Логічна організація файлових систем

- ♦ Структури файлів і файлових систем
- ♦ Організація файлової системи
- ♦ Атрибути файлів
- ♦ Операції над файлами і каталогами
- ♦ Міжпроцесова взаємодія у файловій системі

Файл — це набір даних, до якого можна звертатися за іменем. Файли організованіу *файлові системи.*

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).

Типи файлів

- спеціальні файли, що їх операційна система інтерпретує особливим чином;
- виконувані файли;
- *файли із прямим і послідовним доступом.*

Організація інформації у файловій системі

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.

Каталог — це об'єкт (найчастіше реалізований як спеціальний файл), що містить інформацію про набір файлів.

Зв'язки

Жорсткі зв'язки

За підтримки жорстких зв'язків (hard links) для файла допускається кілька імен. Усі жорсткі зв'язки визначають одні й ті самі дані на диску.

Підтримка жорстких зв'язків у POSIX

Для створення жорстких зв'язків у POSIX призначений системний виклик link(). Першим параметром він приймає ім'я вихідного файла, другим — ім'я жорсткого зв'язку, що буде створений:

```
#include <unistd.h>   // для стандартних файлових операцій P0SIX
link ("myfile.txt",  "myfile-hardlink.txt");
```

Замість системного

виклику вилучення файла використовують виклик вилучення зв'язку (який зазвичай називають unlink()), що вилучатиме один жорсткий зв'язок для заданого файла. Якщо після цього зв'язків у файла більше не залишається, його дані також вилучаються.

```
// вилучити файл, якщо в нього був один жорсткий зв'язок
unlink('myfile.txt");
```

Підтримка жорстких зв'язків у Windows

Для створення жорсткого зв'язку:

```
CreateHardLink(“myfile_hardlmk.txt”,  "myfile.txt", 0);
```

Для вилучення жорстких зв'язків у Win32 API використовують функцію **DeleteFile():**

```
DeleteFile("myfile_hardlink.txt");
```

Жорсткі зв'язки мають певні недоліки, які обмежують їх застосування:

- не можуть бути задані для каталогів;
- усі жорсткі зв'язки одного файла завжди мають перебувати на одному й тому самому розділі жорсткого диска (в одній файловій системі);
- вилучення жорсткого зв'язку потенційно може спричинити втрати даних файла.

Символічні зв'язки

Символічний зв'язок (symbolic link) — зв'язок, фізично відокремлений від даних, на які вказує. Фактично, це спеціальний файл, що містить ім'я файла, на який вказує. Властивості символічних зв'язків:

- Через такий зв'язок здійснюють доступ до вихідного файла.
- При вилученні зв'язку, вихідний файл не зникне.
- Якщо вихідний файл перемістити або вилучити, зв'язок розірветься, і доступ через нього стане неможливий, якщо файл потім поновити на тому самому місці, зв'язком знову можна користуватися.
- Символічні зв'язки можуть вказувати на каталоги і файли, що перебувають на інших файлових системах (на іншому розділі жорсткого диска).

Підтримка символічних зв'язків на рівні системних викликів

Для задання символічного зв'язку у POSIX визначено системний виклик symlink(), параметри якого аналогічні до параметрів link():

```
symlink ("myfile.txt".  "myfile-symlink.txt");
```

Для отримання шляху до файла або каталогу, на який вказує символічний зв'язок, використовують системний виклик readlink().

```
// PATH_MAX - константа, що задає максимальну довжину шляху
char filepath[PATH_MAX+1];
readlink("myfile-symlink.txt", filepath, sizeof(filepath));
// у filepath буде шлях до myfile txt
```

Операції над файлами і каталогами

Загальні відомості про файлові операції

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

- Відкриття файла.
- Закриття файла.
- Створення файла.
- Вилучення файла.
- Читання з файла.
- Записування у файл.
- Переміщення покажчика поточної позиції.
- Отримання і задання атрибутів файла.

Файлові операції POSIX

Відкриття і створення файлів

Системний виклик open():

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

Повертає цілочислове значення — *файловий дескриптор.* Його слід використовувати в усіх викликах, яким потрібен відкритий файл.

У разі помилки поверне -1, а значення змінної errno відповідатиме коду помилки.

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

♦ O_RDONLY, 0_WR0NLY, 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);

♦ O_CREAT — якщо файл із таким ім'ям відсутній, його буде створено, якщо файл є і увімкнуто прапорець O_EXCL, буде повернено помилку;

♦ O_TRUNC — якщо файл відкривають для записування, його довжину покладають рівною нулю;

♦ O_NONBLOCK — задає *неблокувальне введення-виведення*.

Параметр mode потрібно задавати тільки тоді, коли задано прапорець O_CREAT. Значенням у цьому випадку буде вісімкове число, що задає права доступу до файла.

Приклад використання цього системного виклику:

```
// відкриття файла для записування
int fdl = open("./myfile.txt", O_WRONLY |O_CREAT |O_TRUNC, 0644);
// відкриття файла для читання, помилка, якщо файла немає
int fdl = open("./myfile.txt", O_RD0NLY);
```

Закриття файла

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

```
close(fdl);
```

Читання і записування даних

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

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

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

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", O_RDONLY);
  if (infile == -1)  {
    printf ("помилка під час відкриття файла\n"); exit(-1);
}
  // створення результуючого файла, перевірку помилок пропущено
  outfile = open("outfile.txt", O_WRONLY|O_CREAT|O_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);
```

Переміщення покажчика поточної позиції у файлі

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.

Збирання інформації про атрибути файла

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):

- 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).

Приклад відображення інформації про атрибути файла:

```
struct stat attrs;
stat("myfile", &attrs);
  if (attrs.st_mode & 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);
```

Файлові операції Win32 API

Відкриття і створення файлів

```
HANDLE CreateFile (LPCTSTR fname, DWORD amode, DWORD smode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,DWORD cmode, DWORD flags, HANDLE
hTemplateFile);
```

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, у протилежному випадку створити новий.

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

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);
```

Закриття файлів

Для закриття дескриптора файла застосовують функцію CloseHandle:

```
CloseHandle(infile);
```

Читання і записування даних

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 );
```

Приклад реалізації копіювання файлів за допомогою засобів Win32 API:

```
  char buf[1024]; DWORD bytes_read, bytes_wntten;
  HANDLE infile = CreateFileCinfile.txt", GENERIC_READ, 0, 0,
OPEN_EXISTING, 0, 0);
if (infile == INVALID_HANDLE_VALUE) {
   printf("Помилка під час відкриття файла\n"); exit(-l);
  }
  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);
  C1oseHandle(infile);
  CloseHandle(outfile);
```

Прямий доступ до файла

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

Константи режиму переміщення визначені як FILE_BEGIN (аналогічно до SEEK_SET), FILE_CURRENT (аналогічно до SEEK_CUR) і FILE_END (аналогічно до SEEK_END).

Збирання інформації про атрибути файла

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

Поля структури WIN32_FILE_ATRIBUTE_DATA :

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

Операції над каталогами

- **Створення нового каталогу.**
- **Вилучення каталогу.**
- **Відкриття і закриття каталогу.**
- **Читання елемента каталогу.**
- **Перехід у початок каталогу.**

Робота з каталогами в POSIX

Для створення каталогу 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 повертає 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 rewindir(). 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);
```

Робота з каталогами у Win32 API

Для створення каталогу 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), у ньому допустимі символи шаблону «*» і «?». Параметр fattrs - це покажчик на структуру, що буде заповнена інформацією про знайдений файл. Структура подібна до WIN32_FILL_ATTRIBUTE_DATA, але в ній додатково зберігають ім'я файла (поле cFileName).

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. 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);
```