount("/usr");
```

Для отримання інформації про змонтовану файлову систему через інтерфейс VFS у Linux використовують системний виклик `statfs()`:

```
#include <sys/vfs.h>
int statfs(const char *path, struct statfs *fsinfo);
```

де: `path` – шлях до будь-якого файла на цій файловій системі (якщо файл відсутній, виклик поверне -1, і структура не заповниться); `fsinfo` – покажчик на структуру типу `statfs`, куди будуть занесені дані про файлову систему. Серед полів структури `statfs` можна виокремити такі (усі типу `long`):

- `f_blocks` – загальна кількість блоків на файловій системі;
- `f_bavail` – кількість вільних блоків, доступних звичайному користувачу;
- `f_files` – загальна кількість індексних дескрипторів;
- `f_ffree` – кількість вільних індексних дескрипторів.

Наведемо приклад виклику:

```
struct statfs fs;
if (statfs("./myfile.txt". &fs) == -1) {
     printf("помила\n"; exit(-1); }
printf("Усього блоків %d, доступно %d\n", fs.f_blocks,
       fs.f_bavail);
```

Зупинимося на засобах VFS, що забезпечують організацію ***доступу до файлів*** із процесів. Такі засоби є найважливішим елементом цієї інфраструктури.

Спочатку розглянемо об'єкти `inode` і `file`. Обидва ці об'єкти використовують для доступу до окремого файла. Об'єкт файлового дескриптора відображає файл як ціле, тоді як об'єкт відкритого файла зображає точку доступу до даних усередині файла. Процес зазвичай не може одержати доступ до вмісту об'єкта `inode` без попереднього доступу до пов'язаного із ним об'єкта `file`.

Об'єкт `inode` відображає файловий дескриптор відповідної файлової системи. Він містить різну інформацію, специфічну для логічного та фізичного відображення файла файлової системи, зокрема номер відповідного індексного дескриптора на диску, кількість блоків, інформацію про модифіковані блоки, атрибути файла, його розмір.

Об'єкт `file` створюють за допомогою системного виклику відкриття файла (`open`). В об'єкті зберігається режим доступу до файла (читання або записування, поле `f_inode`) і покажчик його поточної позиції (поле `f_pos`). Такий об'єкт, крім того, забезпечує організацію випереджувального читання даних на підставі дій, виконаних процесом. Йому не відповідають дані реальної файлової системи. Усі використовувані об'єкти `file` файлової системи містяться у двозв'язному списку об'єктів відкритих файлів (`s_files`).

Об'єкти `file` в основному належать процесу, що їх відкрив, водночас об'єкти `inode` від процесів не залежать – до одного такого об'єкта можуть звертатися кілька процесів. Справді, навіть якщо до файла не звертається жоден процес, відповідний об'єкт `inode` може зберігатися системою (у кеші індексних дескрипторів – `inode cache`) для підвищення продуктивності, у разі коли файл буде використаний у найближчому майбутньому. Є низка структур даних, які використовуються для організації об'єктів `inode` у системі, серед них список невикористовуваних індексних дескрипторів (що працюють як кеш), два списки використовуваних дескрипторів – модифікованих і чистих, а також хеш-таблиця, що може прискорити пошук у ситуації, коли відомі номер індексного дескриптора і файлова система.

Для реалізації об'єкта `inode` розробник файлової системи має реалізувати набір методів, серед яких є `create()`, `link()`, `unlink()` і т. д.

Відзначимо відмінності в обробці каталогів. Деякі операції, визначені над каталогами (наприклад, `lookup()`, `mkdir()`, `rmdir()` тощо), не потребують попереднього відкриття відповідного каталогу як файлу (створення об'єкта `file`); ці методи реалізовані безпосередньо в об'єкті `inode`.

Крім інформації про файли на диску та їхнє використання процесами система має зберігати відомості про ***елементи каталогу***, відповідальні за перетворення імен файлів

у індексні дескриптори. Це, насамперед, потрібно для того, щоб організувати кешування часто використовуваних елементів каталогу. Під час кешування для них зберігають інформацію про те, який елемент відповідає певному індексному дескриптору. За наявності елемента каталогу в кеші немає потреби переглядати диск у пошуках дескриптора, що може дати досить значний виграш у швидкості. У Linux ці міркування привели до введення ще однієї категорії об'єктів VFS-елементів каталогу (`dentry objects`). Ці об'єкти розташовані між об'єктами відкритих файлів і об'єктами індексних дескрипторів. Кожний такий об'єкт містить інформацію про елемент каталогу (насамперед його ім'я і посилання на об'єкт відповідного індексного дескриптора). Кілька об'єктів `dentry` можуть посилатися на один об'єкт `inode`, якщо вони відповідають жорстким зв'язкам (різним елементам каталогів, що посилаються на один індексний дескриптор).

Об'єкти `dentry`, як і об'єкти `inode`, існують незалежно від процесів. Коли кілька процесів відкривають один і той самий файл через один і той самий елемент каталогу, для них не створюють окремі об'єкти `dentry`; замість цього всі їхні об'єкти `file` посилаються на один такий об'єкт, що відображає цей елемент.

Після використання об'єкти `dentry` поміщають у кеш об'єктів елементів каталогу (`dentry cache`). Він є одним із кешів кускового розподілювача пам'яті. Під час доступу до елемента каталогу спочатку завжди перевіряють, чи немає відповідного об'єкта `dentry` у цьому кеші; якщо він там є, можна негайно отримати доступ до відповідного індексного дескриптора.

Під час пошуку місцезнаходження файла, заданого повним шляхом, цей шлях розбивають на частини, що відповідають каталогам та імені файла, після чого кожна частина відображається окремим об'єктом `dentry`. Кожну із частин у результаті можна отримати з кеша, що дає змогу прискорити пошук.

Тепер розглянемо, яким чином процес отримує ***доступ до об'єктів відкритих файлів***. У керуючому блоці кожного процесу є два поля, що задають зв'язок цього процесу із файловою системою.

Перше з них – поле `fs`, що задає покажчик на структуру типу `fs_struct`. Вона містить поля, що відповідають елементам файлової системи, які задають середовище процесу. Серед них об'єкти `dentry`, що відповідають кореневому каталогу процесу (поле `root`) і його поточному каталогу (поле `pwd`), а також права на доступ до файла за замовчуванням (поле `umask`).

Другим полем є `files`, що задає покажчик на структуру типу `files_struct`. Ця структура описує відкриті файли, із якими в цей час працює процес.

Найважливішим полем структури `files_struct` є поле `fd`, що містить масив покажчиків на об'єкти `file`. Кожен елемент цього масиву відповідає файлу, відкритому процесом. Довжину масиву `fd` зберігають у полі `max_fds`.

Системні виклики UNIX використовують як параметр, що визначає відкритий файл, тобто індекс у масиві `fd`. Такі індекси називають *файловими дескрипторами* (`file descriptors`), а масив `fd` – масивом або таблицею файлових дескрипторів. Під час відкриття файла системний виклик `open()` виконує такі дії: створює новий об'єкт `file`; зберігає адресу цього об'єкта у першому вільному елементі масиву файлових дескрипторів `fd`; повертає індекс цього елемента (файловий дескриптор), який можна використати для роботи з цим файлом.

За замовчуванням під час створення процесу виділяють пам'ять під масив `fd` з 32

елементів. Якщо процес відкриє більше файлів, масив автоматично розширюється. Максимальна кількість відкритих файлів для процесу є одним із лімітів на ресурси для процесу, у Linux за замовчуванням вона дорівнює 1024, але може бути збільшена.

Перші три елементи масиву `fd` задаються автоматично під час запуску процесу, їм відповідають визначені файлові дескриптори.

Кілька елементів масиву `fd` можуть вказувати на один і той самий об'єкт `file`. Доступ через кожний із них призводить до роботи з одним і тим самим відкритим файлом. Є два основні способи домогтися такого дублювання дескрипторів.

По-перше, можна використати системні виклики `dup()` і `dup2()` для дублювання дескриптора в рамках одного процесу.

По-друге, якщо створити процес-нащадок за допомогою `fork()`, елементи таблиці дескрипторів нащадка вказуватимуть на ті самі об'єкти `file`, що й відповідні елементи таблиці дескрипторів предка.

Розглянемо цю ситуацію докладніше. Насамперед з'ясуємо, чому необхідно вводити окремі об'єкти для відкритих файлів і список `s_l` і `st` замість того, щоб помістити інформацію про відкриті файли безпосередньо в таблицю дескрипторів файлів `fd`. Причина полягає в тому, що, якби інформація була поміщена безпосередньо в таблицю, предок і нащадок не могли б спільно використовувати відкриті файли так, як це потрібно для `fork()`.

Прикладом може бути така послідовність подій.

1. Предок відкриває файл за допомогою `open()` і отримує дескриптор 4.
2. Предок записує три байти так: `write(4, "AAA", 3)`. Поточний покажчик позиції для дескриптора 4 переміщають у позицію 3, рахуючи від нуля.
3. Предок створює нащадка за допомогою `fork()` і очікує його завершення за `waitpid()`.
4. Нащадок записує п'ять байтів: `write(4, "BBBBB", 5)`, переміщаючи поточний покажчик у позицію 8, після чого завершується.
5. Предок записує ще три байти: `write(4, "CCC" ,3)`, після чого закриває файл.

У цьому разі коректна реалізація `fork()` вимагає, щоб предок продовжував записування у файл із того місця, на якому завершив своє виведення нащадок (тут – з позиції 8), тобто щоб у результаті було виведено "AAABBBBBCCC". У той же час, якби інформація про відкриті файли містилася в таблиці дескрипторів, предок після завершення нащадка не зміг би довідатися, що нащадок зробив поточну позицію рівною 8 (інформація про це була б вилучена із пам'яті разом із керуючим блоком нащадка після його завершення). У результаті предок продовжив би виведення з того місця, де він його завершив на кроці 2 (із позиції 3), поверх запису нащадка. Результат був би "AAACCCBB", що відповідно до POSIX невірно.

Тепер є вся інформація для того, щоб описати і показати взаємозв'язок між різними компонентами VFS, необхідними для отримання доступу до файла.

1. Керуючий блок процесу містить покажчик на масив файлових дескрипторів `fd`, системні виклики UNIX приймають як параметри індекси в цьому масиві. Кожен процес має окрему його копію.
2. Кожний елемент масиву файлових дескрипторів вказує на об'єкт відкритого файла (структуру `file`), ці об'єкти об'єднані у список. Кілька елементів таблиці `fd` (одного й того ж процесу або різних процесів) можуть вказувати на один і той самий об'єкт `file`.

3. Кожному об'єкту відкритого файла відповідає елемент каталогу, відображений об'єктом елемента каталогу (`dentry`). Кілька об'єктів `file` можуть посилатися на один об'єкт `dentry`; це означає, що файл був відкритий кілька разів із використанням одного й того самого імені (жорсткого зв'язку).
4. Кожному файлу файлової системи відповідає об'єкт індексного дескриптора (`inode`), ці об'єкти об'єднані у хеш-таблицю і декілька списків. Кілька об'єктів `dentry` можуть посилатися на один об'єкт `inode`, якщо вони відповідають жорстким зв'язкам.
5. Файловій системі в цілому відповідає об'єкт файлової системи. Інформація з об'єктів `inode` і об'єктів файлової системи дає змогу доступу до функцій підтримки конкретної файлової системи, що дає доступ до реального файла (на диску, мережного файла тощо).

Взаємозв'язок між різними компонентами VFS, необхідними для отримання доступу до файла, показано на рис. 9.2

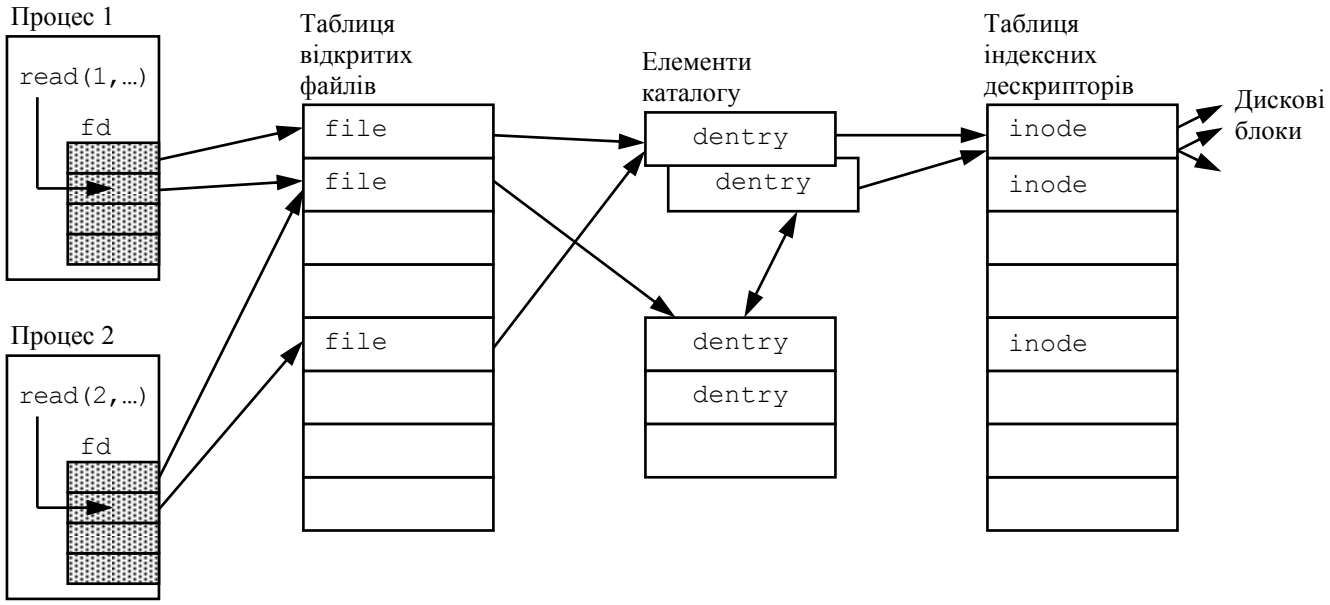

Рис. 9.2. Доступ до файла з процесу Linux

9.2. Файлові системи `ext2fs` і `ext3fs`

Стандартною дисковою файловою системою для Linux є друга розширена файлова система (`ext2fs`). Розглянемо особливості її реалізації.

Однією із базових відмінностей `ext2fs` від FFS є інша політика розподілу дискового простору. Як зазначалося, у FFS було дозволено розподіляти дисковий простір на блоки по 4 і 8 Кбайт, при цьому малі частини таких блоків, що залишилися після розподілу, своєю чергою розділяли на фрагменти по 1 Кбайт. В `ext2fs` ситуація змінилася − дисковий простір розподіляють на блоки тільки одного розміру. За замовчуванням він становить 1 Кбайт, хоча можна під час форматування файлової системи задати й більший розмір – 2 або 4 Кбайт.

Основою обробки запитів введення-виведення в `ext2fs` є кластеризація суміжних запитів для досягнення максимальної продуктивності, для чого, як і в FFS, прагнуть розташовувати зв'язані дані разом.

Для вирішення цієї задачі використовують групування даних, схоже за своїми принципами на використання груп циліндрів у FFS. Відмінності тут переважно зумовлені тим, що сьогодні у дисках використовують циліндри з різною геометрією залежно від

відстані до центра пластини, тому об'єднання таких циліндрів у групи фіксованого розміру часто не відповідає фізичній структурі диска. Виходячи з цього, в `ext2fs` дисковий простір ділять не на групи циліндрів, а просто на групи блоків (`block groups`), не прив'язані до геометрії диска. Такі групи за структурою схожі на групи циліндрів: кожна група блоків теж є зменшеною копією файлової системи із суперблоком, таблицею індексних дескрипторів тощо (рис. 9.3).

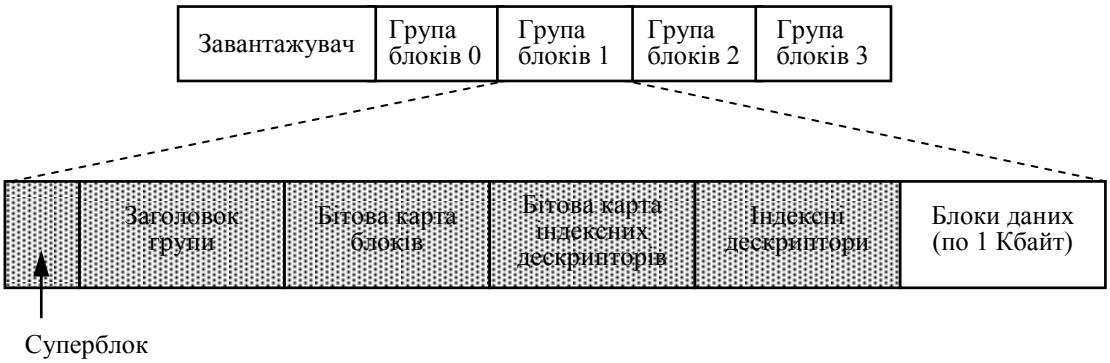

Рис. 9.3. Структура файлової системи ext2fs

Під час розміщення файла для нього спочатку вибирають групу блоків і в ній розміщують індексний дескриптор. При цьому перевагу отримує група блоків каталогу, де перебуває цей файл. Після цього, під час розподілу дискових блоків для файла, пробують вибрати ту саму групу, у якій перебуває індексний дескриптор. Файли каталогів не збирають разом, їх розподіляють рівномірно у доступних групах блоків для зменшення загальної фрагментації та більш рівномірного використання диска (навколо кожного файла каталогу групуватимуться індексні дескриптори його файлів, а навколо цих індексних дескрипторів − блоки їхніх файлів).

Облік вільних блоків та індексних дескрипторів ведуть за допомогою пари бітових карт − по одній на кожну групу блоків. Розмір кожної такої карти дорівнює одному блоку (1 Кбайт), тому максимально можлива кількість блоків та індексних дескрипторів у групі дорівнює 8 Кбайт. Під час розміщення перших блоків нового файла файлова система починає пошук вільних блоків від початку групи блоків, у разі розширення наявного файла пошук триває від блоку, виділеного найпізніше. Цей пошук відбувається у два етапи: на першому в бітовій карті відшукують вільний байт (8 біт), що відповідає 8 неперервно розташованим дисковим блокам (аналогам блоку в 8 Кбайт). Якщо байт знайти не вдалося, виконують ще один пошук, але при цьому відшукують будь-який вільний біт. Якщо байтовий пошук завершився успішно, система шукає вільні біти від початку цього байта назад до першого зайнятого біта. Метою цього пошуку є зменшення фрагментації (так не залишатиметься «дірок» між виділеним інтервалом блоків і попередніми зайнятими блоками).

Після того, як вільний блок було знайдено одним із двох способів, його виділяють для використання файлом. Додатково випереджувальне виділяють до 8 блоків, що розташовуються за ним (якщо зайнятий блок під час цього виділення виявляють раніше, то виділяють менше ніж 8 блоків). Метою такого випереджувального виділення є зменшення фрагментації та витрат часу на виділення дискових блоків. У разі закриття файла всі виділені таким чином блоки, що реально не використані, повертаються у бітову карту як вільні.

Розмір кожного індексного дескриптора становить 128 байт. Використовують 12 прямих блоків і по одному непрямому блоку першого, другого та третього рівнів. Довжина адреси блоку становить 4 байти, що більше, ніж стандартна довжина для

багатьох UNIX-систем, тому можна адресувати більше дискового простору. Місце для розміщення списків керування доступом зарезервоване, але може бути використане тільки в ядрі версії 2.6.

Файлова система ext3fs є розширенням ext2fs внаслідок додавання журналу. Важливою її особливістю є можливість відображення в журналі змін не тільки метаданих файлової системи, але й файлових даних (підтримку цієї можливості задають під час монтування системи). Це призводить до істотного зниження продуктивності, але дозволяє підвищити надійність.

Можуть бути задані три режими роботи із журналом: *режим журналу* (journal), за якого всі зміни даних зберігаються в журналі; *упорядкований режим* (ordered), за якого зберігаються тільки зміни в метаданих, але при цьому файлові операції групують так, щоб блоки даних зберігалися на диску перед метаданими, внаслідок чого знижується небезпека ушкодження інформації всередині файлів; *режим мінімального записування* (writeback), за якого зберігаються тільки метадані.

За замовчуванням використовують упорядкований режим.

Журнал файлової системи ext3fs зберігають у схованому файлі journal у кореневому каталозі файлової системи, її код не працює із файлом журналу безпосередньо, для цього використовують спеціальний рівень коду ядра із назвою JBD (Journaling Block Device Layer). Код JBD групує дискові операції в транзакції, інформацію про які фіксують у журналі. Система гарантує, що інформація про підтверджену транзакцію буде вилучена із журналу лише тоді, коли всі відповідні блоки даних записані на диск.

Під час завантаження системи після збою утиліта e2fsck переглядає журнал і планує до виконання всі операції записування, описані підтвердженими транзакціями.

9.3. Файлова система /proc

Найцікавішим прикладом реалізації інтерфейсу файлової системи VFS (Virtual File System) для ***доступу до даних, що не перебувають на диску***, є файлова система /proc. Ці дані насправді не зберігаються ніде, вміст кожного файла і каталогу генерують «на льоту» у відповідь на запити користувача.

Така файлова система вперше з'явилася в UNIX System V Release 4. Вона ґрунтувалась на тому, що кожному процесові у системі відповідає каталог файлової системи /proc, при цьому ім'я каталогу має збігатися з цифровим зображенням ідентифікатора (pid) цього процесу (наприклад, процесу із pid=25 має відповідати каталог /proc/25). Із каталогу можна дістати доступ до різних файлів із визначеними іменами, при цьому доступ до кожного з них має спричиняти генерування різної інформації про процес у текстовому форматі, зручному для синтаксичного аналізу. Такою інформацією може бути вміст командного рядка процесу – /proc/pid/cmdinfo, відомості про поточне завантаження ним процесорів – /proc/pid/cpu, різноманітна статистика – /proc/pid/status тощо. В цілому вся інформація, відображувана програмою ps, має бути доступна через дану файлову систему. Ця інформація є динамічною. Коли, наприклад, переглянути двічі поспіль файл, що відображає поточне завантаження процесора, можемо отримати різні результати.

У Linux реалізовано вищеописане подання інформації про процеси, але, крім цього, у відображення /proc було додано багато нових файлів і каталогів. Основним призначенням цих засобів доступу є надання різної статистики щодо системи, зокрема частина цих файлів і каталогів надає інтерфейс до служб ядра. За допомогою цієї

системи можна здобути вичерпну інформацію про завантажені модулі ядра (/proc/modules), змонтовані файлові системи (/proc/mounts), зовнішні пристрої (/proc/devices, /proc/pci, /proc/ide), процесори (/proc/cpuinfo), стан пам'яті (/proc/meminfo) тощо.

Особливо важливий інтерфейс, що забезпечує доступ до внутрішніх змінних ядра; він реалізований через файли в каталогах /proc/sys і /proc/sys/kernel. Суттєвим тут є той факт, що /proc підтримує не лише читання значень таких змінних, але їхнє редагування без перезавантаження системи і перекомпіляції ядра (через записування нових значень у відповідні файли). Прикладом змінної, котра може бути відредагована через інтерфейс /proc, є максимально допустима кількість потоків у системі, що може бути модифікована під час її роботи шляхом зміни файла /proc/sys/kernel/threads-max:

```
# echo 10000 > /proc/sys/kernel/threads-max
```

Реалізація цієї файлової системи ґрунтується на тому, що кожному файлу і каталогу в ній присвоюють унікальний і незмінний номер індексного дескриптора. У разі доступу до файла цей номер передають у відповідний метод об'єкта індексного дескриптора VFS (наприклад, метод читання файла). Цей метод замість звертання до диска просто перевіряє номер індексного дескриптора і залежно від його значення виконує потрібний код (наприклад, зчитує інформацію з керуючого блока відповідного процесу).

Використання файлової системи /proc із прикладних програм організоване дуже просто. Необхідно виконувати стандартні системні виклики роботи з файлами, шлях до яких відомий, зчитувати із них інформацію у текстовому форматі, виконувати її синтаксичний розбір і виділяти потрібні дані. Жодних додаткових прав доступу для цього не потрібно. Відомий приклад того, як змінилася реалізація утиліти ps (що відображає інформацію про процеси у системі) із появою /proc. Якщо раніше вона була реалізована як привілейований процес, котрий зчитував інформацію про процеси безпосередньо з пам'яті ядра, то тепер її стало можливо реалізувати як звичайну прикладну програму, що зчитує і форматує текстові дані, доступні через /proc.

Прикладом використання файлової системи /proc може бути визначення загального обсягу пам'яті в системі. Цю інформацію повертають після читання із файла /proc/meminfo. Перші два рядки його мають такий вигляд:

```
      total:    used:     free:   shared:  buffers:   cached:
Mem: 63754240 60063744   3690496    49152   2600960   31346688
```

Тут видно, що потрібна нам інформація перебуває у другому рядку. Програма має відкрити файл, зчитати з нього дані та знайти в них потрібний фрагмент:

```
int fdl, bytes_read, total_mem;
char meminfo[1024], *match;
fdl=open("/proc/meminfo",0_RDONLY);//відкриття /proc/meminfo
bytes_read = read(fdl, meminfo, sizeof(buf));//читання даних
meminfo[bytes_read] = '\0';
match = strstr(meminfo, "Mem:");     // пошук фрагмента
sscanf(match,"Mem: %d", &total_mem);//зчитування інформації
printf("Усього пам'яті: %d\n", total_mem);
close(fdl);
```

9.4. Файлові системи лінії FAT

Згадуючи про файлові системи лінії FAT, матимемо на увазі кілька близьких за організацією файлових систем, які розрізняють за способом адресації кластера на диску (FAT-12, FAT-16, FAT-32, FAT-64) або наявністю підтримки довгих імен (така підтримка є для всіх реалізацій FAT, використовуваних у Consumer Windows і в лінії Windows XP).

Розглянемо тільки системи із підтримкою довгих імен, а на відмінностях в адресації зупинимося окремо. Якщо виклад не зачіпатиме відмінностей в адресації, вживатимемо назву *FAT*, розуміючи під нею кожну із файлових систем цього сімейства.

Розглянемо ***структуру*** розділу, що містить файлову систему FAT.

- Після завантажувального сектора, в якому знаходиться завантажувач системи, розташовані дві копії таблиці розміщення файлів (FAT). Резервну копію FAT використовують для відновлення основної копії у разі її ушкодження. Усі операції, що ведуть до зміни FAT, негайно синхронізують із резервною копією.
- Далі розташований кореневий каталог, під який виділяють 32 Кбайт, що дає змогу зберігати в ньому 512 елементів (на кожний елемент виділено 32 байта).
- За кореневим каталогом слідує ділянка даних, у якій розташовані всі файли і каталоги, крім кореневого.

Під час розміщення файла FAT проглядають від початку в пошуках першого вільного індексного покажчика. Після цього в поле каталогу, що відповідає номеру першого кластера, заносять номер цього покажчика. Далі будують ланцюжок покажчиків у FAT. В останній індексний покажчик файла заносять ознаку кінця файла. Зазначимо, що якщо кластери, які слідують за початковим кластером файла, у момент розміщення виявляться вільними (це найчастіше буває після форматування розділу), файл займе суміжні кластери і буде неперервним, у противному разі між кластерами цього файла на диску розташовуватимуться кластери інших файлів. Така ситуація відповідає зовнішній фрагментації дискового простору.

Тепер зупинимося на ***відмінностях у версіях*** FAT, що ґрунтуються на особливостях адресації. Найважливішою характеристикою FAT є *розмір індексного покажчика*. Оскільки кожен із покажчиків вказує на кластер, від їхнього розміру залежить загальна кількість кластерів на диску і розмір FAT. Відмінності між версіями FAT визначаються розміром індексного покажчика: для FAT-12 він дорівнює 12 біт (що відповідає 4 Кбайт кластерів), для FAT-16 – 16 біт (64 Кбайт кластерів), для FAT-32 – 32 біти (2^{32} кластери).

Максимальний обсяг розділу, що може бути адресований FAT конкретної версії, залежить від розміру кластера і максимальної кількості адресованих кластерів, обумовленої довжиною індексного покажчика. Що більший кластер, то менше їх потрібно для адресації того самого обсягу диска і навпаки; з іншого боку, великий розмір кластера спричиняє значну внутрішню фрагментацію.

Звичайно вибирають мінімальний розмір кластера, який дає змогу адресувати весь розділ визначеного обсягу, при цьому бажано, щоб кластер не був більший за 4 Кбайт. Наприклад, для FAT-12 і розміру кластера 4 Кбайт обсяг розділу не може перевищувати 16 Мбайт (тому таку систему рекомендують лише для дискет). Система FAT-16 за такого розміру кластера може бути застосована для розділів до 512 Мбайт, для більших розділів потрібно збільшувати розмір кластера. Наприклад, для розділу розміром понад 1 Гбайт розмір кластера має бути 32 Кбайт.

Із подоланням цього обмеження насамперед пов'язане впровадження FAT-32, що дає змогу використати кластери на 4 Кбайт для розділів розміром до 8 Гбайт.

Розмір FAT залежить від розміру індексного покажчика та обсягу розділу, у FAT-32

для великих розділів вона може досягати кількох мегабайт. ОС звичайно кешує FAT, але якщо зовнішня фрагментація диска значна і розмір FAT великий, ефективне кешування може бути ускладнене, внаслідок чого знижується продуктивність системи.

Елемент каталогу в FAT містить:

- ім'я файла у форматі 8.3;
- поле атрибутів (1 байт) − тільки для читання, системний, схований;
- дату і час останньої модифікації файла; дату останнього доступу;
- номер першого кластера файла (4 байти);
- розмір файла (4 байти).

У разі вилучення файла перший символ відповідного елемента каталогу заміняють на символ 'x', який означає, що цей елемент можна використовувати заново, а всі кластери файла у FAT позначають як вільні. Номер першого кластера і довжину файла в елементі каталогу не знищують, що дає можливість відновити вміст вилученого файла, коли його кластери розташовані послідовно (на цьому заснована дія утиліт відновлення файлів після вилучення).

До появи Windows 95 FAT надавала змогу використовувати тільки імена, що складаються з 11 значущих символів (8 − ім'я файла, 3 – розширення), при цьому вони зберігалися у відповідному елементі каталогу. Введення довгих імен призвело до того, що на додачу до традиційного імені (яке тепер називають «коротким іменем») у таких записах каталогу зберігають ім'я завдовжки до 255 символів фрагментами по 13 двобайтових Unicode-символів. При цьому коротке ім'я отримують із довгого шляхом додавання до перших 6 символів у верхньому регістрі суфікса ~1, ~2 і т. д., залежно від наявності таких імен до цього часу в каталозі. Для того щоб відрізнити фрагмент довгого імені від елемента каталогу, котрий відповідає файлу, для записів із фрагментами імені задають значення атрибутів 0×0F, що як атрибут файла не може бути використане. Для відстеження відповідності між елементом каталогу і довгим іменем використовують поле контрольної суми.

9.5. Файлова система NTFS

Файлова система NTFS є основною файловою системою ОС лінії Windows XP. Головними її перевагами є надійність, високий ступінь захищеності та раціональне використання дискового простору.

Логічний розділ із розміщеною файловою системою у NTFS називають *томом.* Усі метадані (включаючи інформацію про структуру тому) зберігають у файлах на диску.

Розмір кластера задають під час високорівневого форматування розділу диска, і він може варіюватися від розміру сектора диска до 4 Кбайт. Кожному кластеру присвоюють логічний номер (Logical Cluster Number, LCN), який використовують у системі для ідентифікації кластера. Фізичний зсув на диску в цьому разі може бути отриманий множенням розміру кластера на LCN.

Загальна структура тому NTFS показана на рис. 9.4.

Рис. 9.4. Структура тому NTFS

Файл для NTFS відображається складніше, ніж для файлових систем UNIX та сімейства FAT. Він складається із набору атрибутів, причому кожний з них є незалежним *потоком* (stream) байтів, який можна створювати, вилучати, зчитувати і записувати. Деякі атрибути є стандартними для всіх файлів, до них належать ім'я (FileName), версія (Version), стандартна інформація про файл, що включає час створення, час відновлення тощо (Standard Information). Інші атрибути залежать від призначення файла (наприклад, каталог включає як атрибут Index Root структуру даних, що містить список файлів цього каталогу). Є універсальний атрибут Data, що містить всі дані файла *(потік даних)*. Для звичайних файлів він може бути єдиним із необов'язкових.

Важливою характеристикою NTFS є те, що файл може містити більш як один атрибутів Data, які розрізняють за іменами. У зв'язку з цим кажуть, що дозволена наявність кількох поіменованих потоків даних усередині файла.

За замовчуванням файл містить один потік даних, що не має імені. Для доступу до додаткових потоків використовують звичайні функції роботи із файлами (`CreateFile()`, `WriteFile()`, `ReadFile()` тощо), ім'я потоку при цьому записують через двокрапку після імені файла:

```
DWORD bytes_read. bytes_written;
// створення нового поіменованого потоку у файлі
HANDLE hf = CreateFile("myfile.txt:mystream",
GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_NEW, 0, 0);
// записування у цей потік
WriteFile(hf, "hello\n", 7, &bytes_written, 0);
CloseHandle(hf);
```

Зазначимо, що під час звертання до такого файла за іменем (`myfile.txt`) отримається доступ тільки до стандартного безіменного потоку. З іншого боку, у разі спроби перенести такий файл у файлову систему, що не підтримує кілька потоків усередині файла (наприклад, FAT), буде видано попередження про можливу втрату даних.

Кожний файл у NTFS описують одним або кількома записами в масиві, що зберігається у спеціальному файлі − *головній таблиці файлів* (Master File Table, MFT). Розмір такого запису залежить від розміру розділу і звичайно становить 2 Кбайт. Атрибути невеликого розміру (наприклад, ім'я файла), які зберігають у записі MFT безпосередньо, називаються *резидентними атрибутами.* Атрибути великого обсягу (наприклад, ті, що містять дані файла), зберігають на диску окремо − це *нерезидентні атрибути.* Розміщують такі дані екстентами (групами кластерів заданої довжини), при цьому покажчик на кожен екстент зберігають у записі MFT (такий покажчик складається із трьох елементів: порядкового номера кластера на томі (LCN), номера кластера всередині файла (VCN) і довжини екстента). Максимальний розмір MFT становить 245 записів.

Можна виділити кілька способів зберігання файлів залежно від їхнього розміру.

- Файли малого розміру можуть бути розміщені повністю в одному записі MFT і виділення додаткових екстентів не потребують.
- Файли більшого розміру потребують використання нерезидентних атрибутів зі зберіганням покажчиків на кожен екстент такого атрибута в атрибуті Data вихідного запису MFT.
- Файли із великим набором атрибутів або високою фрагментацією (коли атрибут Data не вміщує всіх покажчиків на екстенти) можуть потребувати для розміщення кілька записів MFT. Основний запис у цьому разі ще називають базовим записом файла (base file record), інші − записами переповнення (overflow records). Базовий запис файла містить список номерів записів переповнення в атрибуті Attribute List.
- У разі файлів надзвичайно великого розміру в базовому записі може бракувати

місця для зберігання списку номерів записів переповнення, тому атрибут Attribute List можна зробити нерезидентним.

Кожен файл на томі NTFS має унікальний ідентифікатор − файлове посилання (file reference). Розмір такого посилання становить 64 біти, з них 48 біт займає номер файла (індекс у MFT), а 16 біт − номер послідовності, який збільшують під час кожного нового задавання елемента MFT і використовують для контролю несуперечливості файлової системи.

Ідентифікація файла за його номером у MFT подібна до індексованого розміщення файлів, при цьому MFT відіграє роль ділянки індексних дескрипторів. Однак, на відміну від індексних дескрипторів, записи MFT можуть безпосередньо містити дані файла.

Каталоги у NTFS зберігають двома способами. Для невеликих каталогів зберігають лише список їхніх файлів, використовуючи для цього резидентний атрибут Index Root. Коли каталог досягає певного розміру, з ним пов'язують спеціальну структуру даних − B+-дерево, основною властивістю якого є те, що шлях від кореня до будь-якого вузла завжди однієї довжини. Кожен елемент такого дерева містить відповідне файлове посилання, а також копії деяких атрибутів MFT (серед них ім'я файла, час доступу, розмір файла). Це зроблено для прискорення виконання операцій, які не вимагають доступу до даних файла, наприклад перелічення елементів каталогу.

Приклад: якщо вірус *заховав* каталоги на диску (на флешці), потрібно змінити атрибути цих схованих каталогів. Найпростіший спосіб – запустити на виконання bat-файл (у томі, де цей вірус «попрацював»), вміст якого наступний:

```
@echo off
mode con codepage select=1251 > nul
echo Please wait
attrib -s -h -r -a /s /d
```

Як зазначалося, усі метадані зберігають у спеціальних файлах. Першим із них ($Mft) є MFT (тобто як перший запис MFT містить посилання на себе), другий ($MftMirr) містить резервну копію перших 16 записів MFT. Розглянемо деякі інші спеціальні файли.

- Файл журналу ($LogFile) – містить інформацію про журнал файлової системи.
- Файл тома ($Volume) − містить ім'я тома, версію NTFS і біт, вмикання якого означає, що том, можливо, був ушкоджений і його потрібно перевірити під час завантаження.
- Таблиця визначення атрибутів ($AttrDef) − задає набір атрибутів тома і допустимих операцій над ними.
- Файл \ − відображає кореневий каталог.
- Файл бітової карти ($BitMap) − містить бітову карту кластерів диска з індикацією зайнятих кластерів.
- Файл $Boot – відображає завантажувальний сектор, розташований за адресою, де його може знайти завантажувач BIOS. У ньому також зберігають адрес MFT.
- Файл зіпсованих кластерів ($BadClus) − відстежує зіпсовані кластери на диску. Система NTFS використовує технологію перерозподілу збійних кластерів, що дає змогу підставляти під час звертання до збійних кластерів дані коректних, приховуючи цим наявність проблем.
- Файл $Security − містить загальні атрибути безпеки.

Починаючи із Windows 2000, у NTFS є можливість задавати *точки повторного аналізу* (reparse points). Така точка − це спеціальний файл, пов'язаний із блоком даних,

що містить виконуваний код. Коли під час аналізу шляху до файла трапляється така точка, відбувається програмне переривання, що передає керування коду пов'язаного з нею блоку. Він зазвичай переносить подальший пошук на інший каталог або інший том – так можуть бути реалізовані зв'язки і монтування файлових систем.

Для монтування файлової системи із прикладної програми використовують функцію `SetVolumeMountPoint()`:

```
BOOL SetVolumeMountPoint(LPCTSTR mount_point, LPCTSTR volume);
```

Тут `mount_point` – ім'я наявного каталогу (включаючи завершальний зворотний слеш), що стане точкою монтування, наприклад `"c:\\disk1\\"`; `volume` – унікальне ім'я, що ідентифікує том із файловою системою і має формат `\\?\Volume{GUID}`, де GUID є унікальним 128-бітовим ідентифікатором, який широко застосовують у Windows-системах.

Для того щоб отримати унікальне ім'я тому за його символьним іменем (C:\ тощо), потрібно виконати такий код:

```
char volname[1024];     // унікальне ім'я тому
GetVolumeNameForVolumeMountPoint("C:\\",volname,sizeof(volname));
```

Тепер код для монтування матиме такий вигляд:

```
SetVolumeMountPoint("D:\\disk1\\", volname);
```

Для розмотування потрібно використати функцію `DeleteVolumeMountPoint()`:

```
DeleteVolumeMountPoint("D: \\disk1\\");
```

Система NTFS може робити ***стискання*** як окремих файлів, так і всіх файлів у каталозі. Для цього дані файла розділяють на одиниці стискання, що є групами із 16 суміжних кластерів. Під час записування на диск кожної такої групи до неї застосовують алгоритм стискання. Якщо при його застосуванні було досягнуто виграш у дисковому просторі, після даних стиснутих 16 кластерів міститься посилання на екстент із нульовою адресою, яке означає, що попередні 16 кластерів були стиснуті. Якщо стиснути групу кластерів не вдалося, її зберігають на диску без змін. Коли необхідно прочитати файл, NTFS визначає всі стиснуті групи (після яких є посилання на екстенти із нульовими адресами), дані цих екстентів зчитують і розпаковують у випереджувальному режимі. Зауважимо, що для довільного доступу до стиснутого файлу необхідно розпакувати всі дані від початку файла до позиції доступу, вибір 16 кластерів як одиниці стискання є спробою досягти в цьому разі компромісу між швидкістю доступу та ефективністю стискання. Функція `GetFileSize()` для таких файлів повертає розмір файлу без урахування стискання. Для того щоб дізнатися, скільки місця на диску займає стиснутий файл (тобто його розмір після стискання), необхідно скористатися функцією `GetCompressedFileSize()`:

```
HANDLE fh = CreateFile("myfile.txt", GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
DWORD size = GetFileSize(fh, NULL);
DWORD size_on_disk=GetCompressedFileSize("myfile.txt", NULL);
Printf("розмір без стискання: %d,
            зі стисканням: %d\n", size, size_on_disk);
```

Починаючи із версії для Windows 2000, у NTFS з'явилася підтримка розріджених файлів. Для таких файлів задається спеціальний атрибут, після чого в послідовності номерів кластерів елемента MFT можуть бути утворені «діри», якщо після створення файла до цих кластерів не було звертання. У подальшому, якщо під час читання файла трапляється така «діра», замість неї без доступу до диска система повертає блок,

заповнений нулями. Зазначимо, що можна явно задавати ділянки файла, до яких не планують звертатися.

Для задавання атрибутів стискання і розрідженості необхідно використати низькорівневу функцію керування введенням-виведенням `DeviceIoControl()`.

Основою забезпечення надійності у NTFS є те, що це − журнальна файлова система. Усі зміни структур даних файлової системи відбуваються всередині атомарних транзакцій, які виконуються повністю або не виконуються зовсім. У журнал записують інформацію про початок і підтвердження транзакції. У разі відновлення після збою спочатку повторно виконують дії всіх підтверджених транзакцій, а потім відміняють дії всіх тих, які не були підтверджені (у журналі транзакцій завжди наявна інформація, необхідна для виконання як однієї, так і іншої дії).

Кожні 5 с у журналі зберігають інформацію про точку перевірки, тоді ж інформацію про зміни файлів зберігають на диску остаточно. Записи журналу транзакцій до точки перевірки не потрібні й можуть бути вилучені.

Журнал зберігають у файлі метаданих із назвою $LogFile, який створюють під час форматування розділу. Він складається з двох секцій: ділянки записів журналу (logging area), яка є циклічним списком записів журналу, і ділянки перезапуску (restart area), що містить дві ідентичні копії поточної інформації про стан журналу (наприклад, там зберігають позицію, з якої потрібно буде почати відновлення).

Якщо в журналі бракує місця, Windows XP ставить транзакції в чергу, і всі нові операції введення-виведення відміняються. Коли всі поточні операції виконані, NTFS звертається до менеджера кеша, щоб той записав весь кеш на диск, після чого журнал очищають і виконують усі відкладені транзакції.

9.6. Особливості кешування у Windows XP

У Windows XP реалізовано єдиний кеш, що обслуговує всі файлові системи. Керування цим кешем здійснює *менеджер кеша* (Cache Manager).

Основною особливістю менеджера кеша є те, що він працює на більш високому рівні, ніж файлові системи (на відміну, наприклад, від традиційної архітектури UNIX, де кеш розташований нижче від файлових систем).

Менеджер кеша тісно взаємодіє із менеджером віртуальної пам'яті. Розмір кеша міняють динамічно залежно від обсягу доступної пам'яті. Менеджер віртуальної пам'яті резервує для системного кеша половину системної ділянки адресного простору процесу (верхні 2 Гбайт). У своїй роботі менеджер кеша використовує технологію відображуваної пам'яті: файли відображаються на цей адресний простір, після чого відповідальність за керування введенням-виведенням передають менеджерові віртуальної пам'яті. Кеш розділяють на блоки по 256 Кбайт, кожен із блоків може відображати ділянку файла.

Блоки у кеші описують за допомогою керуючого блоку віртуальної адреси або блоку адреси (Virtual Address Control Block, VACB), що містить віртуальну адресу файла і зсув усередині файла для цього блоку. Глобальний список таких блоків підтримує менеджер кеша.

Для відкритих файлів підтримують окремий масив покажчиків на блоки адреси. Кожен елемент масиву відповідає ділянці файла розміром 256 Кбайт і вказує на блок адреси, якщо ділянка файла перебуває в кеші, у протилежному випадку він містить нульовий покажчик.

У разі спроби доступу до файла менеджер кеша визначає за зсувом, якому блоку адреси відповідає запит. Якщо відповідний елемент масиву нульовий, відсилають запит

драйверу файлової системи на читання відповідного фрагмента файла із диска в кеш, після чого копіюють дані з кеша у буфер застосування користувача.

Для підвищення продуктивності менеджер кеша відстежує історію трьох попередніх запитів, і коли він може знайти в них закономірність, буде виконано випереджувальне читання на підставі цієї закономірності (наприклад, якщо задають послідовність читання файла, випереджувальне читання зберігатиме в кеші додаткові блоки, розташовані у напрямку читання). Обсяг випереджувального читання фіксований і становить 192 Кбайт.

Зазначимо, що файл завжди відображають у пам'ять тільки один раз незалежно від того, скільки процесів до нього звертаються. При цьому сторінки із кеша копіюватимуться у буфер кожного процесу, так може бути забезпечена несуперечливість відображення файла для різних процесів.

9.7. Системний реєстр Windows XP

Реєстр – це ієрархічно організоване сховище інформації про налаштування системи і прикладних програм. Крім цього, реєстр використовують для перегляду даних про поточний стан системи.

Незважаючи на те, що на фізичному рівні реєстр не є файловою системою, його доцільно розглянути саме у цій лекції, оскільки на логічному рівні він дуже подібний до файлової системи. Крім того, реєстр зберігають у файлах на диску.

Важливість реєстру зумовлена тим, що в ньому міститься інформація, необхідна для завантаження і функціонування системи. Втрата або некоректна зміна даних реєстру можуть спричинити непрацездатність системи.

На логічному рівні реєстр можна розглядати як ієрархічну файлову систему із кількома кореневими каталогами. Аналогом каталогів у цьому разі є *ключі* (keys), аналогом файлів − *значення* (values). Ключі характеризуються іменами і містять значення або інші ключі. Кожне значення характеризується іменем, типом і даними, які воно містить. Найпоширенішими типами значень є REG_SZ – текстовий рядок, REG_DWORD − ціле число розміром 4 байти, REG_BINARY − двійкові дані довільної довжини. Крім цього, можливі посилання на інші значення або ключі (ці посилання аналогічні до символічних зв'язків файлових систем).

Як і у файловій системі, кожне значення характеризується повним шляхом, що включає всі імена ключів, розташованих над ним.

Розглянемо кореневі каталоги реєстру (ключі верхнього рівня). Найважливішими з них є HKEY_LOCAL_MACHINE (скорочено HKLM) і HKEY_USERS (HKU). Саме ці ключі відповідають фізичним даним реєстру. Ключ HKLM містить інформацію про всю систему, HKU − дані окремих користувачів.

Підмножину дерева ключів, починаючи із ключа другого рівня, називають вуликом (hive). Під ключем HKLM розташований ряд важливих вуликів:

- HARDWARE − містить інформацію про поточну апаратну конфігурацію системи; його вміст формують динамічно і на диску не зберігають;
- SAM − база даних облікових записів, містить інформацію про імена і паролі користувачів, необхідну для реєстрації у системі;
- SOFTWARE − зберігає налаштування прикладного програмного забезпечення (звичайно підключі цього вулика називають за іменем фірми-виробника);
- SYSTEM – містить інформацію, необхідну під час запуску системи, зокрема список драйверів і служб, які необхідно завантажити, а також їхні налаштування.

В реєстрі можуть зберігатися різні значення.

- Прикладом загальносистемного налагодження є значення HKLM\SYSTEM\CurrentControlSet\Services\Cdrom\Autorun типу REG_DWORD, що може містити 0 або 1. Коли воно містить 1, вставлення нового диска у CD-привід приводить до автоматичного запуску застосування autorun.exe, якщо воно є на цьому диску.
- Прикладом засобу зберігання налагодження програмного продукту може бути ключ HKLM\SOFTWARE\Adobe\Acrobat Reader\6.0, що містить значення конфігураційних параметрів цієї версії продукту.

Ключ HKU містить профілі користувачів (налаштування їхнього робочого стола, конфігурацію застосувань тощо). Інформацію щодо кожного користувача зберігають у вулику, ім'я якого збігається з ідентифікатором безпеки (SID) цього користувача.

Інші ключі верхнього рівня відображають динамічні дані або посилання на інші ключі. Наприклад, ключ HKEY_PERFORMANCE_DATA є засобом доступу до різних поточних характеристик ОС, подібно до файлової системи /proc. А ключ HKEY_CURRENT_USER це посилання на ключ HKU\SIO-поточного_користувача і відповідає налаштуванням поточного користувача.

Більша частина реєстру зберігається на диску у файлах, що відповідають вуликам *(файлах вуликів* – hive files). Файли вуликів HKLM розташовані в підкаталозі System32\Config системного каталогу Windows. Імена цих файлів збігаються з іменами вуликів (System, Software тощо). Вулики налаштувань користувачів (HKUXSID) зберігаються як файли Documents And Settings\ім'я_Користувача\NTUSER.DAT.

Зміни файлів вуликів відбуваються за тими самими правилами, що і для журнальних файлових систем (на базі атомарних транзакцій), крім того, для вулика System автоматично підтримують резервну копію.

Win32 API надає функції, що дозволяють виконувати різні дії з реєстром.

Для ***читання*** інформації з реєстру необхідно насамперед відкрити ключ, у якому перебуває потрібне значення. Для цього використовують функцію RegOpenKeyEx():

```
HKEY hk;
RegOpenKeyEx(HKEY_LOCAL_MACHINE, // HKEY_CURRENT_USER тощо
"SYSTEM\CurrentControlSet\Services\Cdrom", 0, KEY_READ, &hk);
```

Останнім параметром ця функція приймає покажчик на змінну, в яку буде записано дескриптор ключа реєстру. Після цього необхідно отримати дані потрібного значення за допомогою функції RegQueryValueEx(), куди передають такий відкритий дескриптор:

```
DWORD vsize, autorun;
// RegOpenKeyEx(..., &hk);
RegQueryValueEx(hk, "Autorun", NULL, NULL, (LPBYTE)&autorun, &vsize);
// autorun містить 0 або 1
```

Після роботи із ключем потрібно його ***закрити*** за допомогою функції

```
RegCloseKey(hk);
```

Для створення нового ключа використовують функцію RegCreateKeyEx(), а для створення нового значення всередині ключа – RegSetValueEx(). Наведемо приклад їхнього використання:

```
char myval[] = "my new data";
HKEY hknew;
RegCreateKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\myapp", 0,
      NULL, 0, 0, NULL, &hknew, &res);
RegSetValueEx(hknew, "myval", 0, REG_SZ, (LPBYTE)myval, sizeof(myval));
RegCloseKey(hk);
```

Висновки

4 Важливою концепцією доступу до файлової системи, реалізованою в UNIX-системах, зокрема в Linux, є абстрагування інтерфейсу різних файлових систем від прикладних програм за допомогою програмного забезпечення віртуальної файлової системи (VFS). VFS дає змогу в разі використання різних файлових систем обмежитися базовим набором операцій доступу. На основі VFS можна реалізовувати доступ через інтерфейс файлової системи до даних, які за своєю природою *з* дисковими файлами не пов'язані.

4 Основною файловою системою, яку використовують у Linux, є `ext2fs`. За структурою вона подібна до системи FFS.

4- В ОС лінії Windows XP здебільшого використовують NTFS. Ця система підтримує журнал, а також шифрування і стискання даних.

Контрольні запитання та завдання

1.Опишіть можливу реалізацію системних викликів openO, read O icloseO у Linux з урахуванням інтерфейсу VFS і наявності дискового кеша.

2.Поясніть, коли використання реалізації VFS для Linux призведе до того, що:

а) два файлові дескриптори посилатимуться на один файловий об'єкт;

б) один файловий дескриптор посилатиметься на два файлових об'єкти;

в) два файлові об'єкти посилатимуться на один об'єкт елемента каталогу;

г) один файловий об'єкт буде відповідати двом елементам каталогу;

д) два елементи каталогу посилатимуться на один об'єкт індексного дескриптора;

е) один об'єкт каталогу відповідатиме двом об'єктам індексного дескриптора;

ж) об'єкт індексного дескриптора буде описувати два файли на диску;

з) об'єкт індексного дескриптора не описуватиме жодного файла на диску.

3.Опишіть, яким чином у реалізації VFS для Linux можна одержати повний шлях до каталогу за номером його індексного дескриптора.

4.Поясніть роботу наступного коду:

```
int mainO { Int fd.nl;
fd = open( "tmpfile". 0_RDWR|0_CREAT|0_TRUNC. 0644 );
unlinkC "tmpfile" ): nl = write( fd. "Hello". 5 ); }
```

Якщо в цьому коді немає помилки, поясніть, навіщо він може знадобитися.

5.Опишіть послідовність дій, що повинні виконуватися в Linux під час збереження файла на диску в текстовому редакторі. Використана файлова система `ext2fs`.

6.Як зміниться послідовність дій у попередній вправі, якщо припустити, що замість `ext2fs` використовують `ext3fs`?

7.Яким чином файлова система `ext2fs` забезпечує те, що блоки даних файлів одного каталогу на фізичному рівні розташовують близько один від одного?

8.Чи можна реалізувати підтримку жорстких і символічних зв'язків у файловій системі FAT? Якщо так, то яким чином?

9.Вилучення файла в FAT позначає всі займані ним кластери як вільні, але не очищує їхній вміст. Які при цьому можуть виникнути проблеми?

10.Розробіть застосування для Linux, що відображає тактову частоту процесора.

11.Розробіть застосування для Windows XP, що відображає розмір усіх файлів заданого каталогу до і після стискання. Ім'я каталогу вводить користувач, або його

задають у командному рядку.

12.Модифікуйте застосування із попередньої вправи так, щоб ім'я останнього переглянутого каталогу зберігалося в системному реєстрі. Воно має бути використане для перегляду, якщо під час запуску застосування ім'я каталогу не зазначене в командному рядку.