МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

МЕТОДИЧНІ ВКАЗІВКИ

до виконанні лабораторних робіт з курсу
«Операційні системи»

(Частина друга)

для студентів денної форми навчання.
Напрям підготовки – 050101 Комп’ютерні науки,
рівень підготовки “бакалавр”

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

МЕТОДИЧНІ ВКАЗІВКИ

до виконанні лабораторних робіт з курсу
«Операційні системи»

(Частина друга)

для студентів денної форми навчання.
Напрям підготовки – 050101 Комп'ютерні науки,
рівень підготовки "бакалавр"

ЗАТВЕРДЖЕНО

на засіданні методичної комісії
факультету комп'ютерних наук

Протокол № __ від "__" _____ 2014 р.

Методичні вказівки до виконанні лабораторних робіт з курсу «Операційні системи» (частина друга) для студентів денної форми навчання. Напрям підготовки – 050101 Комп'ютерні науки, рівень підготовки "бакалавр". Вказівки призначені для надання допомоги зі створення і компілювання програм мовою C/C++, при виконанні лабораторних робіт в операційних системах Windows та Linux. / Рольщиков В.Б. Одеса, ОДЕКУ, 2014, 157 с., укр. мова.

Укладач ст. викл. каф. інформаційних технологій Рольщиков В.Б.

ЗМІСТ

ЛАБОРАТОРНА РОБОТА № 2
ФАЙЛИ Й ФАЙЛОВІ СИСТЕМИ ОС LINUX І WINDOWS

Література до роботи: [1] стор. 253 – 335, [2] стор. 14 – 65,
[3] стор. 424 – 502, [4] стор. 442 – 557

МЕТА РОБОТИ

Вивчити будову файлових систем ОС Linux і Windows, навчитися за допомогою системних викликів Linux і функцій Win32 API створювати дерево каталогів, включати в нього файли різних типів, використовувати стандартні пристрої вводу-виводу.

2.1 Огляд теоретичної частини

2.1.1 Робота з файлами в операційній системі Linux

У цьому розділі розглядаються файли й каталоги ОС Linux і способи роботи з ними. Описано створення файлів, відкриття, читання їх, запис у них і видалення них. Також описано, як у програмах обробляти каталоги, наприклад, створювати, переглядати й видаляти.

Крім того коротко розглядаються поняття, пов'язані з файлами, каталогами й пристроями. Для керування файлами й каталогами виконуються системні виклики (аналог Windows API у системах UNIX і Linux), але, для забезпечення більш ефективного управління файлами й сумісності на рівні вихідного коду програм, існує великий набір бібліотечних функцій стандартної бібліотеки вводу/виводу stdio.

2.1.1.1 Файли і каталоги в Linux

У середовищі Linux, як і у UNIX, файли – особливо важливі поняття, оскільки вони забезпечують простий і узгоджений взаємозв'язок зі службами опера-

ційної системи й пристроями. Автор однієї з книг по програмуванню в Linux жартує – «В ОС Linux *файл – це все що завгодно.* Ну, або майже все!». Дійсно, в ОС Linux майже все представлено у вигляді файлів або може бути доступно за допомогою спеціальних файлів. Досить сказати, що навіть процеси в системі організовані у вигляді окремої частини файлової системи – каталогу procs. І основна ідея зберігається навіть, незважаючи на те, що в силу необхідності існують невеликі відмінності від традиційних файлів.

Це означає, що в основному програми можуть обробляти дискові файли, послідовні порти, принтери й інші пристрої точно так само, як вони використовують файли. При цьому найчастіше застосовуються п'ять базових функцій: open, close, read, write і ioctl. Каталоги – також спеціальний тип файлів. У сучасних версіях UNIX, включаючи Linux, навіть суперкористувач не робить безпосередній запис до них. Звичайно всі користувачі для читання каталогів застосовують інтерфейс opendir/readdir, і немає потреби знати подробиці реалізації каталогів у системі.

Крім умісту файли характеризуються ім'ям і набором властивостей, або «адміністративної інформації» – дата створення/модифікації файла й права доступу до нього. Властивості зберігаються у ***файловому індексі*** (inode), спеціальному блоці даних файлової системи, що також містить відомості про довжину файла й місце зберігання файла на диску. При роботі система використовує номер файлового індексу.

Каталог – це файл, що містить номери індексів і імена інших файлів. Кожний елемент каталогу – посилання на файловий індекс; при видаленні ім'я файла, віддаляється тільки посилання на нього. У системі можна створити посилання на той самий файл у різних каталогах. Тобто коли видаляється файл, видаляється елемент каталогу для цього файла, і кількість посилань на файл зменшується на одиницю. Дані файла можуть бути усе ще доступні завдяки іншим посиланням на цей же файл. Коли число посилань на файл стає рівним нулю, індекс файла й блоки даних, на які він посилається, більше не використовуються й позначаються як вільні.

Файли (точніше кажучи, записи про них) поміщаються в каталоги, які можуть містити підкаталоги. Так формується ієрархія файлової системи. Звичайно користувач зберігає файли у вихідному (/home/<ім'я_користувача>) каталозі з підкаталогами для зберігання електронної пошти, ділових листів, службових програм і т.і. Вихідні каталоги користувачів – це, як правило, підкаталоги каталогу більш високого рівня, що створюється спеціально для цієї мети, у нашім випадку це каталог /home, що у свою чергу є підкаталогом кореневого каталогу /, розташованого на верхньому рівні ієрархії й такого, що утримує всі системні файли й підкаталоги.

До файлів і пристроїв можна звертатися й управляти ними, застосовуючи невеликий набір функцій. Ці функції, відомі як ***системні виклики***, безпосередньо надаються системою UNIX (і Linux) і слугують інтерфейсом самої операційної системи.

Файли й пристрої використовуються однаково: вони можуть відкриватися, читатися, на них можна записувати і їх можна закривати. Наприклад, один і той виклик open, використовуваний для доступу до звичайного файла, застосовується для звертання до користувальницького термінала, принтеру або до стрічкового накопичувача.

До основних функцій низького рівня або системних викликів, що використовуються для звертання до драйверів пристроїв, належать наступні:

- open – відкриває файл або пристрій;
- read – читає з відкритого файла або пристрою;
- write – пише у файл або пристрій;
- close – закриває файл або пристрій;
- ioctl – передає керуючу інформацію.

Системний виклик ioctl застосовується для апаратно-залежного керування (як альтернатива стандартного вводу/виводу), тому він у кожного пристрою свій, і розгляд цього системного виклику не входить до завдання даної лабораторної роботи.

Довідкову інформацію із системних викликів можна одержати в розділі 2

інтерактивного довідкового керівництва (для цього слугує системна команда man 2 <ім'я_виклику>). Прототипи функцій зі списком параметрів, що використовуються в системних викликах і типом значення, котре повертається функцією, а також пов'язані з ними директиви #define з визначенням констант представлені у файлах include.

2.1.1.2 Бібліотечні функції

Використання системних викликів низького рівня безпосередньо для вводу й виводу може бути не дуже ефективним. Проблема полягає в тім, що, по-перше, при виконанні системних викликів ОС змушена переключатися з виконання користувальницького програмного коду на власний код ядра й потім вертатися до виконання програми. Тому бажано змушувати кожний такий виклик виконувати максимально можливий обсяг роботи, наприклад, зчитувати й записувати за один раз великі обсяги даних, а не одиночні символи. І, по-друге, у устаткування є обмеження, що накладаються на розмір блоку даних, які можуть бути зчитані або записані в будь-який конкретний момент часу й за цим треба стежити.

Для формування інтерфейсу високого рівня для пристроїв і дискових файлів дистрибутив Linux (і UNIX) надає низку стандартних бібліотек. Вони являють собою колекції функцій, які можна включати до програми. Гарним прикладом слугує стандартна бібліотека вводу/виводу, що забезпечує буферізований вивід. Але у програмах можна ефективно записувати блоки даних різних розмірів, застосовуючи бібліотечні функції, які виконують системні виклики низького рівня, постачаючи їх повними блоками. Це істотно знижує витрати системних викликів.

Бібліотечні функції, як правило, описуються в розділі 3 інтерактивного довідкового керівництва (man 3 <ім'я_функції>) й мають стандартний файл директиви include, наприклад, stdio.h для стандартної бібліотеки вводу/виводу.

На рис. 2.1 наведена умовна схема системи Linux, на якій показане як розташовані різні функції роботи з файлами відносно користувача, драйверів

пристроїв, ядра системи й устаткування.

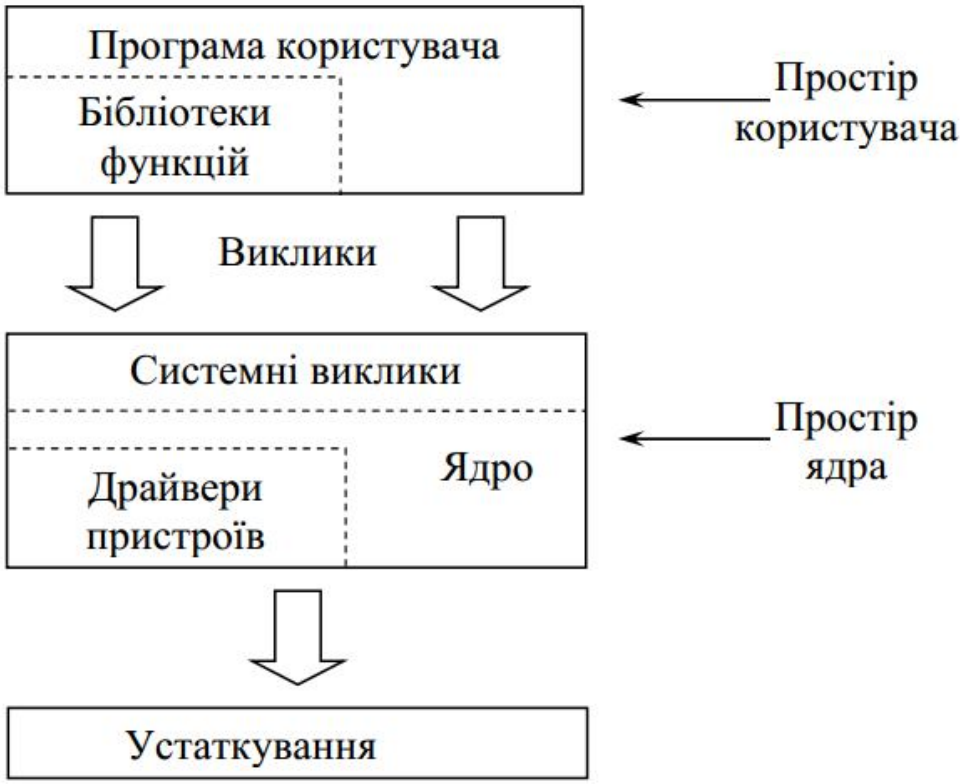

Рисунок 2.1 – Схематичне представлення використання системних викликів і бібліотечних функцій в ОС Linux

2.1.1.3 Доступ низького рівня до файлів

У лабораторній роботі ми будемо користуватися тільки системними викликами без звертання до бібліотечних функцій.

У кожної програми, що виконується, і називається ***процесом***, є ряд пов'язаних з нею ***дескрипторів файлів***. Дескриптор – ціле (small integer) число, що використовується для звертання до відкритих файлів і пристроїв. Кількість дескрипторів залежить від конфігурації системи. Коли програма запускається, у неї звичайно вже відкриті три дескриптори. До них належать наступні:

- 0 (STDIN_FILENO) – стандартний пристрій вводу;
- 1 (STDOUT_FILENO) – стандартний вивід;
- 2 (STDERR_FILENO) – стандартний потік помилок.

Дескриптори файлів, котрі відкриті автоматично, уже дозволяють створю-

вати прості програми за допомогою звернення до виклику write. З файлами й пристроями можна зв'язати інші дескриптори файлів, використовуючи системний виклик open.

write

Системний виклик write призначений для запису з buf перших nbytes байтів у файл, асоційований з дескриптором fildes. Він повертає кількість реально записаних байтів, що може бути менше nbytes, якщо в дескрипторі файла знайдена помилка або дескриптор файла, котрий розташований на більш низькому рівні драйвера пристрою, є чутливим до розміру блоку. Якщо функція повертає 0, це означає, що нічого не записане; якщо вона повертає -1, у системному виклику write виникла помилка, котра описується в глобальній змінній errno.

Далі наведений синтаксичний запис цього системного виклику.

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

Отриманих відомостей про функцію write і стандартні дескриптори цілком достатньо щоб написати першу програму, simple_write.c

```
#include <unistd.h>
#include <stdlib.h>
int main() {
  if ((write (1, "Here is some data\n", 18)) != 18)
      write (2, "A write error has occurred on file
             descriptor 1\n", 46);
  exit(0); }
```

Ця програма просто поміщає повідомлення в стандартний вивід. Коли вона завершується, всі відкриті дескриптори файлів автоматично закриваються, і їх не потрібно закривати явно. Але у випадку буферізованого виводу це не так.

Виклик write може повідомити про те, що при його завершенні записане менше байтів, чим запитувалося. Це не обов'язково помилка. У програмах для

виявлення помилок необхідно перевіряти змінну errno і, при необхідності, ще раз викликати write для запису даних, що залишилися.

read

Системний виклик read зчитує до nbytes байтів даних з файла, асоційованого з дескриптором файла fildes, і поміщає їх до області даних buf. Він повертає кількість дійсно прочитаних байтів, котра може бути менше необхідної кількості. Повернення 0 означає досягнення кінця файла, повернення -1 – помилку при виклику.

```
#include <unistd.h>
size_t read (int fildes, void *buf, size_t nbytes);
```

Програма simple_read.c копіює перші 128 байтів зі стандартного вводу на стандартний вивід, або всі данні, що були введені, якщо їх кількість менше за 128 байтів.

```
#include <unistd.h>
#include <stdlib.h>
int main() {
  char buffer[128];
  int nread;
  nread = read(0, buffer, 128);
  if (nread == -1)
    write(2, "A read error has occurred\n", 26);
  if ((write(1, buffer, nread)) != nread)
    write(2, "A write error has occurred\n", 27);
  exit(0); }
```

open, creat

Для створення дескриптора нового файла використовується системний виклик open.

```
#include <fcntl.h>
#include <sys/types.h>
```

```
#inelude <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

При виклику open стандарт POSIX не вимагає підключення файлів sys/types.h і sys/stat.h, але в деяких системах UNIX вони можуть знадобитися.

Виклик open встановлює шлях до файла або пристрою. Якщо встановлення минуло успішно, вертається дескриптор файла, що може застосовуватися в системних викликах read, write і ін. Дескриптор файла унікальний і не використовується спільно іншими процесами, які можуть у цей момент виконуватися. Якщо файл відкритий одночасно у двох програмах, вони підтримують окремі і різні дескриптори файла. Якщо вони обидві пишуть у файл, то дані однієї програми можуть бути записані поверх даних іншої.

Ім'я файла або пристрою, що відкривається, передається як параметр path; параметр oflags застосовується для зазначення дій, що вживаються при відкритті файла.

Параметр oflags задається як комбінація обов'язкового режиму доступу до файла й інших необов'язкових режимів. Список можливих значень параметра oflags наведений у табл. 2.1

Таблиця 2.1 – Константи параметра oflags у системному виклику open

Константи	Призначення параметра
	Обов'язкові режими
O_RDONLY	Відкриття тільки для читання
O_WRONLY	Відкриття тільки для запису
O_RDWR	Відкриття для читання й запису
	Необов'язкові режими
O_APPEND	Поміщає записувані дані в кінець файла
O_TRUNC	Задає нульову довжину файла, відкидаючи існуючий вміст
O_CREAT	При необхідності створює файл із правами доступу, заданими в параметрі mode
O_EXCL	Застосовується з режимом O_CREAT і гарантує, що викликаюча програма створить файл

Виклик open – ***атомарний***, тобто він виконується тільки цілком за один

виклик функції. Це запобігає створенню файла з тим самим дескриптором двома програмами одночасно.

Всі можливі значення параметра oflags описані на сторінці інтерактивного довідкового керівництва, котра присвячена open (команда man 2 open).

У випадку успішного завершення виклик open повертає новий дескриптор файла – позитивне ціле число або -1 у випадку невдачі. У випадку невдачі open також встановлює глобальну змінну errno, щоб показати причину невдачі. У нового дескриптора файла завжди найменший невикористаний номер дескриптора, іноді це дуже корисно. Наприклад, якщо програма закриває свій стандартний вивід, а потім знову викликає open, то повторно використовується дескриптор з номером 1 і стандартний вивід перенаправляється в інший файл або на інший пристрій.

Стандарт POSIX визначає також системний виклик creat, але він застосовується рідко. Виклик не тільки створює файл, але й відкриває його. Виклик є еквівалентний виклику open з параметром oflags = O_CREAT | O_WRONLY | O_TRUNC.

Кількість файлів, одночасно відкритих у будь-якій програмі, що виконується, обмежена значенням константи OPEN_MAX у файлі limits.h і змінюється від системи до системи, але стандарт POSIX вимагає, щоб воно було не менше за 16. В ОС Linux це граничне значення можна змінювати під час виконання програми й OPEN_MAX вже не є константою, а її початкове значення дорівнює 256.

close

Системний виклик close застосовується для розривання зв'язку файлового дескриптора fildes з його файлом. Дескриптор файла після цього може використовуватися повторно. При успішному завершенні виклик повертає 0 і -1 при помилці.

```
#include <unistd.h>
int close (int fildes);
```

Коли файл створюється із застосуванням прапора O_CREAT, то в системному виклику open необхідно використовувати форму з трьома параметрами. Третій параметр mode формується із прапорів, визначених у заголовному файлі sys/stat.h (див. табл. 2.2) і з'єднаних порозрядною операцією OR. Наприклад, виклик

```
open ("myfile", O_CREAT, S_IRUSR | S_IXOTH);
```

у результаті призведе до створення файла з ім'ям myfile і тільки із правами на читання для власника й на виконання для інших. Задані таким чином права встановлюються при створенні файла. Крім того, на права доступу до створюваного файла оказує вплив маска користувача, що задається командою командної оболонки umask. При цьому значення параметра mode у виклику open поєднується по AND з інвертованою маскою користувача вже на етапі виконання програми. Тобто надання запитаних прав, залежить від значення umask (системної змінної, в котрій міститься маска для прав доступу до файла) під час виконання.

Таблиця 2.2 – Прапори параметра mode для установки прав доступу

Константи	Призначення параметра
S_IRUSR	Право на читання, власник
S_IWUSR	Право на запис, власник
S_IXUSR	Право на виконання, власник
S_IRGRP	Право на читання, група
S_IWGRP	Право на запис, група
S_IXGRP	Право на виконання, група
S_IROTH	Право на читання, інші
S_IWOTH	Право на запис, інші
S_IXOTH	Право на виконання, інші

2.1.1.4 Інші системні виклики для керування файлами

Існує ряд інших системних викликів, що оперують дескрипторами файлів низького рівня. Вони дозволяють програмі контролювати використання файла,

повертаючи інформацію про його стан.

lseek

Системний виклик lseek задає покажчик поточної позиції читання/запису файла, тобто він застосовується для зазначення місця, з якого буде відбуватися наступне зчитування або на яке буде провадитися наступний запис. Можна задати покажчик на абсолютну позицію у файлі або позицію відносно до поточного положення покажчика або відносно до кінця файла.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Параметр offset застосовується для зазначення позиції, а параметр whence визначає спосіб застосування offset. Можливі значення параметра наведені в табл. 2.3.

Таблиця 2.3 – Значення параметра whence у системному виклику lseek

Константи	Призначення параметра
SEEK_SET	Абсолютна позиція у файлі
SEEK_CUR	Позиція відносно поточної
SEEK_END	Позиція відносно кінця файла

При вдалому завершенні виклик lseek повертає величину зсуву відносно початку файла або -1 у противному випадку. Тип даних off_t залежить від реалізації системи й визначається у файлі sys/types.h.

fstat, stat і lstat

Ці системні виклики призначені для повернення інформації про стан файла, асоційованого з відкритим дескриптором файла. Інформація записується в структуру buf.

Синтаксичний запис викликів має вигляд:

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Функції stat і lstat працюють із файлами відкритими не по файловому дескриптору, а по символічному посиланню й повертають інформацію про стан такого файла. Виклик lstat повертає дані про саме посилання, а виклик stat – про файл, на який посилання вказує.

Елементи викликуваної структури stat можуть бути різними в різних UNIX-подібних системах. У табл. 2.4 наведені такі поля структури, що є обов'язковими для всіх операційних систем.

Таблиця 2.4 – Обов'язкові поля структури stat

Поле	Призначення поля
st_mode	Права доступу до файла й відомості про тип файла
st_ino	Індекс, асоційований з файлом
st_dev	Пристрій, на якому розміщений файл
st_uid	Ідентифікатор (user identity) власника файла
st_gid	Ідентифікатор групи (group identity) власника файла
st_atime	Час останнього звертання
st_ctime	Час останньої зміни прав доступу, власника, групи або об'єм
st_mtime	Час останньої модифікації вмісту
st_nlink	Кількість твердих посилань на файл

У прапорів st_mode, що повертаються в структурі stat, у заголовному файлі sys/stat.h є ряд асоційованих макросів (див. табл. 2.5). У ці макроси включені імена прапорів для прав доступу й типів файлів і деякі маски, що допомагають перевіряти специфічні типи й права.

Прапори для прав доступу визначаються тими ж константами, що й при створенні файла.

Таблиця 2.5 Макроси прапорів поля st_mode структури stat

Мнемонічне ім'я	Призначення прапорця, маски або макроса
	Прапори для типів файла:
S_IFBLK	Блоковий пристрій
S_IFDIR	Каталог
S_IFCHR	Символьний пристрій
S_IFIFO	FIFO (іменований канал)
S_IFREG	Звичайний файл
S_IFLNK	Символічне посилання
	Інші прапори режимів файла
S_ISUID	Елемент одержує setUID при виконанні
S_ISGUID	Елемент одержує setGID при виконанні
	Маски, що інтерпретують прапори поля st_mode
S_IFMT	Тип файла
S_IRWXU	Права користувача на читання/запис/виконання
S_IRWXG	Права групи на читання/запис/виконання
S_IRWXO	Права інших на читання/запис/виконання
	Макроси для визначення типу файла
S_ISBLK	Перевірка для блокового файла
S_ISCHR	Перевірка для символьного файла
S_ISDIR	Перевірка для каталогу
S_ISFIFO	Перевірка для FIFO
S_ISREG	Перевірка для звичайного файла
S_ISLNK	Перевірка для символічного посилання

Макроси для визначення типу файла допомагають визначити типи файлів, тобто належним чином порівнюють установлені прапори режиму файла з відповідним прапором, типу пристрою.

Наприклад, для перевірки того, що файл не є каталогом і в нього є права на виконання тільки для власника й більше ніяких інших прав, можна скористатися наступним тестом:

```
struct stat statbuf;
mode_t modes;
stat("filename", &statbuf);
modes = statbuf.st_mode;
if (!S_ISDIR(modes) && (modes & S_IRWXU) = S_IXUSR)
        ...
```

dup, dup2

Ці системні виклики дозволяють дублювати дескриптор файла, надаючи два або кілька різних дескрипторів, які звертаються до того самого файла. Ця можливість може застосовуватися для читання й запису в різні частини файла (наприклад, при закачуванні файла з Інтернету в кілька потоків).

Синтаксичний запис для викликів такий:

```
#include <unistd.h>
int dup(int tildes);
int dup2(int fildes, int fildes2);
```

Також ці виклики корисні, коли декілька процесів, взаємодіють через іменовані канали.

2.1.1.5 Приклад програми копіювання файла за допомогою системних викликів

Системні виклики дозволяють копіювати файл блоками доволі великого розміру, що дозволяє домогтися високої швидкості копіювання. Найпростіша програма copy_block.c, що працює із блоками 1 КіБ може мати такий вигляд:

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main(){
  char block[1024];
  int in, out;
  int nread;
  in = open("file.in". O_RDONLY);
  out = open("file.out". O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
  while((nread = read(in, block, sizeof(block))) > 0)
    write(out, block, nread);
  exit(0);}
```

У прикладі не передбачені перевірки результатів виконання системних

викликів, які в справжній програмі обов'язково повинні виконуватися.

2.1.1.6 Ведення файлів і каталогів

На додаток роботи з файлами, системні виклики разом з функціями зі стандартної бібліотеки вводу/виводу С дозволяють забезпечити повний контроль над створенням і веденням файлів і каталогів у файловій системі.

chmod

Права доступу до файла або каталогу після їхнього завдання при відкритті файла можна змінити за допомогою системного виклику chmod.

Синтаксичний запис виклику має вигляд:

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

Права доступу до файла, ім'я якого задається параметром path, змінюються у відповідності зі значенням параметра mode як у системному виклику open (див. вище). Якщо програма не має достатніх повноважень, то тільки власник файла й суперкористувач системи мають можливість змінювати права доступу до файла.

chown

Крім зміни прав доступу до файла суперкористувач може змінити і його власника. Досягається це використанням системного виклику chown з таким прототипом:

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

Числові значення ідентифікаторів (ID) нового користувача й групи можна

одержати із системних викликів getuid і getgid. Власник і група файла змінюються, тільки якщо задані відповідні повноваження.

unlink, link і *symlink*

Ці системні виклики використовуються для включення файла в який-небудь каталог, його видалення або створення символічного посилання на файл. Їхні прототипи указані нижче:

```
#include <unistd.h>
int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

Системний виклик unlink застосовується для видалення файла. Точніше сказати при використанні цього виклику видаляється тільки запис про файл у каталозі й зменшується на одиницю лічильник посилань на файл. Виклик повертає значення 0, якщо видалення посилання пройшло успішно, і -1 у випадку помилки. Для виконання виклику необхідно мати права на запис і виконання в каталозі, що зберігає посилання.

Якщо лічильник стає рівним нулю й файл не є відкритим у жодному процесі, він віддаляється повністю. У дійсності елемент каталогу завжди видаляється негайно, а місце, зайняте вмістом файла, не очищується доти, поки останній із процесів, що володіють файлом, не закриє його.

Навпаки системний виклик link створює нове жорстке посилання на існуючий файл path1 і приєднує його до каталогу path2. Аналогічно символічні посилання створюються за допомогою системного виклику symlink. Символічні посилання на файл не збільшують значення лічильника посилань і не заважають видаленню файла.

mkdir і *rmdir*

Системні виклики mkdir і rmdir призначені для створення й видалення ка-

талогів.

Системний виклик mkdir із прототипом

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

використовується для створення каталогів. При його виклику формується новий каталог з ім'ям, зазначеним у параметрі path. Права доступу до каталогу передаються в параметрі mode аналогічно такому ж параметру системного виклику open і так само залежать від змінної umask.

```
#include <unistd.h>
int rmdir(const char *path);
```

Системний виклик rmdir, із указаним прототипом, видаляє каталоги за умови, що вони порожні. Значення, що повертаються, 0 і -1 є стандартними.

chdir і *getcwd*

Програма може змінювати поточний каталог аналогічно тому, як користувач виконує зміну каталогу у файловій системі, застосовуючи команду оболонки cd.

```
#include <unistd.h>
int chdir(const char *path);
```

Якщо в програмі необхідно визначити поточний робочий каталог, то це можна зробити шляхом системного виклику getcwd.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Функція getcwd записує ім'я поточного каталогу в заданий буфер buf. Во-

на повертає NULL, якщо ім'я каталогу перевищить розмір буфера (помилка ERANGE), котрій задається у параметрі size. У випадку успішного завершення вона повертає buf. Функція getcwd може також повернути значення NULL, якщо під час виконання програми каталог вже був видаленим (EINVAL) або змінилися його права доступу (EACCESS).

2.1.1.7 Перегляд каталогів

Досить часто в програмах виникає задача знаходження потрібного файла у файловій системі. Для цього можна, наприклад, відкрити будь-який каталог як звичайний файл і безпосередньо зчитати його елементи, але різні структури і реалізації файлових систем зробили цей підхід таким, що не може бути перенесеним з машини на машину. Тому був розроблений стандартний комплект бібліотечних функцій, що істотно спрощує перегляд каталогів.

Функції роботи з каталогами оголошені в заголовному файлі dirent.h. У них використовується структура DIR як основа обробки каталогів. Покажчик на цю структуру, називаний ***потоком каталогу*** (DIR*), діє багато в чому так само, як діє потік файла (FILE*) при роботі зі звичайним файлом, за допомогою стандартної бібліотеки вводу/виводу. Оскільки досить не бажано безпосередньо змінювати поля в структурі DIR, елементи каталогу вертаються в структурі dirent, що також оголошена у файлі dirent.h.

При роботі з каталогами найчастіше використовуються наступні функції:

- opendir, closedir;
- readdir;
- telldir;
- seekdir;
- closedir.

opendir

Функція opendir відкриває каталог і формує його потік. Якщо вона завер-

шується успішно, то повертає покажчик на структуру DIR, котра використовується для читання елементів каталогу. У противному випадку вертається порожній покажчик (NULL). Оголошення функції наступне:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

readdir

Успішний виклик функції readdir із прототипом

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

повертає покажчик на структуру dirent, що містить наступний елемент каталогу. Переданий у функцію параметр dirp указує на структуру відкритого каталогу – DIR. Наприкінці каталогу readdir повертає NULL. Це ж значення вертається й при виникненні помилки, але стандарт POSIX у цьому випадку вимагає встановлення нового значення змінної errno.

У полі d_ino типу ino_t у структурі dirent утримується такий реквізит елемента каталогу, як індекс файла (inod), а в полі d_name[ ] типу char – ім'я файла. Інші реквізити файла в каталозі визначаються стандартним образом за допомогою системного виклику stat.

Перегляд каталогу за допомогою функції readdir не гарантує формування списку всіх файлів (і підкаталогів) у каталозі, якщо в цей час виконуються інші процеси, що створюють і видаляють файли в каталозі.

telldir

Функція telldir повертає значення, що реєструє поточну позицію в потоці каталогу. Прототип її наступний:

```
#include <sys/types.h>
#include <dirent.h>
long int telldir(DIR *dirp);
```

seekdir

Функція seekdir встановлює покажчик на елемент каталогу в потоці каталогу, котрий задається в параметрі dirp. Значення параметра loc, що застосовується для установки позиції, необхідно попередньо одержати з виклику функції telldir.

```
#inelude <sys/types.h>
#inelude <dirent.h>
void seekdir (DIR *dirp, long int loc);
```

closedir

Нарешті, функція closedir закриває потік каталогу й звільняє ресурси, що йому виділені. Як і завжди, вона повертає 0 у випадку успіху й -1 при наявності помилки.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

Для демонстрації використання описаних функцій нижче наведена програма printdir.c. Вона створює простий перелік умісту каталогу. Кожний файл представляється окремим рядком. Ім'я кожного підкаталогу завершується символом косої риси, а файли, що втримуються в підкаталозі, виводяться з відступом шириною в чотири пробіли.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
```

```
#include <stdlib.h>
void printdir(char *dir, int depth) {
  DIR *dp;
  struct dirent *entry;
  struct stat statbuf;
  if ((dp = opendir(dir)) == NULL) {
    fprintf(stderr, "cannot open directory: %s\n", dir);
    return; }
  chdir(dir);
  while((entry = readdir(dp)) != NULL) {
    lstat(entry->d_name, &statbuf);
    if (S_ISDIR(statbuf.st_mode)) {
/* Знаходить каталог, але ігнорує каталоги з ім'ям . і .. */
      if (strcmp(".",entry->d_name) == 0
          || strcmp("..",entry->d_name) == 0) continue;
      printf("%*s%s/\n", depth, " ", entry->d_name);
/* Рекурсивний виклик з новим відступом */
      printdir(entry->d_name, depth+4);}
    else printf("%*s%s\n", depth, " ", entry->d_name); }
  chdir("..");
  closedir(dp); }

int main() {
/* Огляд каталогу /home */
  printf("Directory scan of /home:\n");
  printdir("/home", 0);
  printf("done.\n");
  exit(0); }
```

2.1.1.8 Обробка помилок

Як вказувалося вище, багато системних викликів і функції вказують причину збою операції, яка щойно виконувалася, встановлюючи значення зовнішньої змінної errno. Ця змінна використовується як стандартний спосіб оповіщення про проблеми, що виниклі. Програми повинні перевіряти значення змінної errno відразу ж після виникнення проблеми у функції.

Імена констант і варіанти помилок перераховані в заголовному файлі errno.h і наведені в табл. 2.6.

Таблиця 2.6 – Символічні імена констант помилок і їхня розшифровка

Мнемонічне ім'я	Зміст помилки
EPERM	Operation not permitted (Операція не дозволена)
ENOENT	No such file or directory (Немає такого файла або каталогу)
EINTR	Interrupted system call (Перерваний системний виклик)
EIO	I/O Error (Помилка вводу/виводу)
EBUSY	Device or resource busy (Пристрій або ресурс зайнятий)
EEXIST	File exists (Файл існує)
EINVAL	Invalid argument (Невірний аргумент)
EMFILE	Too many open files (Занадто багато відкритих файлів)
ENODEV	No such device (Немає такого пристрою)
EISDIR	Is a directory (Це каталог)
ENOTDIR	Isn't a directory (Це не каталог)

У бібліотеках мови С є дві дуже корисних функцій strerror і perror, що оперують зі змінною errno і повідомляють про помилки у разі їх виникнення при виконанні різних системних викликів та бібліотечних функцій.

Перша функція strerror, що має наступний синтаксичний запис

```
#include <string.h>
char *strerror(int errnum);
```

перетворює номер помилки в рядок, що описує тип виниклої помилки. Вона може бути корисна для реєстрації умов, що викликають помилку.

Прототип функції perror оголошується у заголовному файлі stdio.h і має вигляд:

```
#include <stdio.h>
void perror(const char *s);
```

Функція перетворює поточну помилку зі змінної errno, у рядок і виводить його до стандартного потоку помилок. Строковому повідомленню про помилку передує повідомлення, котре задається в рядку s (якщо покажчик не дорівнює NULL), за яким ідуть двокрапка й пробіл.

2.1.2 Робота з файлами і каталогами в системі Windows

На підключених пристроях Windows підтримує файлові системи чотирьох типів, але тільки одна з них рекомендується компанією Microsoft для використання в якості основної. Це файлова система NTFS (New Technology File System) – одна із кращих сучасних файлових систем, що підтримує довгі імена файлів, безпеку, стійкість до збоїв, шифрування, стиснення файлів, розширені атрибути, і дозволяє працювати з дуже великими файлами й обсягами даних.

Доступ до файлових систем будь-якого типу в Windows здійснюється однаковим образом, іноді з деякими обмеженнями. Наприклад, підтримка захисту файлів забезпечується тільки в NTFS.

Windows як і Linux підтримує ієрархічну систему імен файлів, але угоди її трохи відрізняються від угод, що є звичними для користувачів Linux. Основна відмінність полягає в тому, що повне ім'я файла починається із зазначення буквеного імені диска. Крім того, існує й інший можливий варіант завдання повного шляху доступу – використання універсального кодування імен (Universal Naming Code, UNC). У цьому випадку перша частина повного шляху доступу в цьому випадку буде мати вигляд: \\servername\sharename.

При зазначенні повного шляху доступу як роздільник звичайно використовується символ зворотної косої риси \, але в параметрах API для цієї мети можна використовувати й символ прямої косої риси /.

На відміну від Linux, рядкові й прописні букви в іменах каталогів і файлів не різняться між собою, тобто імена не чутливі до регістра (case-insensitive), але в той же час вони запам'ятовують регістр (case-retaining).

Одиночний символ крапки **.** і два символи крапки **..** як і в Linux використовуються як імена поточного каталогу і каталогу, що є його батьком, відповідно.

2.1.2.1 Операції відкриття, читання, записи й закриття файлів

CreateFile

Для створення нових і для відкриття існуючих файлів в Windows використовується функція API (Application Programming Interface) CreateFile. Надалі, при описі всіх функцій API, спочатку буде приводитися прототип, а потім описуватися її параметри й порядок використання.

Прототип функції має вигляд:

```
HANDLE CreateFile(LPCTSTR lpName, DWORD dwAccess,
  DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreate, DWORD dwAttrsAndFlags, HANDLE hTemplateFile)
```

У випадку успішного виконання, функція повертає дескриптор відкритого файла (в Windows, на відміну від Linux, він має тип HANDLE, а не int). При помилці вертається INVALID_HANDLE_VALUE.

Як видно із прототипу функція має набагато більшу кількість параметрів, чим системний виклик Linux open.

В Windows для імен параметрів існують деякі угоди. Вони стосуються префіксів параметрів. Так префікс dw використовується в іменах параметрів з типом DWORD. У таких параметрах зберігаються набори прапорів або деякі числові значення. Префікс lp (довгий покажчик), використовується для рядків, що містять шляхи доступу, або інших значень типу покажчик.

Параметр lpName – покажчик на рядок символів, що містить ім'я файла або каналу, який потрібно відкрити або створити й має тип LPCTSTR. Використання цього типу пояснюється тим, що в API Windows замість звичайного типу char для символів і символьних рядків використовується тип TCHAR. Крім того існує визначений тип LPTSTR, що є покажчиком на узагальнену строкову змінну й тип LPCTSTR – покажчик на узагальнену строкову константу.

Параметр dwAccess – визначає тип доступу до файла – читання або запис, що відповідно вказується прапорами GENERIC_READ і GENERIC_WRITE. Для одержання доступу до файла, як по читанню, так і по запису ці прапори мо-

жна поєднувати операцією порозрядного OR.

Параметр dwShareMode – визначає можливість спільного використання файла декількома процесами. Можливі значення параметра наведені в табл. 2.7. Як і завжди, значення можуть поєднуватися по OR

Таблиця 2.7 – Значення констант параметра dwShareMode

Мнемонічне ім'я	Призначення параметра
0	Забороняє спільне використання файла й відкриття другого дескриптора навіть у рамках того самого процесу;
FILE_SHARE_READ	Дозволено відкривати цей файл всім процесам для паралельного доступу тільки по читанню;
FILE_SHARE_WRITE	Дозвіл на паралельний запис у файл.

lpSecurityAttributes – параметр, що вказує на структуру атрибутів безпеки SECURITY_ATTRIBUTES. Якщо завдання атрибутів безпеки не передбачено, то використовується значення NULL.

Параметр dwCreate призначений для уточнення операції над файлом: створення нового файла, перезапис існуючого тощо. Можливі значення параметра (див. табл. 2.8) можуть поєднуватися за допомогою порозрядного OR.

Таблиця 2.8 – Можливі значення параметра dwCreate

Ім'я параметра	Призначення параметра
CREATE_NEW	Створити новий файл, якщо файл вже є, виконання функції завершується невдачею
CREATE_ALWAYS	Створити новий файл, існуючий файл перезаписується
OPEN_EXISTING	Відкрити існуючий файл, якщо файла немає виконання функції завершується невдачею
OPEN_ALWAYS	Відкрити файл, якщо файл не існує, він буде створений
TRUNCATE_EXISTING	Відкрити файл із розміром рівним нулю. Рівень доступу до файла у параметрі dwAccess повинен бути не нижче GENERIC_WRITE. Якщо зазначений файл існує, його вміст знищується. На відміну від випадку CREATE_NEW виконання функції буде успішним навіть у тих випадках, коли зазначений файл не існує

За допомогою параметра dwAttrsAndFlags установлюються атрибути файла й прапори. Усього є 16 прапорів і атрибутів. Атрибути характеризують саме файл, а не пов'язаний з ним дескриптор, і ігноруються, якщо відкривається існуючий файл. Деякі з найбільш важливих значень прапорів наведені в табл. 2.9.

Таблиця 2.9 – Найчастіше використовувані значення прапорів параметра dwAttrsAndFlags

Ім'я прапорця	Призначення прапорця
FILE_ATTRIBUTE_NORMAL	Цей атрибут можна використовувати лише за умови, що одночасно з ним не встановлюються ніякі інші атрибути (тоді як для всіх інших прапорів одночасне встановлення допускається)
FILE_ATTRIBUTE_READONLY	Цей атрибут забороняє додаткам здійснювати запис у даний файл або видаляти його
FILE_FLAG_DELETE_ON_CLOSE	Цей прапор корисно застосовувати у випадку тимчасових файлів. Файл буде вилучений відразу ж після закриття останнього з його відкритих дескрипторів
FILE_FLAG_OVERLAPPED	Цей прапор відіграє важливу роль при виконанні операцій асинхронного вводу/виводу
FILE_FLAG_WRITE_THROUGH	Встановлює режим наскрізного запису проміжних даних безпосередньо у файл на диску, минаючи кеш
FILE_FLAG_NO_BUFFERING	Встановлює режим відсутності проміжної буферізациії або кешуваниня, при якому обмін даними відбувається безпосередньо з буферами даних програми, зазначеними при виклику функцій ReadFile або WriteFile. Потрібно, щоб початки програмних буферів збігалися із границями секторів диска, а розміри були кратними розміру сектора. Для визначення розміру сектора можна користуватися функцією GetDiskFreeSpace

Продовження таблиці 2.9

Ім'я прапорця	Призначення прапорця
FILE_FLAG_RANDOM_ACCESS	Передбачається відкриття файла для довільного доступу; Windows буде намагатися оптимізувати кешування файла стосовно до цього типу доступу
FILE_FLAG_SEQUENTIAL_SCAN	Передбачається відкриття файла для послідовного доступу; Windows буде намагатися оптимізувати кешування файла стосовно до цього типу доступу. Обоє останніх режиму реалізуються системою лише в міру можливостей

У тих випадках, коли потрібно, щоб атрибути створюваного файла збігалися з атрибутами вже існуючого у системі файла, використовується параметр hTemplateFile – дескриптор із правами доступу GENERIC_READ до шаблона файла, що надає розширені атрибути, які будуть застосовані до створюваного файла замість атрибутів, зазначених у параметрі dwAttrsAndFlags. Звичайне значення цього параметра встановлюється таким, що дорівнює NULL. При відкритті існуючого файла параметр hTemplateFile ігнорується.

У серверних версіях Windows додатково надається функція ReOpenFile, що повертає новий дескриптор з іншими прапорами, правами доступу та іншими параметрами ніж ті, які були зазначені при первісному відкритті файла, якщо тільки це не приводить до виникнення конфлікту між новими й колишніми правами доступу.

CloseHandle

Для закриття об'єктів будь-якого типу (не тільки файлів), оголошення недійсними їхніх дескрипторів і звільнення системних ресурсів майже у всіх випадках використовується та сама універсальна функція:

```
BOOL CloseHandle(HANDLE hObject)
```

У випадку успішного виконання функції повертається значення TRUE, ін-

акше – FALSE.

Закриття дескриптора супроводжується зменшенням на одиницю лічильника посилань на об'єкт, що уможливлює видалення таких об'єктів, котрі не зберігаються постійно (nonpersistent), наприклад, тимчасових файлів або подій.

Спроба закриття недійсних дескрипторів або повторного закриття того самого дескриптора викликає генерацію винятку.

Треба мати на увазі, що *не слід закривати дескриптори стандартних пристроїв*.

ReadFile

Прототип цієї функції описується як:

```
BOOL ReadFile (HANDLE hFile, LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped)
```

Як видно із прототипу, функція повертає значення TRUE у випадку успішного виконання (успішною вважається навіть спроба, коли через читання з виходом за межі файла не зчитується жоден байт, при цьому кількість зчитаних байтів *lpNumberOfBytesRead установлюється такою, що дорівнює 0). У противному випадку, якщо значення дескриптора файла або інших параметрів, використовуваних при виклику функції, виявилися недійсними, виникає помилка, і вертається значення FALSE.

Якщо при відкритті файла в параметрі dwAttrsAndFlags не встановлений прапор перекриття вводу/виводу – FILE_FLAG_OVERLAPPED читання починається з поточної позиції покажчика файла, і покажчик файла переміщується на число зчитаних байтів.

Зміст параметрів, переданих до функції, є досить очевидним з їхньої назви й типів. Тому нижче наведені тільки їхні короткі описи.

Дескриптор файла, котрий зчитується – hFile повинен бути створений тільки з параметром, що обов'язково включає значення GENERIC_READ. Па-

раметр lpBuffer є покажчиком на буфер у пам'яті, куди поміщаються дані, котрі зчитуються. Відповідно, nNumberOfBytesToRead – кількість байтів, які повинні бути зчитані з файла. Як вже відзначалося, lpNumberOfBytesRead – покажчик на змінну, у якій зберігається фактично зчитане число байт. Параметр lpOverlapped – покажчик на структуру OVERLAPPED, за допомогою якої задається зсув покажчика усередині файла. Якщо інформація з файла зчитується послідовно, то цьому параметру привласнюється NULL.

WriteFile

Функція читання із прототипом

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD
  nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped)
```

Як і завжди, функція повертає значення TRUE у випадку успішного виконання, а інакше – FALSE.

Значення параметрів функції особливого пояснення не вимагають, оскільки вони здебільш знайомі з попередніх описів.

Якщо при відкритті файла на запис, не був використаний прапор FILE_FLAG_WRITE_THROUGH, то успішне виконання запису не обов'язково означає, що дані дійсно записані на диск. Якщо при виклику функції покажчик файла був встановлений на його кінець, Windows збільшує довжину існуючого файла.

Нижче наведений приклад коду програми, що робить копіювання одного файла до іншого. Імена файлів передаються програмі як параметри командного рядка.

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 16384
int main (int argc, LPTSTR argv [])
```

```
{
  HANDLE hIn, hOut;
  DWORD nIn, nOut;
  CHAR buffer[BUF_SIZE];
  if (argc != 3) {
    fprintf (stderr, "Usage: %s file1 file2\n", argv [0]);
    return 1; }
  hIn = CreateFile (argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if (hIn == INVALID_HANDLE_VALUE) {
    fprintf (stderr, "Cannot open input file. Error: %x\n",
              GetLastError());
    return 2; }
  hOut = CreateFile (argv[2], GENERIC_WRITE, 0, NULL,
          CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
  if (hOut == INVALID_HANDLE_VALUE) {
    fprintf (stderr, "Cannot open output file. Error: %x\n",
             GetLastError());
    CloseHandle(hIn);
    return 3; }
  while (ReadFile (hIn, buffer, BUF_SIZE, &nIn, NULL)
         && nIn > 0) {
    WriteFile (hOut, buffer, nIn, &nOut, NULL);
    if (nIn != nOut) {
      fprintf (stderr, "Fatal write error: %x\n",
               GetLastError());
      CloseHandle(hIn);
      CloseHandle(hOut);
      return 4; } }
  CloseHandle (hIn);
  CloseHandle (hOut);
  return 0;
}
```

Як видно з наведеного коду, у випадку збою при виконанні файлових операцій, необхідно за допомогою функції GetLastError визнавати характер виниклої помилки.

Продуктивність програм роботи з файлами підвищується при використанні буфера більшого розміру, крім того рекомендується при виклику функції CreateFile встановлювати прапор FILE_FLAG_SEQUENTIAL_SCAN.

2.1.2.2 Стандартні пристрої й консольний ввод/вивід

В Windows, як і в Linux, передбачені три стандартних пристрої – для вводу даних (input), виводу даних (output) і виводу повідомлень про помилки (error). Але, на відміну від Linux, в Windows за ними не закріплені які-небудь певні дескриптори, а доступ до пристроїв здійснюється за допомогою дескрипторів типу HANDLE, одержати які можна за допомогою спеціальної функції.

GetStdHandle

Функція із прототипом

```
HANDLE GetStdHandle(DWORD nStdHandle)
```

використовується для одержання дескриптора будь-якого стандартного пристрою. При невдачі вертається значення INVALID_HANDLE_VALUE.

Значення констант для параметра nStdHandle наведені в табл. 2.10.

Таблиця 2.10 – Значення констант параметра nStdHandle

Ім'я пристрою	Призначення пристрою
STD_INPUT_HANDLE	Стандартний пристрій вводу
STD_OUTPUT_HANDLE	Стандартний пристрій виводу
STD_ERROR_HANDLE	Стандартний пристрій повідомлень про помилки

Послідовні виклики функції GetStdHandle з тим самим аргументом не приводять до створення нових або дублюванню існуючих дескрипторів, а повертають те саме значення. *Закриття дескриптора стандартного пристрою робить цей пристрій недоступним*. Тому рекомендують повторно відкривати дескриптор без закриття пристрою.

SetStdHandle

Ця функція оголошується в такий спосіб:

```
BOOL SetStdHandle(DWORD nStdHandle, HANDLE hHandle)
```

і зв'язує дескриптор, отриманий для стандартного пристрою nStdHandle, з раніше відкритим файлом, що має дескриптор hHandle, тим самим цей файл призначається як стандартний пристрій.

ReadConsole, *WriteConsole*

Функції ReadConsole і WriteConsole аналогічні функціям ReadFile і WriteFile.

```
BOOL ReadConsole(HANDLE hConsoleInput,  LPVOID lpBuffer,
  DWORD cchToRead, LPDWORD lpcchRead, LPVOID lpReserved)

BOOL WriteConsole(HANDLE hConsoleOuput, LPVOID lpBuffer,
  DWORD cchToWrite, LPDWORD lpcchWriten, LPVOID lpReserved)
```

Значення обох параметрів, пов'язаних з кількістю символів, що підлягають зчитуванню, (cchToRead) і фактично зчитаних (lpcchRead) символів, виражаються в термінах узагальнених символів, а не байтів, а значення параметра lpReserved повинне дорівнювати NULL.

Для консольного вводу/виводу можуть використовуватися й функції ReadFile і WriteFile, але переважніше використовувати спеціальні консольні функції, насамперед тому, що вони маніпулюють не байтами, а узагальненими символами (TCHAR), і обробляють символи відповідно до поточних режимів консолі, які встановлюються функцією SetConsoleMode.

SetConsoleMode

Функція призначена для завдання режиму консолі й має такий прототип:

```
BOOL SetConsoleMode(HANDLE hConsoleHandle, DWORD fdevMode)
```

Дескриптор буфера вводу консолі або буферу дисплея hConsoleHandle, повинен створюватися із правами доступу GENERIC_WRITE, навіть якщо пристрій призначений тільки для вводу інформації.

Параметр fdevMode задає спосіб обробки символів. У табл. 2.11 наведені п'ять найчастіше використовуваних прапорів цього параметра, причому всі вони встановлюються за замовчуванням.

Таблиця 2.11 – Найчастіше використовувані прапори параметра fdevMode

Мнемонічне ім'я прапорця	Призначення прапорця
ENABLE_LINE_INPUT	Повернення з функції читання (ReadConsole) відбувається тільки після зчитування символу повернення каретки
ENABLE_ECHO_INPUT	Луна-відображення на екрані символів, що вводяться
ENABLE_PROCESSED_INPUT	Встановлення прапора приводить до обробки системою керуючих символів повернення на одну позицію, повернення каретки й переходу на новий рядок
ENABLE_PROCESSED_OUTPUT	Встановлення цього прапора призводить до обробки системою керуючих символів повернення на одну позицію, табуляції, подачі звукового сигналу, повернення каретки й переходу на новий рядок
ENABLE_WRAP_AT_EOL_OUTPUT	Перехід на наступний рядок екрана, як при звичайному виводі символів, так і при їх луна-відображенні в процесі вводу

2.1.2.3 Керування файлами й каталогами

Як і інш сучасні ОС, Windows надає ряд функцій, призначених для керування файлами й каталогами.

Робота з функціями для керування файлами звичайно не становить складності. З їхньою допомогою можна видаляти, копіювати й перейменовувати файли. Існує також функція, призначена для створення імен тимчасових файлів.

DeleteFile

При видаленні файла, як видно із прототипу, досить указати його ім'я.

```
BOOL DeleteFile(LPCTSTR lpFileName)
```

При зазначенні імені бажано використовувати повні імена файлів, що починаються з букви диска або з ім'я сервера (UNC). Відкритий файл видалити неможливо. Спроба виконання подібної операції закінчиться невдачею.

CopyFile

Щоб скопіювати файл можна, але зовсім не обов'язково створювати цілу програму, досить використати одну функцію:

```
BOOL CopyFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName,
  BOOL fFailIfExists)
```

Функція копіює існуючий файл із заданим ім'ям і привласнює копії зазначене нове ім'я. У випадку існування файла з таким же ім'ям він буде замінений новим файлом тільки в тому випадку, якщо значенням параметра fFailIfExists є FALSE.

CreateHardLink

Починаючи з версії файлової системи NTFS5, з'явилася можливість створювати тверде посилання на файл, аналогічне твердим посиланням в Linux, використовуючи для цього функцію CreateHardLink:

```
BOOL CreateHardLink(LPCTSTR lpFileName,
  LPCTSTR lpExistingFileName, BOOL lpSecurityAttributes)
```

Два перших аргументи такі ж як і у функції CopyFile, але розташовані у зворотному порядку. Обидва імені файла, нове і існуюче, повинні належати до одного й того ж тому файлової системи, але можуть відповідати різним катало-

гам. Атрибути захисту файла lpSecurityAttributes, якщо такі є, застосовні й до нового ім'я файла.

Кількість твердих посилань на файл вказується в полі структури BY_HANDLE_FILE_INFO. Це поле перевіряється у функції DeleteFile для визначення того, може або не може даний файл бути вилучений. Як і в Linux, поки кількість посилань на файл не дорівнює 0, віддаляються тільки записи про файл з каталогу.

Ярлики файлів в Windows надають засоби, подібні до гнучких посилань в Linux, але скористатися ними можуть тільки користувачі оболонки, оскільки, на відміну від Linux, ярлики є властивістю тільки оболонки, а не ядра ОС.

MoveFile і *MoveFileEx*

Доступнимі є дві функції, що дозволяють перейменовувати, або переміщати, файли. Ці ж функції застосовуються й до каталогів.

```
BOOL MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName)

BOOL MoveFileEx(LPCTSTR lpExistingFileName,
  LPCTSTR lpNewFileName, DWORD dwFlags)
```

Якщо новий файл вже існує, функція MoveFile завершується з помилкою; у цьому випадку слід використовувати функцію MoveFileEx.

Призначення параметрів lpExistingFileName і lpNewFileName, а так само їхній тип цілком очевидні з опису.

У випадку функції MoveFile, до її виклику, нового файла або каталогу з ім'ям lpNewFileName не повинне існувати. Новий файл може належати іншій файловій системі або перебувати на іншому диску, але нові каталоги обов'язково повинні перебувати на тому самому томі. Якщо значення цього параметра покласти рівним NULL, то існуючий файл буде видалений. Значення опцій параметра dwFlags наведені в табл. 2.12.

Таблиця 2.12 – Опції параметра dwFlags

Мнемонічне ім'я прапорця	Призначення прапорця
MOVEFILE_REPLACE_EXISTING	Дозволяє заміну існуючого файла
MOVEFILE_WRITETHROUGH	Використовується, якщо необхідно, щоб функція виконала повернення лише після того, як файл буде фактично переміщений на диску
MOVEFILE_COPY_ALLOWED	Якщо новий файл знаходиться на іншому томі, переміщення здійснюється шляхом послідовного виконання функцій CopyFile і DeleteFile
MOVEFILE_DELAY_UNTIL_REBOOT	Цей прапор встановлюється тільки адміністратором системи, і не може застосовуватися разом із прапором MOVEFILE_COPY_ALLOWED, при цьому фактичне переміщення файла здійснюється тільки після перезавантаження системи

CreateDirectory і *RemoveDirectory*

Створення й видалення каталогів здійснюється за допомогою двох простих функцій.

```
BOOL CreateDirectory(LPCTSTR lpPathName,
   LPSECURITY_ATTRIBUTES lpSecurityAttributes)

BOOL RemoveDirectory(LPCTSTR lpPathName)
```

В цьому разі, як і завжди, параметр lpPathName – покажчик на рядок, що містить шлях до каталогу, котрий створюється або видаляється. Якщо не передбачається захист каталогу, другий параметр у виклику функції CreateDirectory можна покласти рівним NULL.

SetCurrentDirectory і *GetCurrentDirectory*

Як і в Linux, у кожного процесу є поточний робочий каталог, крім того на кожному диску є свій робочий каталог. У системі є можливість встановлення й одержання інформації про поточний каталог.

Для установки каталогів призначена функція:

```
BOOL SetCurrentDirectory(LPCTSTR lpPathName)
```

Параметр lpPathName визначає шлях до нового поточного каталогу, можна використовувати як відносний шлях, так і повний шлях із вказівкою букви диска й двокрапки, або ім'я UNC. Якщо зазначено тільки ім'я диска, то робочим каталогом програми стає робочий каталог даного диска

Функція:

```
DWORD GetCurrentDirectory(DWORD cchCurDir, LPTSTR lpCurDir)
```

через покажчик lpCurDir повертає повний шлях до поточного каталогу. Буфер для рядка встановлює програміст, а в першому параметрі – cchCurDir програміст указує розмір буфера для ім'я каталогу. Цей розмір визначається в символах з урахуванням завершального нульового символу рядка, а не в байтах.

Функція повертає дійсну довжину рядка для поточного каталогу, або необхідний розмір буфера, якщо він малий і, нарешті, 0 у випадку помилки. Тому при тестуванні успішності виконання функції слід обов'язково перевіряти дві умови: чи дорівнює значення, що повертається, нулю й чи не перевищує воно значення, що задане аргументом cchCurDir.

Нижче наведений код, що пояснює застосування цих функцій.

```
#include <windows.h>
#include <stdio.h>
#define DIRNAME_LEN (MAX_PATH + 2)
int main (int argc, LPTSTR argv []) {
  TCHAR pwdBuffer [DIRNAME_LEN];
  DWORD lenCurDir;
  lenCurDir = GetCurrentDirectory(DIRNAME_LEN, pwdBuffer);
  if (lenCurDir == 0)
    fprintf(stderr, "\nНе вдається одержати шлях\n");
  if (lenCurDir > DIRNAME_LEN)
    fprintf(stderr, "\nзанадто довгий шлях\n");
    fprintf(stdout, "\n%s\n\n", pwdBuffer);
  system("pause");
  return 0; }
```

2.1.2.4 Робота з покажчиками файлів

SetFilePointer

В Windows аналогічно Linux для кожного дескриптора файла підтримується ***покажчик файла*** (file pointer), що вказує поточну позицію у файлі. При відкритті файла шляхом виклику функції CreateFile покажчик файла приймає нульове значення, котре збільшується при читанні або записі. Для забезпечення прямого доступу до даних у файлі, використовується функція SetFilePointer:

```
DWORD SetFilePointer(HANDLE, LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod)
```

Зміст параметра hFile, що передається до функції є цілком очевидним – це дескриптор файла відкритого для запису, читання або запису/читання, у якому необхідно перемістити покажчик.

Покажчик файла (зсув у файлі) задається цілим числом розміром у вісім байтів. Такий покажчик, що має тип LONGLONG, передається у функцію SetFilePointer за допомогою двох параметрів, перший з яких lDistanceToMove містить молодшу частину покажчика (подвійне слово зі знаком), а старша частина передається не безпосередньо, а за допомогою покажчика на подвійне слово – lpDistanceToMoveHigh. Якщо робота ведеться з файлами, розмір яких не більше за 2^{32}-2 байтів, то значення цього параметра встановлюється в NULL. У підручниках рекомендують застосовувати цю функцію для невеликих файлів.

Значення функції, що повертається, у випадку невдалого її завершення дорівнює 0xffffffff.

Якщо ж операція завершилася без помилок, то функція повертає новий покажчик файла. Причому молодша частина вертається самою функцією (подвійне слово без знаку), а старша частина значення цього покажчика вертається через уміст покажчика lpDistanceToMoveHigh (якщо при виклику він був відмінний від NULL).

Останній параметр – dwMoveMethod задає один із трьох можливих режимів переміщення покажчика файла. Мнемонічні константи для цього пара-

метра наведені в табл. 2.13.

Таблиця 2.13 – Значення опцій параметра dwMoveMethod

Ім'я константи	Призначення параметра
FILE_BEGIN	Покажчик файла позиціонується відносно до початку файла, причому параметр DistanceToMove інтерпретується як беззнакове число
FILE_CURRENT	Покажчик файла переміщається убік більших або менших значень щодо поточної позиції, причому параметр DistanceToMove інтерпретується як число зі знаком. Позитивним значенням відповідає переміщення покажчика файла убік більших значень
FILE_END	Покажчик файла переміщається убік більших або менших значень відносно до позиції кінця файла

Крім цього безпосереднього способу завдання покажчикика у файлі, його позицію можна встановити за допомогою структури OVERLAPPED

Як уже вказувалося раніше, останнім параметром у функціях ReadFile і WriteFile є адреса структури перекриття OVERLAPPED. Ця структура, крім інших, має два поля Offset і OffsetHigh. Задаючи значення полів структури OVERLAPPED, можна операції вводу/виводу починати із зазначеної позиції. Ще одним полем цієї структури є дескриптор hEvent, значення якого повинне встановлюватися рівним NULL.

2.1.2.5 Визначення розміру файла

GetFileSize

З метою одержання даних про розмір файла простіше за все скористатися функцією GetFileSize:

```
DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh)
```

Із прототипу видно, що для повернення розміру файла використовується, по суті, той же спосіб, що й для повернення фактичного покажчика файла функ-

цією SetFilePointer, тобто молодше подвійне слово розміру файла вертається самою функцією, а старше через розіменування покажчика lpFileSizeHigh.

Значення, що повертається, в разі коли воно дорівнює 0xffffffff, указує на можливу помилку. Для перевірки наявності помилок *обов'язково* слід використовувати функцію GetLastError.

2.1.2.6 Атрибути файлів і керування каталогами

API Windows забезпечує можливість перегляду каталогів з метою пошуку файлів і інших каталогів. Одночасно з пошуком можна одержати атрибути файлів. Пошук здійснюється за допомогою спеціального ***дескриптора пошуку***, що повертається функцією FindFirstFile. Безпосередньо для знаходження файлів використовується функція FindNextFile. Пошук припиняється функцією FindClose.

FindFirstFile

Прототип функції має вигляд:

```
HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpffd)
```

У функції два параметри. Перший з них – lpFileName є покажчиком на рядок, що містить ім'я каталогу або повне ім'я файла. При зазначенні імен можна використовувати метасимволи ? та *.

Параметр lpffd – покажчик на структуру WIN32_FIND_DATA, у яку поміщається інформація про перший знайдений файл або каталог, якщо такий був знайдений.

Визначення структури WIN32_FIND_DATA наведено нижче.

```
typedef struct_WIN32_FIND_DATA {
DWORD dwFileAttributes;
FILETIME ftCreationTime;
FILETIME ftLastAccessTime;
```

```
FILETIME ftLastWriteTime;
DWORD nFileSizeHigh;
DWORD nFileSizeLow;
DWORD dwReserved0;
DWORD dwReserved1;
TCHAR cFileName[MAX_PATH];
TCHAR cAlternateFileName[14];
} WIN32_FIND_DATA;
```

Поля структури містять повну інформацію про файл: атрибути файла, установлені функцією CreateFile і ряд інших, час створення, час останнього звертання й час останньої зміни файла, розмір файла, ім'я файла, котре не утримує шлях доступу, ім'я файла у форматі DOS 8.3 (застаріла інформація).

При вдалому завершенні функція повертає дескриптор пошуку, інакше значення INVALID_HANDLE_VALUE.

FindNextFile

Подальший пошук файлів здійснюється функцією із прототипом

```
BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpffd)
```

Параметри функції очевидні з її оголошення. Функція повертає значення FALSE, якщо аргументи недійсні або не вдається знайти файл. В останньому випадку, функція перевірки помилки GetLastError, повертає значення ERROR_NO_MORE_FILES.

FindClose

Після завершення пошуку, дескриптор пошуку повинен бути закритий функцією:

```
BOOL FindClose(HANDLE hFindFile)
```

Застосування звичайної функції CloseHandle у цьому випадку неможли-

во, таке закриття дескриптора пошуку приводить до генерації винятку.

2.1.2.7 Приклад програми, що виводить список атрибутів файла

Програма, що наведена нижче, переглядає каталог для пошуку файлів, котрі відповідають шаблону пошуку. Для кожного знайденого файла програма відображає ім'я файла й, якщо був заданий параметр -l, то і його атрибути. Дана програма ілюструє функції Windows, призначені для роботи з каталогами.

Програма використовує прив'язку до поточного каталогу, тому необхідно задавати відносні повні імена файлів.

```
/* Командний рядок Dir [параметри] [файли] */
#include "EveryThing.h"
static BOOL TraverseDirectory(LPCTSTR, DWORD, LPBOOL);
static DWORD FileType(LPWIN32_FIND_DATA);
static BOOL ProcessItem(LPWIN32_FIND_DATA, DWORD, LPBOOL);

int _tmain(int argc, LPTSTR argv[]) {
  BOOL Flags [MAX_OPTIONS], ok = TRUE;
  TCHAR PathName [MAX_PATH +1], CurrPath [MAX_PATH + 1];
  LPTSTR pSlash, pFileName;
  int i, FileIndex;
/* "Розібрати" шаблон пошуку на "батьківську частину" і ім'я
файла. */
  FileIndex = Options(argc, argv, _T("R1"), &Flags[0],
                      &Flags[1], NULL);
// Зберегти поточний шлях доступу.
  GetCurrentDirectory(MAX_PATH, CurrPath);
/* Шлях доступу не зазначений. Використовувати поточний
каталог. */
  if (argc < FileIndex +1)
    ok = TraverseDirectory(_T("*"), MAX_OPTIONS, Flags);
  else for (i = FileIndex; i < argc; i++) {
// Обробити всі шляхи, зазначені в командному рядку.
    ok = TraverseDirectory(pFileName, MAX_OPTIONS, Flags) && ok;
// Відновити вихідний каталог.
  SetCurrentDirectory(CurrPath); }
  return ok ? 0 : 1; }

static BOOL TraverseDirectory(LPCTSTR PathName, DWORD NumFlags,
                              LPBOOL Flags)
```

```
/* Обхід дерева каталогів; виконати функцію ProcessItem для
кожного випадку збігу. PathName: відносне або абсолютне ім'я
каталогу, що переглядається.*/
{
  HANDLE SearchHandle;
  WIN32_FIND_DATA FindData;
  BOOL Recursive = Flags[0];
  DWORD FType, iPass;
  TCHAR CurrPath[MAX_PATH + 1];
  GetCurrentDirectory(MAX_PATH, CurrPath);
  for (iPass = 1; iPass <= 2; iPass++) {
// Прохід 1: вивід списку файлів.
// Прохід 2: обхід дерева каталогів (якщо задано опцію -R).
    SearchHandle = FindFirstFile(PathName, &FindData);
    do {
// Файл або каталог?
      FType = FileType(&FindData);
// Вивести ім'я й атрибути файла.
      if (iPass == 1)
        ProcessItem(&FindData, MAX_OPTIONS, Flags);
// Обробити підкаталог.
      if (FType == TYPE_DIR && iPass == 2 && Recursive) {
        _tprintf(_T ("\n%s\\%s:"), CurrPath, FindData.cFileName);
// Підготовка до обходу каталогу.
      SetCurrentDirectory(FindData.cFileName);
      TraverseDirectory(_T("*"), NumFlags, Flags);
// Рекурсивний виклик.
      SetCurrentDirectory(_T("..")); } }
    while (FindNextFile(SearchHandle, &FindData));
    FindClose (SearchHandle); }
  return TRUE; }
static BOOL ProcessItem(LPWIN32_FIND_DATA pFileData, DWORD
                        NumFlags, LPBOOL Flags)
// Виводить список атрибутів файла або каталогу.
{
  const TCHAR FileTypeChar[] = {' ', 'd'};
  DWORD FType = FileType(pFileData);
  BOOL Long = Flags[1];
  SYSTEMTIME LastWrite;
  if (FType != TYPE_FILE && FType != TYPE_DIR)
    return FALSE;
  _tprintf(_T ("\n"));
// Чи був параметр "-l" зазначений у командному рядку?
  if (Long) {
    _tprintf(_T("%c"), FileTypeChar[FType - 1]);
```

```
    _tprintf(_T("%10d"), pFileData->nFileSizeLow);
    FileTimeToSystemTime(&(pFileData->ftLastWriteTime),
                         &LastWrite);
    _tprintf(_T(" %02d/%02d/%04d %02d:%02d:%02d"),
      LastWrite.wMonth, LastWrite.wDay, LastWrite.wYear,
      LastWrite.wHour, LastWrite.wMinute,
      LastWrite.wSecond); }
  _tprintf(_T(" %s"), pFileData->cFileName);
  return TRUE;
}

static DWORD FileType(LPWIN32_FIND_DATA pFileData)
/* Підтримувані типи файлів - TYPE_FILE: файл; TYPE_DIR: каталог;
  TYPE_DOT: каталоги . або .. */
{
  BOOL IsDir;
  DWORD FType;
  FType = TYPE_FILE;
  IsDir = (pFileData->dwFileAttributes &
           FILE_ATTRIBUTE_DIRECTORY) != 0;
  if (IsDir)
    if (lstrcmp(pFileData->cFileName, _T(".")) == 0 ||
        lstrcmp(pFileData->cFileName, _T("..")) == 0)
       FType = TYPE_DOT;
    else FType = TYPE_DIR;
  return FType;
}
```

2.2 Завдання на самостійну роботу

1) По основній і додатковій літературі ознайомитися з функціями роботи з файлами й з файловими системами в ОС Linux і Windows.
2) Знати принципи роботи із програмами XShell і mc (див. метод. вказ. до лабор. раб. №1).
3) Ознайомитися з компілятором gcc, а, також, із програмою make (лабор. раб. №1).
4) Вивчити структуру й команди файла makefile для компіляції C/C++ проектів (лабор. раб. №1).
5) Розглянути й проаналізувати makefile, автоматично створений у середовищі компіляції Dev-CPP.

6) Підготувати й помістити до протоколу makefile файл для компіляції програми до лабораторної роботи.

2.3 Варіанти завдань до лабораторної роботи

У всіх варіантах завдання необхідно розробити програму, у якій, використовуючи системні виклики ОС Linux, виконати наступні дії:

- у каталозі «/home/<ЛОГІН_КОРИСТУВАЧА>/» створити дерево каталогів відповідно до номера варіанта;
- с консолі ввести й записати у файл ReadMe.txt, розташований у каталозі FILES/, номер лабораторної роботи, тему роботи, групу, прізвище, ім'я та по батькові автора програми;
- в одному з каталогів за завданням, у файл int.bin записати 1000 випадкових цілих чисел, визначити і надрукувати на консоль розмір файла;
- аналогічно, у файл double.bin записати 1000 випадкових чисел; типу double, визначити і надрукувати на консоль розмір файла;
- у файл text.txt програмно скопіювати довільний текст, наприклад власну С програму;
- у файл symbols.chr занести 1000 випадкових байтів у діапазоні від 0x20 до 0xfe;
- зробити перевизначення виводу зі стандартного пристрою виводу у файл на диску зробити програмний обхід дерева каталогів.

Повторити всі ті ж дії в програмі для ОС Windows, використовуючи системні виклики API цієї системи Дерево каталогів, з файлами, що втримувалися в ньому, створювати на загальнодоступному диску N: або на власному флеш носії.

Варіант 1

```
FILES/
|    ReadMe.txt
|---Overheads/
|        Ses00
|        int.bin
|---Demo/
     |    Extra.bin
     |    text.txt
     |---PTR01/
     |        double.bin
     |        symbols.chr
     |---CHAR/
              file.c
              long.asc
```

Варіант 2

```
FILES/
|    ReadMe.txt
|---facts/
|    |    long.asc
|    |---Unicode/
|         |    double.bin
|         |---Icu/
|              text.txt
|              icud.dat
|---Mappings/
|        int.bin
|---Adobe/
         HKSCS.txt
         symbols.chr
```

Варіант 3

```
FILES/
|    ReadMe.txt
|---TechInfo/
|    |    double.bin
|    |---Lang/
|              af.txt
|              text.txt
|---ABBYY/
     |---Reader/
          |    int.bin
          |    Zlib.dll
          |---Support/
                  long.asc
                  symbols.chr
```

Варіант 4

```
FILES/
|    ReadMe.txt
|---System/
|        int.bin
|---Mappings/
|        long.asc
|---Adobe/
     |    HKSCS.txt
     |    text.txt
     |---Unicode/
          |    double.bin
          |---Icu/
               symbols.chr
               icud.dat
```

Варіант 5

```
FILES/
|    ReadMe.txt
|---PlugIns/
|        File.diz
|---BY/
     |---Fine/
     |        long.asc
     |        Ablib.dll
     |---Lingvo/
     |        double.bin
     |---Reg/
     |        text.txt
     |        int.bin
     |        symbols.chr
     |---Src/
              Src.rar
```

Варіант 6

```
FILES/
|    ReadMe.txt
|    double.bin
|---Panel/
|    |    long.asc
|    |---Lingvo/
|         |    int.bin
|         |    Abbrev.lsd
|         |---Dic/
|         |    |    text.txt
|         |    |---Index/
|         |---Support
|---Adobe/
         symbols.chr
         Index.dat
```

Варіант 7

```
FILES/
    ReadMe.txt
    Acronis/
        Sandal/
            symbols.chr
            text.txt
            Common/
                int.bin
                link.dll
        TrueHome/
            license.txt
            long.asc
            Common/
                double.bin
    Image/
```

Варіант 8

```
FILES/
    ReadMe.txt
    BVRDE/
        symbols.chr
        Lex/
            int.bin
        Solutions/
        Sounds/
            long.asc
        Templates/
            text.txt
            Files/
            double.bin
            Wizard/
                makefile
    Sym/
```

Варіант 9

```
FILES/
    ReadMe.txt
    Comp/
        text.txt
        manifest
        Comm/
            Code/
                double.bin
            Docu/
                int.bin
                V24.odc
                ru/
                    long.asc
                    ver.odc
            Sym/
                symbols.chr
    Index/
```

Варіант 10

```
FILES/
    ReadMe.txt
    CaliPo/
        libre.exe
        Calibre/
            recomp.exe
            long.asc
            plugins/
                imagefor/
                    text.txt
            resour/
                symbols.chr
                content/
                    int.bin
                    box.png
                    read/
    fonts/
        double.bin
```

Варіант 11

```
FILES/
    ReadMe.txt
    Ascii/
        long.asc
        Form/
            Code/
                double.bin
                symbols.chr
            Docu/
                text.txt
            Sym/
                int.bin
    Index/
    Ole/
        Integers.bin
```

Варіант 12

```
FILES/
    ReadMe.txt
    Eclipse/
        symbols.chr
        facts.xml
        PlugIns/
            double.bin
            AltHistory/
                Alt.dll
                long.asc
            BackGroundCopy/
                File_ID.diz
                Reg/
                    text.txt
                Src/
                    int.bin
            System/
                Dest.on
    Temp/
```

2.4 Контрольні питання

1) Як розшифровується абревіатура gcc?
2) Який компілятор використовується в інтегрованому середовищі програмування DEV-Cpp?
3) Чим відрізняються опції командного рядка від інших параметрів команди?
4) Які існують категорії ключів (опцій) командного рядка компілятора gcc?
5) Назвіть найважливіші, на ваш погляд, частини gcc, котрі звичайно встановлюються в ОС.
6) Назвіть приклади додаткового програмного забезпечення тісно пов'язаного з компілятором, але такого, що не входить в його склад.
7) Назвіть кілька суфіксів файлів, які зв'язуються з компілятором gcc у режимі компіляції програм мовою C.
8) Назвіть кілька суфіксів файлів, які зв'язуються з компілятором gcc у режимі компіляції програм мовою C++.
9) Приведіть найпростіший командний рядок для компіляції програми мовою C.
10) Приведіть найпростіший командний рядок для компіляції програми мовою C++.
11) З яким ім'ям створюється виконуваний файл при використанні найпростішого командного рядка компіляції?
12) Як указати компіляторові ім'я результуючого виконуваного файла?
13) Які дії виконує компілятор по команді **`gcc -c anyfile.c`**?
14) Чим дії команди **`g++ prog.cpp`** відрізняються від дій команди **`gcc prog.c`**?
15) Назвіть найбільше часто використовувані ключі gcc?
16) Чому не рекомендується безпосереднє використання команди gcc у

випадку складних програмних проектів?

17) Для чого служить програма make?
18) Що необхідно зробити перед використанням програми make?
19) Якою командою ОС виконується компіляція за допомогою програми make?
20) Навіщо потрібний і що із себе становить makefile?
21) Із правил, якого виду складається найпростіший makefile?
22) Що таке пререквізит?
23) Що таке команда в makefile?
24) Назвіть основне призначення команд в makefile?
25) Як змусити програму make виконати правило makefile, метою якого не є відновлення змінених файлів?
26) Які цілі називаються абстрактними?
27) Що таке головна мета в makefile?
28) У якій частині makefile розташовується головна мета?
29) З якою метою в makefile використовуються змінні?
30) Назвіть деякі найбільш уживані імена змінних?
31) Укажіть синтаксис використання змінних у командах.
32) Чи можна, і якщо так, то чому, явно не вказувати команди компіляції в правилах, які описують створення об'єктних модулів?
33) Чи можуть записи в makefile групуватися не по пререквізитах, а по цілям?
34) Яка обов'язкова вимога накладається на синтаксис команд в makefile?
35) З яких п'яти видів конструкцій складається makefile?
36) Що таке явне правило?
37) Для чого служить неявне правило?
38) Як використовувати не стандартне ім'я для makefile?
39) Назвіть п'ять базових функцій при роботі з файлами.
40) Чи є каталоги файлами чи ні?

41) Що собою становить й для чого служить файловий індекс (inode)?
42) Що собою становить елемент каталогу?
43) Що таке системні виклики й для чого вони слугують?
44) Чи існує в ОС Linux можливість звертання до пристроїв як до файлів?
45) Чим бібліотечні функції відрізняються від системних викликів?
46) Що таке дескриптор файла?
47) Які стандартні дескриптори автоматично відкриваються в кожній програмі?
48) Які значення вертаються системними викликами open, read, write і close?
49) Де зберігається значення помилки, що виникла в результаті виконання системних викликів Linux?
50) Для якої мети служить параметр void *buf у системних викликах read і write?
51) Чи є можливість одночасно відкрити й створити файл?
52) Поясніть призначення системного виклику lseek.
53) Для яких цілей слугують системні виклики fstat, stat і lstat?
54) Чи можна за допомогою системного виклику creat створити каталог?
55) Як змінити права доступу до каталогу і його власника?
56) Які системні виклики підключають файл до каталогу й видаляють його?
57) Як діє системний виклик unlink?
58) Чи може виклик rmdir видалити не порожній каталог?
59) Які функції найбільше часто використовуються при роботі з каталогами в ОС Linux?
60) Назвіть функції, котрі полегшують роботу зі змінною errno?
61) Яка функція API Windows використовується для створення нових і відкриття існуючих файлів?

62) Порівняйте властивості дескрипторів файла у двох розглянутих ОС.
63) Чи можна параметру lpSecurityAttributes привласнити значення NULL при виклику функції CreateFile?
64) З якою метою у функції CreateFile використовується параметр hTemplateFile?
65) Яка універсальна функція використовується для звільнення системних ресурсів і закриття об'єктів будь-якого типу?
66) Значення якого типу повертають функції ReadFile і WriteFile?
67) Які дескриптори файлів закріплені за стандартними пристроями вводу/виводу в Windows?
68) Як в ОС Windows одержати дескриптор стандартного пристрою вводу/виводу?
69) Як зробити читання/запис з/на консоль в ОС Windows?
70) Для чого призначена функція SetConsoleMode?
71) Назвіть функції для керування файлами.
72) Чи можна в ОС Windows створювати тверді посилання на файли як в UNIX-подібних системах?
73) Чи можна за допомогою функції MoveFile перейменовувати або переміщати каталоги?
74) Як створюються й видаляються каталоги в ОС Windows?
75) Яка функція в ОС Windows відповідає системному виклику lseek в UNIX?
76) Укажіть призначення й значення, що повертається функцією FindFirstFile.
77) Яка функція безпосередньо використовується для знаходження файлів у каталозі Windows?

ЛАБОРАТОРНА РОБОТА № 3
ПРОЦЕСИ В LINUX І WINDOWS

Література до роботи: [1] стор. 61 – 69, 76 – 81, 110 – 148
[3] стор. 97 – 105, 126 – 156, [4] стор. 68 – 174

МЕТА РОБОТИ

Розглянути процеси у ОС Linux і Windows, за допомогою системних викликів Linux і функцій Win32 API навчитися створювати і керувати процесами в цих системах (без урахування потоків і нитей).

3.1 Огляд теоретичної частини

3.1.1 Основні відомості про процеси Linux

Процеси й ***сигнали*** формують головну частину операційного середовища Linux. Вони управляють майже всіма видами діяльності ОС Linux і UNIX-подібних комп'ютерних систем.

Стандарти UNIX, а саме IEEE Std 1003.1, 2004 Edition, визначають процес як «Адресний простір з одним або декількома потоками, що виконуються в ньому, і системні ресурси, необхідні цим потокам». У першому наближенні під процесом можна розуміти просто будь-яку програму, що виконується в системі.

Linux, будучи багатозадачною багатокористувальницькою системою, дозволяє багатьом користувачам одночасно звертатися до системи. Кожний користувач у той саме час може запускати багато програм або навіть кілька екземплярів однієї й тієї ж програми. Сама система виконує в цей час інші програми, що управляють системними ресурсами й контролюють доступ користувачів.

Програма, що виконується, або процес складається із програмного коду, даних, змінних (котрі займають системну пам'ять), відкритих файлів (файлових дескрипторів) і оточення. В системі Linux процеси працюють так, що вони спі-

льно використовують код і системні бібліотеки, таким чином в будь-який момент часу в пам'яті перебуває тільки одна копія програмного коду.

На рис. 3.1 схематично показана структура двох процесів, котрі запущені різними користувачами: Користувач_А и Користувач_Б. Обидва користувача одночасно запустили програму grep для пошуку різних рядків у різних файлах.

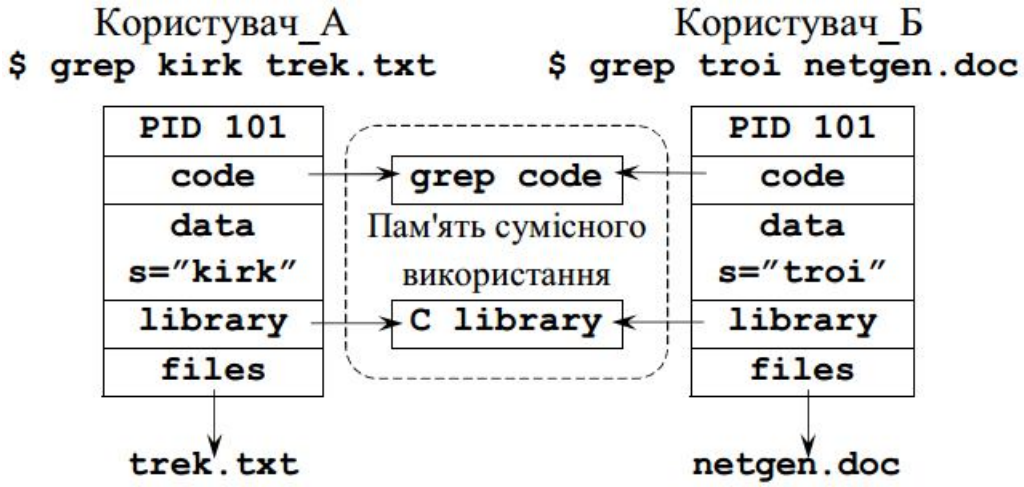

Рисунок 3.1 – Співіснування двох процесів в операційній системі

На рисунку видно, що кожному процесу виділений унікальний номер PID – ***ідентифікатор процесу***. Звичайно це позитивне ціле в діапазоні від 2 до 32768. При старті процесу вибирається найближче невикористане число. Ідентифікатор процесу рівний 1 у системі зарезервований для спеціального процесу init, що управляє іншими процесами. На рисунку двом процесам, котрі запущені користувачами, виділені ідентифікатори 101 і 102.

Код програми grep, що зберігається у файлі на диску, завантажується в область пам'яті, котра позначається як доступна тільки для читання й спільно використовується обома процесами. Системні бібліотеки також розміщуються в цій пам'яті і використовуються сумісно. Тому в пам'яті потрібна тільки одна копія будь-якої функції, наприклад, printf, навіть якщо вона викликається багатьма програмами.

З наведеної схеми видно, що дисковий файл з програмою grep, може не містити в собі код спільно використовуваної бібліотеки. Такий підхід – одер-

жання програмами часто використовуваних підпрограм зі стандартних бібліотек, наприклад, мовою C заощаджує значний обсяг ресурсів операційної системи.

Є цілком очевидним, що не все, що потребується програмою може бути спільно використано. Наприклад, змінні окремо використовуються кожним процесом. У прикладі рядок для пошуку, який є переданим команді grep, – це змінна s, що належить до простору даних кожного процесу. Ці простори розділені й, як правило, не можуть читатися іншим процесом. Файли, які застосовуються у двох командах grep, теж різні; у кожного процесу є свій набір файлових дескрипторів, котрі використовуються для доступу до файлів.

Крім того, у кожного процесу є власний стек, що застосовується для локальних змінних у функціях і для керування викликами функцій і поверненням з них. У кожного процесу є також власне оточення, що містить ***змінні оточення***, які можуть задаватися тільки для застосування в даному процесі. Процес завжди повинен підтримувати власний лічильник програми, запис того місця, до якого він добрався за час виконання, або потік виконання. У загальному випадку, процеси можуть мати кілька потоків виконання.

Процеси в Linux організовані в спеціальний набір «файлів» у каталозі /proc. Це спеціальні файли, які дозволяють «заглянути усередину» процесів під час їхнього виконання. Крім того, із процесами зв'язана ***таблиця процесів***. Вона являє собою набір структур даних – ***керуючих блоків процесів***, що описують процеси, котрі завантажені у теперішній момент. У керуючих блоках зберігаються ***ідентифікатор процесу*** – PID, стан процесу, рядок системної команди й інша інформація. ОС керує процесами за допомогою їх PID, які застосовуються як покажчики в таблиці процесів. Число процесів, що підтримуються системою, обмежується тільки обсягом пам'яті, котра є доступною для формування елементів таблиці процесів. В останніх версіях ядра системи, таблиці керуючих блоків замінені двома окремими структурами:

- ***хеш-таблицею*** для швидкого пошуку процесу по його PID;
- ***кільцевим двохзв'язним списком*** для забезпечення циклічної роботи

всіх процесів.

Здатність запускати нові процеси й чекати на них закінчення – одна з основних характеристик системи. Здебільшого кожний процес запускається іншим процесом, котрій зветься ***батьківським***, ***процесом предком*** або ***процесом, що породжує***. Подібним чином запущений процес називають ***дочірнім***, ***нащадком*** або ***породженим***.

Коли ОС Linux стартує, то перш за все завантажується і запускається на виконання системна програма, що одержує ідентифікатор процесу номер 1 – процес init. Практично це диспетчер процесів операційної системи й перший предок – прабатько всіх інших процесів. Інші системні процеси запускаються процесом init або іншим процесом, котрий попередньо був запущений процесом init.

Щоб вирішити, який процес дістане наступний квант часу, ядро Linux застосовує для цього спеціальний планувальник процесів. Рішення планувальником приймається виходячи із пріоритету процесу. Процеси з високим пріоритетом виконуються частіше, а інші, наприклад, такі як низькопріоритетні фонові завдання, – рідше. В ОС Linux процеси не можуть перевищити виділений їм квант часу. Вони переважно належать до різних завдань, тому припиняються й відновляються без взаємодії один з одним.

У багатозадачних системах кілька програм можуть претендувати на той самий ресурс, тому програми з короткими робочими циклами, що перериваються для вводу, вважаються такими, що поводять себе краще, чим програми, що використовують процесор для тривалого обчислення якого-небудь значення або безперервних запитів до системи, котрі стосуються її готовності до вводу/виводу даних. Програми, що поводять себе добре, називають nice-програмами. Цю «привабливість» можна виміряти. Операційна система визначає пріоритет процесу на основі значення nice (за замовчуванням такого, що дорівнює 0) і поведінки програми. Програми, що виконуються без пауз протягом довгих періодів, як правило, одержують більш низькі пріоритети. Програми, що роблять паузи час від часу, наприклад чекаючи вводу, одержують нагороду. Це до-

помагає зберегти чуйність програми, котра взаємодіє з користувачем; поки вона чекає якого-небудь вводу від користувача, система збільшує її пріоритет, щоб, коли програма буде готова відновити виконання, у неї був високий пріоритет. Задати значення nice для процесу можна за допомогою команди nice, а змінити його – за допомогою команди renice. Команда nice збільшує на 10 значення nice процесу, привласнюючи йому більш низький пріоритет.

3.1.1.1 Запуск нових процесів

У системі Linux, як і в інших UNIX-сумісних системах існує кілька різних можливостей запуску нових процесів з інших програм.

system

Найбільш простим способом запуску нового процесу є використання в програмі бібліотечної функції system. Функція оголошена як:

```
#include <stdlib.h>
int system(const char *string);
```

Функція system виконує команду, котра передається їй як рядок символів, і чекає її завершення. При звертанні до функції, значіння, що вона повертає, може мати три різні значення. Коли код котрий повертається, дорівнює 127, це означає, що командна оболонка не може бути запущена для виконання команди. Код рівний -1 сигналізує про наявність якоїсь іншої помилки і, як це прийнято у системі, тип помилки поміщається в змінну errno. У випадку вдалого виконання system повертає код завершення програми, котра виконувалась. Нижче наведений приклад використання функції system.

```
#include <stdlib.h>
#include <stdio.h>
int main()
  {
    printf("Запуск ps функцією system\n");
```

```
    system("ps x");
    printf("Завершено\n");
    exit(0);
}
```

Запуск відкомпільованого приклада програми призведе до виводу на екран термінала відомостей про процеси, котрі були запущені у системі. Можливий результат роботи програми наведений нижче.

```
wasq@linserv:~/Labs$ ./sys
Запуск ps функцією system
  PID TTY      STAT   TIME COMMAND
17922 ?        S      0:00 sshd: wasq@pts/0
17923 pts/0    Ss     0:00 -bash
18021 pts/0    S+     0:01 mc
18023 pts/1    Ss     0:00 bash -rcfile .bashrc
18213 pts/1    S+     0:00 ./sys
18214 pts/1    S+     0:00 sh -c ps x
18215 pts/1    R+     0:00 ps x
Завершено
```

Необхідно відзначити, що застосування функції system, є не самим гарним способом створення процесів тому, що програма, котра запускається використовує командну оболонку системи і, в зв'язку з чим, цей спосіб сильно залежить від варіанта установки командної оболонки й застосовуваного оточення. І останнє, програма, котра робить виклик, змушена чекати поки не завершиться процес, початий викликом system, і не може продовжити виконання інших завдань.

Замість виклику функції system, більш бажаною є заміна образа процесу

exec

В ОС Linux існує ціле сімейство споріднених системних викликів, котрі згруповані під заголовком exec. Вони відрізняються способом запуску процесів і передачею до них аргументів програми. Функції заміщають поточний процес новим, заданим в аргументі path або file. Функції exec можна застосовувати для передачі виконання від однієї програми до іншої програми. Наприклад, пе-

ред запуском якого-небудь додатка з політикою обмеженого застосування можна перевірити ім'я користувача й пароль. Функції exec більш ефективні в порівнянні з system, тому що вихідна програма вже більше не буде виконуватися після запуску нової програми, оскільки нова програма повністю витісняє й замінює ту програму, що викликала функцію exec. Прототипи всього сімейства наведені нижче:

```
#include <unistd.h>
char **environ;
int execl(const char *path, const char *arg0, ...,
                (char *)0);
int execlp(const char *file, const char *arg0, ...,
                 (char *)0);
int execle(const char *path, const char *arg0, ...,
                 (char *)0, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char
                *const envp[]);
```

Як видно з наведених прототипів, ці функції поділяються на дві групи – таку, що утримує суфікс l і функції із суфіксом v. Функції із суфіксом l – execl, execlp і execle приймають список аргументів, кількість яких може змінюватись при потребі, список закінчується покажчиком null ((char *)0). Другий набір функцій, із суфіксом v – execv, execvp і execve відрізняється тим, що у якості другого аргументу їм передається масив рядків. При цьому нова програма стартує з аргументами, котрі містяться в масиві argv, який, в свою чергу, передається у функцію main програми, що буде виконуватись.

Деякі з функцій містять ще й додатковий суфікс p або e. Суфікс p, слугує для визначення шляху до файла нової програми, котра буде виконуватися і передбачує проведення пошуку змінної оточення PATH. Якщо ця змінна не дозволяє знайти потрібний файл, то, у якості параметру, до функції необхідно передавати повне ім'я файла, включаючи й каталоги починаючи з кореневого. Якщо виникає в цьому потреба, функціям, за допомогою глобальної змінної-

покажчика char** environ, можна передати значення оточення програми.

Функції ж із другим додатковим суфіксом e (execle і execve) рядки, котрі будуть використані як оточення нової програми, дістають через додатковий аргумент – покажчик на масив рядків char *const envp[].

Для запуску програми за допомогою функцій exec можна вибирати й застосовувати будь-яку функцію із цього сімейства. Так нижче представлені варіанти використання всіх функцій сімейства для запуску тієї ж самої програми ps з її параметром x, що і в попередньому прикладі з функцією system.

```
#include <unistd.h>
// Приклад списку аргументів
// В argv[0] необхідно поміщати ім'я програми
char *const ps_argv[] = {"ps", "x", 0};
// Можливий приклад оточення
char *const ps_envp[] = {"PATH=/bin:/usr/bin",
                         "TERM=console", 0};
// Можливі виклики функцій exec
execl("/bin/ps", "ps", "x", 0);
// Передбачається, що файл ps у каталозі /bin
execlp("ps", "ps", "x", 0);
// Передбачається, що каталог /bin занесений до PATH
execle("/bin/ps", "ps", "x", 0, ps_envp);
// Передається своє власне оточення
execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Ну, а безпосередньо попередній приклад із запуском процесу ps з іншої програми за допомогою виклику exec буде виглядати в такий спосіб:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
  {
    printf("Запуск ps функцією execlp\n");
    execlp("ps", "ps", "x", 0);
    printf("Завершено\n");
    exit(0);
  }
```

Якщо порівняти вивід попередньої й цієї програми, то можна помітити, що, у випадку нової програми, серед запущених процесів відсутніми є копія процесу командної оболонки й процес самої програми, котра зробила виклик, а так само й повідомлення про її завершення.

```
wasq@linserv:~/Labs$ ./exe
Запуск ps функцією execlp
  PID TTY      STAT   TIME COMMAND
17922 ?        S      0:00 sshd: wasq@pts/0
17923 pts/0    Ss     0:00 -bash
18021 pts/0    S+     0:01 mc
18023 pts/1    Ss     0:00 bash -rcfile .bashrc
18218 pts/1    R+     0:00 ps x
```

Така поведінка другої програми пов'язана з тим, що функції сімейства exec не завантажують копію командної оболонки, натомість повністю заміщають процес, що зробив виклик exec і, як правило, не повертають керування до нього. Але у випадку виникнення помилки все ж відбувається повернення в вихідну програму з кодом повернення з функції рівним -1 і встановленням відповідного значення змінної errno.

Нові процеси, котрі запущені exec, успадковують багато властивостей вихідного процесу. Зокрема, відкриті файлові дескриптори залишаються відкритими в новому процесі, поки не встановлений їхній прапор FD_CLOEXEC, але, на відміну від файлів, будь-які відкриті у вихідному процесі потоки каталогів закриваються.

3.1.1.2 Дублювання образа процесу

fork

Обидва з описаних вище способів запуску нового процесу мають один загальний недолік – вони *не дають можливості паралельного виконання* вихідного й запущеного процесів. Для здійснення паралельної роботи процесів служить системний виклик fork.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Він дублює поточний процес, створюючи новий елемент у таблиці процесів з безліччю атрибутів таких самих, як і в поточному процесі. Новий процес майже повністю є ідентичним до вихідного, виконує той же програмний код, але робить це у своєму просторі даних, своєму оточенні й зі своїми файловими дескрипторами. Використання виклику fork разом з функціями exec – це найбільш часто застосовуваний при програмуванні у UNIX-подібних системах спосіб створення нових процесів.

Схематично дія виклику fork показана на рис. 3.2.

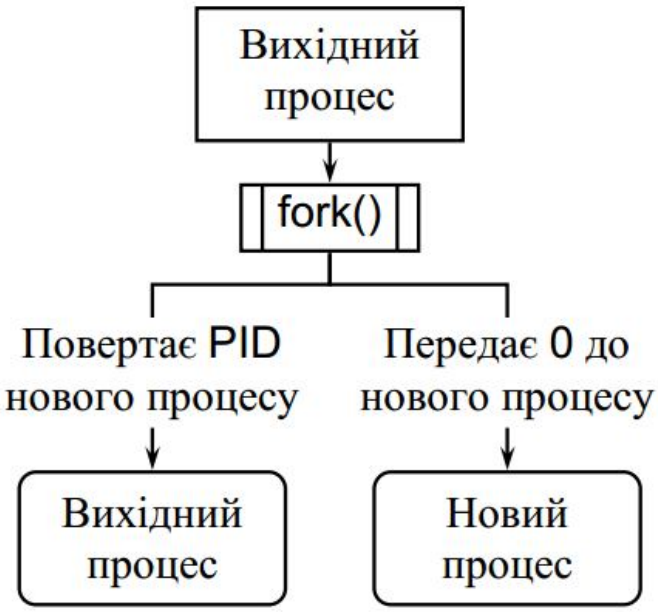

Рисунок 3.2 – Дія системного виклику fork

Як видно із цього рисунка, виклик повертає в батьківський процес PID нового дочірнього процесу. Новий процес продовжує виконання так само, як і вихідний, за винятком того, що в дочірній процес виклик fork передає 0. Це дозволяє визначити який із процесів є батьківським, а який – дочірнім.

Якщо виклик fork завершується аварійно, він повертає -1. Частіше за все це відбувається через велику кількість дочірніх процесів, яку може мати бать-

ківський процес. У цьому випадку змінної errno привласнюється значення EAGAIN. Якщо для елемента таблиці процесів недостатньо місця або не вистачає віртуальної пам'яті, змінна errno дістане значення ENOMEM.

Схема типового використання системного виклику fork може бути проілюстрована наступним кодом:

```
pid_t new_pid;
new_pid = fork();
switch(new_pid)
  {
    case -1:
      // Тут обробляється помилка
      break;
    case 0:
      // Тут міститься код дочірнього процесу
      break;
    default:
      // Тут розташований код батьківського процесу
      break;
  }
```

Нижче наведений найпростіший приклад спільної роботи батьківського й дочірнього процесів.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
  pid_t pid;
  char* message;
  int n;
  printf("fork program starting\n");
  pid = fork();
  switch(pid) {
  case -1:
    perror("fork failed");
    exit(1);
  case 0:
    message = "This is the child";
```

```
    n = 5;
    break;
  default:
    message = "This is the parent";
    n = 3;
    break; }           // switch
  for (; n > 0; n--) {
    puts(message);
    sleep(1); }        // for
  exit(0); }           // main
```

У цій програмі, до того як буде виконане звертання до системного виклику fork, існує й працює тільки один процес. Після виклику fork починають виконуватися два незалежних процеси, кожний з яких має свій власний простір імен. Це видно з того факту, що кожний із процесів привласнює своє власне значення змінним n і message. Цикл for так само виконується одночасно обома процесами в окремих адресних просторах, хоча зовні це виглядає як один оператор програми.

3.1.1.3 Очікування процесу

Коли дочірній процес запускається за допомогою виклику fork, він починає виконуватися незалежно. Іноді виникає необхідність знати, коли закінчився дочірній процес.

wait

За допомогою системного виклику wait можна змусити батьківський процес дочекатися завершення дочірнього процесу перед своїм продовженням.

Системний виклик має такий прототип:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

Системний виклик wait є найпростішим засобом синхронізації процесів і

змушує батьківський процес зробити паузу доти, поки один з його дочірніх процесів не зупиниться. Виклик повертає PID дочірнього процесу, що завершився. Не нульовий покажчик stat_loc указує на місце, куди записується інформація про стан дочірнього процесу. Ці відомості дозволяють батьківському процесу визначити статус завершення дочірнього процесу, тобто значення, що повертається з функції main дочірнього процесу або передається до функції exit в ньому.

У заголовному файлі sys/wait.h оголошено кілька макросів, що дозволяють інтерпретувати інформацію про стан процесу. Ці макроси описані в наведеній нижче табл. 3.1.

Таблиця 3.1 – Макроси для інтерпретації стану дочірнього процесу

Макрос	Опис
WIFEXITED(stat_val)	Ненульовий, якщо дочірній процес завершений нормально
WEXITSTATUS(stat_val)	Якщо WIFEXITED ненульовий, повертає код завершення дочірнього процесу
WIFSIGNALED(stat_val)	Ненульовий, якщо дочірній процес завершується сигналом, котрий не перехоплюється
WTERMSIG(stat_val)	Якщо WIFSIGNALED ненульовий, повертає номер сигналу
WIFSTOPPED(stat_val)	Ненульовий, якщо дочірній процес зупинився
WSTOPSIG(stat_val)	Якщо WIFSTOPPED ненульовий, повертає номер сигналу

Нижче наведена найпростіша програма спільної роботи двох процесів, але в порівнянні з попереднім прикладом, у цій програмі батьківський процес припиняє свою роботу й очікує завершення дочірнього, після чого продовжує свою роботу.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
```

```
  pid_t pid;
  char* message;
  int n;
  int exit_code;
  printf("fork program starting\n");
  pid = fork();
  switch(pid)
    {
      case -1:
        perror("fork failed");
        exit(1);
      case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
      default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }
  for (; n > 0; n--)
    {
      puts(message);
      sleep(1);
    }
// Далі очікування закінчення дочірнього процесу:
  if (pid != 0) {
    int stat_val;
    pid_t child_pid;
    child_pid = wait(&stat_val);
    printf("Child has finished: PID = %d\n",
           child_pid);
    if (WIFEXITED(stat_val))
      printf("Child exited with code %d\n",
             WEXITSTATUS(stat_val));
    else printf("Child terminated abnormally\n");
  }
  exit(exit_code);
}
```

waitpid

Для очікування закінчення дочірнього процесу, можна використовувати ще один системний виклик. Він називається waitpid і застосовується для очікування завершення не будь-якого дочірнього процесу, а певного, що визначається у параметрах виклику.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

PID конкретного дочірнього процесу, закінчення якого потрібно чекати, передається у виклик, як перший параметр (аргумент pid). Якщо в якості його передати -1,то waitpid поверне інформацію про будь-який дочірній процес. Як і виклик wait, цей виклик записує інформацію про стан процесу в місце, зазначене аргументом stat_loc, якщо він не NULL. Аргумент options дозволяє змінити поведінку waitpid. Найбільш корисна опція WNOHANG, вона заважає виклику waitpid припиняти виконання процесу, що його викликав. Виклик можна застосовувати для з'ясування, чи завершився який-небудь із дочірніх процесів, і якщо ні, то продовжувати виконання. Таким чином, якщо необхідно, щоб батьківський процес періодично перевіряв завершення конкретного дочірнього процесу, у коді батьківського процесу можна використовувати виклик waitpid, наприклад, так:

```
ch_pid = waitpid(child_pid, (int *)0, WNOHANG);
```

Такий виклик поверне нуль, якщо дочірній процес не завершився й не зупинений, або child_pid, якщо це відбулося. У випадку помилки waitpid повертає -1 і встановлює змінну errno. Є три можливих значення цієї змінної:

- ECHILD – немає дочірніх процесів;
- EINTR – виклик перерваний сигналом;
- EINVAL – невірний аргумент options.

Застосування виклику fork для створення процесів може виявитися дуже

корисним, але *в обов'язковому порядку* необхідно відслідковувати виконання дочірніх процесів. Коли дочірній процес завершується, зв'язок його з батьком зберігається доти, поки батьківський процес у свою чергу не завершиться нормально, або не викличе wait. Отже, запис про дочірній процес не зникає у таблиці процесів негайно. Стаючи неактивним, дочірній процес усе ще залишається в системі, оскільки його код завершення повинен бути збережений, на випадок якщо батьківський процес надалі викличе wait. Такий процес стає, так званим, ***мертвим процесом*** або ***процесом-зомбі***.

Якщо після переходу дочірнього процесу в стан зомбі, батьківський процес завершиться незвичайно, то дочірній процес автоматично повинен одержати нового батька. Таким батьком стає процес із PID, що дорівнює 1 – процес init. Але дочірній процес-зомбі залишається в таблиці процесів, поки він не буде перехоплений процесом init. Чим більше таблиця процесів, тим довше не відбудеться захоплення. Слід уникати процесів-зомбі, оскільки вони споживають ресурси доти, поки процес init не вичистить їх.

3.1.1.4 Сигнали і їхня обробка

Сигнал – подія, що генерується системами UNIX і Linux у відповідь на деяку ситуацію. Процес, що одержав сигнал, може певним чином зреагувати на його одержання.

Сигнали збуджуються деякими помилковими ситуаціями, наприклад, порушеннями сегментації пам'яті, помилками процесора при виконанні операцій із плаваючою крапкою або некоректними командами. Крім того вони генеруються командною оболонкою й оброблювачами терміналів для виклику переривань і можуть явно пересилатися від одного процесу до іншого як спосіб передачі інформації або корекції поведінки. Сигнали можуть збуджуватися, уловлюватися й відповідно оброблятися або ігноруватися, принаймні, деякі.

Імена сигналів задаються за допомогою підключення до програми заголовного файла signal.h. Вони починаються із префікса SIG і наведені у табл. 3.2

Якщо процес одержує один з основних сигналів без попередньої підготовки до його перехоплення, він негайно завершується.

Таблиця 3.2 – Імена сигналів, що задані у файлі signal.h

Ім'я сигналу	Опис
	Основні
SIGABORT	Процес аварійно завершується*
SIGALRM	Сигнал тривоги
SIGFPE	Виняток операції із плаваючою крапкою*
SIGHUP	Несподіване зупинення або роз'єднання
SIGILL	Некоректна команда*
SIGINT	Переривання термінала
SIGKILL	Знищення (не може бути перехоплений або ігнорований)
SIGPIPE	Запис у канал без зчитування
SIGQUIT	Завершення роботи термінала
SIGSEGV	Некоректний доступ до сегмента пам'яті*
SIGTERM	Завершення, вихід
SIGUSR1	Сигнал 1, визначений користувачем
SIGUSR2	Сигнал 2, визначений користувачем
	Додаткові
SIGCHLD	Дочірній процес зупинений або завершився
SIGCONT	Продовжити виконання, якщо процес був припинений
SIGSTOP	Зупинити виконання (не може захоплюватися або ігноруватися)
SIGTSTP	Сигнал зупинення, що посилається з термінала
SIGTTIN	Фоновий процес намагається читати
SIGTTOU	Фоновий процес намагається писати

З додаткових сигналів, особливий інтерес представляє сигнал SIGCHLD. Цей сигнал корисний для керування дочірніми процесами. За замовчуванням він ігнорується. Інші сигнали змушують процеси, що їх одержали, зупинитися, за винятком сигналу SIGCONT, що викликає поновлення процесу. Крім названих, додаткові сигнали рідко використовуються в користувальницьких програмах.

signal

Програми обробляють сигнали за допомогою бібліотечної функції signal.

* Можуть бути початі дії, що залежать від конкретної реалізації.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Це досить складне оголошення говорить про те, що signal – це функція, що приймає два параметри: sig – сигнал, який потрібно перехопити або ігнорувати й func – функція, яку слід викликати при одержанні даного сигналу. Остання повинна приймати єдиний аргумент типу int (прийнятий сигнал) і мати тип void. Функція сигналу повертає функцію того ж типу, котра є попереднім значенням функції, що вказана для обробки сигналу, або одне з двох спеціальних значень:

- SIG_IGN – ігнорувати сигнал;
- SIG_DFL – відновити поведінку за замовчуванням.

Наведений нижче приклад пояснює використання функції signal. Програма, що виконує код цього приклада, замість звичайного завершення по натисканню комбінації <Ctrl – C> (відправлення процесу, що виконується, сигналу SIGINT з клавіатури) реагує виводом відповідного повідомлення. Повторне натискання <Ctrl – C> завершує програму.

У цьому прикладі функція ouch реагує на сигнал, що передається у параметрі sig. Ця функція викликається, коли виникне сигнал. Вона виводить повідомлення й потім у наступному рядку – (void)signal(SIGINT, SIG_DFL) відновлює обробку сигналу за замовчуванням для сигналу SIGINT. Функція main взаємодіє із сигналом SIGINT, котрий генерується при натисканні комбінації клавіш <Ctrl – C>. В інший час вона перебуває в нескінченному циклі, виводячи один раз у секунду повідомлення «Hello World!».

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig) {
  printf("OUCH! - Отриманий сигнал %d\n", sig);
  (void)signal(SIGINT, SIG_DFL);
}
```

```
int main() {
  (void)signal(SIGINT, ouch);
  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Як видно з даного приклада, функція обробки сигналу ouch приймає один цілочисельний параметр – номер сигналу, що призводить до виклику функції. Це зручно, якщо та сама функція у загальному випадку може застосовуєтьс я для обробки декількох сигналів. У програмах *завжди необхідно користуватися іменами сигналів*, а не їхніми числовими значеннями, які можуть різнитися в різних операційних системах і, навіть, у різних версіях однієї системи.

sigaction

Тут слід обов'язково сказати, що хоча перехоплення сигналів за допомогою функції signal досить часто застосовувалося в старих UNIX-програмах, але сучасні стандарти X/Open і специфікації UNIX рекомендують застосування більш надійного програмного інтерфейсу для сигналів – sigaction.

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

Структура sigaction застосовується для визначення дій, що вживаються при одержанні сигналу, переданого в аргументі sig. Структура оголошена у файлі signal.h і, як мінімум, включає наступні елементи:

```
void (*)(int)sa_handler // функція, SIG_DFL або SIG_IGN
sigset_t sa_mask        /* сигнали, що заблоковані для
                           sa_handler */
int sa_flags            // модифікатори дій сигналу
```

У структурі sigaction поле sa_handler являє собою покажчик на функ-

цію, котра повинна бути викликана при одержанні сигналу. Вона дуже схожа на функцію func, що передавалася до функції signal. Цьому полю можна також привласнювати спеціальні значення SIG_IGN і SIG_DFL для позначення того, що сигнал повинен ігноруватися або повинна бути відновлена дія за замовчуванням.

Функція sigaction зв'язує сигнал sig з дією, котра задається у структурі sigaction. Цей зв'язок виражається в наступному:

- якщо значення другого – і третього параметрів (act і oact) не дорівнюють null, то для сигналу задається дія, котра є заданою в першому параметрі, а в адресі, на яку вказує oact, запам'ятовується дія, що виконувалася раніше;
- якщо act дорівнює null, а oact не дорівнює null, то здійснюється тільки запам'ятовування старої дії пов'язаної із сигналом sig і більше ніяких дій не виконується;
- якщо ж другий параметр є покажчиком на дійсну структуру sigaction, а параметр oact дорівнює null, то буде тільки виконана нова дія, пов'язана з сигналом, запам'ятовування старої дії провадитися не буде.

Як і функція signal, функція sigaction повертає 0 у випадку успішного виконання й -1, якщо заданий сигнал некоректний або була зроблена спроба захопити, або проігнорувати сигнал, для якого цього робити не можна. Одночасно змінна errno одержує значення EINVAL.

Поле sa_mask описує безліч сигналів, які додаються в маску сигналів процесу перед викликом функції sa_handler. Ця безліч сигналів блокується й не може бути переданою процесу. Застосування поля sa_mask усуває стан гонок, це явище може виникнути при застосуванні функції signal, у разі коли сигнал може бути повторно отриманий до того моменту, як його оброблювач завершиться.

Сигнали, котрі захоплюються оброблювачами, заданими в структурі sigaction, не відновлюються за замовчуванням, як це має місце при застосуванні функції signal. Для їхнього відновлення в полі sa_flags необхідно встановити значення SA_RESETHAND.

З урахуванням сказаного, попередній приклад можна переписати з використанням не функції signal, а функції sigaction спільно зі структурою.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig) {
  printf("OUCH! - Отриманий сигнал %d\n", sig);
}

int main() {
  struct sigaction act;
  act.sa_handler = ouch;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);
  while (1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Як видно з вихідного коду, для завдання функції ouch, як оброблювача сигналу SIGINT (натискання <Ctrl – C>), використовується функція sigaction замість signal. При цьому визначається структура sigaction. У прикладі не використовуються ніякі прапори, а за допомогою функції sigemptyset створюється порожня маска сигналів.

3.1.1.5 Відправлення сигналів

kill

Сигнал від одного процесу до іншого відправляється за допомогою виклику функції kill. Причому одержувачем сигналу може виступати й сам процес відправник. Виклик завершається аварійно, якщо у процесу відправника немає повноважень на відправлення сигналу. Найчастіше це буває у випадку, коли процес одержувач належить іншому користувачеві, тобто в обох процесів по-

винен бути той самий ID – ***ідентифікатор користувача***. Ця функція еквівалентна команді оболонки з тим же ім'ям.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Функція kill посилає заданий сигнал sig процесу з ідентифікатором, що вказується в аргументі pid. У випадку успіху вона повертає 0.

Функція kill завершується аварійно, повертає -1 і встановлює змінну errno у наступних випадках:

- задано невірний сигнал (errno = EINVAL);
- процес немає відповідних повноважень (errno = EPERM);
- процес із вказаним pid не існує (errno = ESRCH).

alarm

Спеціальний сигнал зі значенням SIGALRM може бути збуджений не тільки передачею його у функцію kill, але і за допомогою спеціальної функції alarm, що називають будильником або сигналом тривоги. Функція збуджує сигнал SIGALRM у визначений час у майбутньому.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Єдиний параметр функції – seconds, задає час затримки посилки сигналу SIGALRM у секундах щодо моменту виклику функції. Якщо значення параметра дорівнює 0, то відбувається скасування будь-якого невиконаного запиту на сигнал будильника. У кожного процесу може бути тільки один невиконаний сигнал будильника. Функція alarm повертає кількість секунд, що залишилися до відправлення будь-якого невиконаного виклику alarm, або -1 у випадку аварійного завершення.

Наведена нижче програма демонструє використання функцій alarm і kill.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
static int alarm_fired = 0;
void ding(int sig) {
  alarm_fired = 1;
  }
int main() {
  pid_t pid;
  printf("alarm application starting\n");
  pid = fork();
  switch(pid) {
    case -1:          // Аварійне завершення
      perror("Процес не створений");
      exit(1);
    case 0:           // Дочірній процес
      alarm(10);
  }
// Це код батьківського процесу
  (void)signal(SIGALRM, ding);
  pause();            // Очікування сигналу
  if (alarm_fired) printf("Дзинь-ь-ь-ь\n");
  kill(pid, SIGINT);
  printf("От і все\n");
  exit(0);
}
```

У програмі створюється два процеси. Дочірній процес викликає функцію alarm, що із затримкою в 10 секунд збуджує сигнал SIGALRM. Батьківський же процес викликом функції signal встановлює оброблювач сигналів на функцію ding і переходить у режим очікування сигналу (функція pause). З приходом сигналу виконується перевірка одержання саме сигналу будильника (установка прапора alarm_fired), а також завершується дія функції pause, потім перевіряється наявність сигналу і виводиться повідомлення. Вивід повідомлення за допомогою printf усередині самого оброблювача не є бажаним, оскільки в оброблювачі можна застосовувати не будь-які функції мови С, а тільки, так звані, реентерабельні (printf до них не належить). Далі батьківський процес відправляє сигнал припинення своєму дочірньому процесу (виклик kill) і завершує роботу

сам. Виконання цієї операції є обов'язковим, інакше дочірній процес перейде в стан «зомбі» і буде займати ресурси системи, не виконуючи ніякої роботи.

Застосування сигналів і припинення виконання програми – важливі складові програмування в ОС Linux. Це означає, що програма необов'язково повинна виконуватися увесь час. Замість того щоб довго працювати в циклі, перевіряючи, чи не відбулася та або інша подія, програма може просто чекати на її настання. Це особливо важливо в багатокористувальницькому середовищі, де процеси спільно використовують один процесор, і такий вид діяльного очікування дуже впливає на продуктивність системи.

Однак слід відзначити, що до використання сигналів у програмі слід підходити досить ретельно. По-перше, потрібно враховувати, що деякі системні виклики можуть закінчитися аварійно, якщо сигнал створить помилкову ситуацію, що не була врахована ще до того, як була додана обробка сигналів. По-друге, можлива поява стану гонок, один випадок виникнення такої ситуації був розглянутий у лекціях. Тому завжди необхідно дуже уважно перевіряти програмний код, що використовує сигнали.

3.1.2 Процеси Windows

Як і UNIX, операційна система Windows для паралельного виконання завдань надає в розпорядження програміста процеси. Процес в Windows являє собою об'єкт, що володіє власним незалежним віртуальним адресним простором, у якому можуть розміщатися код і дані, захищені від інших процесів. Але, на відміну від UNIX, процес тут відразу створюється з одним або декількома потоками усередині, які можуть виконуватися незалежно друг від друга.

Можна сказати, що процеси UNIX є подібними до процесів Windows, що мають єдиний потік. Але при всій своїй порівнянності, моделі процесу в UNIX і Windows значно відрізняються друг від друга. Насамперед, в Windows відсутній еквівалент UNIX-функції fork, що створює копію батьківського процесу, включаючи його простір даних, купу й стек. В Windows важко домогтися точної емуляції fork. У той же час, створення процесу Windows є доволі аналогічним із

звичайнім для UNIX ланцюжком послідовних викликів функцій fork і execl (або будь-якої іншої із сімейства exec). Але, на відміну від Windows, шляхи доступу в UNIX визначаються винятково змінною середовища PATH.

В UNIX-сумісних системах прийнято говорити про процеси-предки, або батьківські процеси, і процеси-нащадки, або дочірні процеси, однак між процесами Windows ці відносини фактично не підтримуються. Хоча термінологія батько/нащадок і використовується, але це є лише зручним способом відображення того факту, що один процес породжується іншим. Оскільки відносини предок/нащадок між процесами в Windows не підтримуються, то виконання дочірнього процесу буде тривати навіть після того, як завершиться батьківський процес. Крім того, в Windows відсутні групи процесів. Існує, однак, обмежена форма групи процесів, у якій всі процеси дістають керуючі події від консолі.

Процеси Windows ідентифікуються не тільки ідентифікаторами процесів, як в UNIX, але й ***дескрипторами***, як і будь-які інші об'єкти системи.

Створюючи процеси й управляючи ними, додатки можуть організовувати паралельне виконання декількох завдань, що забезпечують обробку файлів, проведення обчислень або зв'язок з іншими системами в мережі.

З погляду програміста, кожному процесу належать ресурси, представлені наступними компонентами:

- один або декілька потоків;
- віртуальний адресний простір, відмінний від адресних просторів інших процесів, якщо не вважати областей пам'яті, розподілених явно для спільного використання декількома процесами, причому поділювані відображені файли спільно використовують фізичну пам'ять тоді як процеси, що їх розділяють, використовують різні віртуальні адресні простори.
- один або декілька сегментів коду, включаючи код DLL;
- один або декілька сегментів даних, що містять глобальні змінні;
- рядки, що містять інформацію про оточення, наприклад, інформацію про поточний шлях доступу до файлів.
- купа процесу;

– різного роду ресурси, наприклад, дескриптори відкритих файлів і інші купи.

3.1.2.1 Створення процесу Windows

CreateProcess

Однієї з найважливіших функцій Windows, що забезпечують керування процесами, є функція CreateProcess, котра створює новий процес із єдиним потоком. При зверненні до цієї функції потрібно вказати ім'я файла програми, котра буде виконуватися.

Гнучкі й потужні можливості функції CreateProcess забезпечуються десятьма параметрами, що їй передаються. У практичному програмуванні для спрощення звичайно використовуються лише значення параметрів, що задаються за замовчуванням. Для більш легкого вивчення інших аналогічних функцій, має сенс докладно розглянути кожний з тих параметрів, що передаються у функцію CreateProcess.

Прототип функції має вигляд:

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine, LPSECURITY_ATTRIBUTES lpsaProcess,
  LPSECURITY_ATTRIBUTES lpsaThread, BOOL bInheritHandles,
  DWORD dwCreationFlags, LPVOID lpEnvironment,
  LPCTSTR lpCurDir, LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcInfo)
```

Значення, що повертається функцією у випадку успішного створення процесу та його потоку – TRUE, інакше – FALSE.

Останній параметр у списку являє собою покажчик на структуру, через яку функція повертає два окремих дескриптори типу HANDLE, по одному для процесу й потоку. Ці дескриптори належать до створюваного функцією CreateProcess нового процесу і його основного потоку. Оскільки вертається не один, а два дескриптори, то при створенні й знищенні процесів у програмі *необхідно ретельно стежити за своєчасним закриттям обох дескрипторів*,

коли вони більше не потрібні. Забудькуватість при закритті дескрипторів потоків є однією з найпоширеніших помилок і призводить до витоку ресурсів у процесі роботи. У свою чергу, закриття дескриптора потоку не призводить до припинення його виконання; функція CloseHandle лише видаляє посилання на потік усередині процесу, що викликав функцію CreateProcess.

Інші параметри в описі функції являють собою наступне:

– два покажчики – lpApplicationName і lpCommandLine типу LPCTSTR і LPTSTR, відповідно, використовуються разом для зазначення програми, що буде виконуватися, і аргументів її командного рядка;
– за допомогою двох наступних параметрів, обидва з яких мають тип LPSECURITY_ATTRIBUTES – lpsaProcess і lpsaThread у функцію передаються покажчики на структури атрибутів захисту процесу й потоку. Якщо розроблювача задовольняють атрибути захисту, що задані за замовчуванням, то як фактичний параметр передається значення NULL;
– логічний параметр – bInheritHandles показує, чи успадковує новий процес різні відкриті дескриптори (файлів, відображень файлів і т.і.) від вихідного процесу. Наслідувані дескриптори мають ті ж атрибути, що й вихідні;
– параметр типу DWORD – dwCreationFlags обумовлює параметри процесу й потоку, що створюються, і може поєднувати в собі декілька значень прапорців, включаючи наступні:
    а) CREATE_SUSPENDED – основний потік створюється в припиненому стані й у подальшому запускається вже викликом функції ResumeThread;
    б) DETACHED_PROCESS і CREATE_NEW_CONSOLE – перший прапор означає створення нового процесу, у якого консоль відсутня, а другий – процесу, у якого є власна консоль, *одночасне застосування обох прапорців є неприпустимим*;
    в) CREATE_NEW_PROCESS_GROUP – створюваний процес є кореневим для нової групи процесів, якщо процеси групи розді-

ляють загальну консоль, то вони всі одержують керуючі сигнали консолі (Ctrl – C або Ctrl – Break);

г) прапорці, що керують пріоритетами потоків нового процесу, найчастіше з багатьох різних прапорців використовується режим за замовчуванням – NORMAL_PRIORITY_CLASS.

– призначення покажчику типу LPVOID – lpEnvironment є передача блоку параметрів настроювання оточення нового процесу. Якщо задано значення NULL, то новий процес буде використовувати значення параметрів оточення батьківського процесу;
– наступний параметр-покажчик – lpCurDir типу LPCTSTR служить для посилання на рядок, що містить шлях до поточного каталогу нового процесу. Якщо задано значення NULL, то як поточний каталог буде використовуватися робочий каталог батьківського процесу;
– параметр lpStartupInfo типу LPSTARTUPINFO – покажчик на структуру, що описує зовнішній вигляд основного вікна й містить дескриптори стандартних пристроїв нового процесу. В разі необхідності, цю структуру можна одержати з батьківського процесу за допомогою функції GetStartupInfo. У структурі STARTUPINFO для вказівки стандартних пристроїв вводу, виводу й повідомлень про помилки необхідно визначити значення полів дескрипторів стандартних пристроїв (hStdInput, hStdOutput і hStdError), а так само поля dwFlags, привласнивши йому значення STARTF_USESTDHANDLES. Крім того необхідно визначити всі дескриптори, які будуть потрібні дочірньому процесу й переконатися в тім, що ці дескриптори є наслідуваними й при виклику функції CreateProcess значення параметра bInheritHandles встановлене рівним TRUE;
– останній параметр – lpProcInfo, як уже говорилося вище, повертає покажчик на структуру PROCESS_INFORMATION, у яку містяться значення ***дескрипторів*** і ***глобальних ідентифікаторів*** процесу й потоку. Оголошення структури має такий вигляд:

```
typedef struct _PROCESS_INFORMATION
  {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
  } PROCESS_INFORMATION;
```

У цю структуру поміщаються дескриптори й глобальні ідентифікатори процесу й потоку. Така надмірна інформаційна структура потрібна тому, що одні функції керування процесами при їхньому виклику вимагають завдання ідентифікаторів процесів, а інші – дескрипторів, наприклад, універсальні функції очікування, що забезпечують відстеження переходів об'єктів різного типу в певні стани.

3.1.2.2 Зазначення модуля, що виконується, і командного рядка

Для зазначення ім'я файла модуля, що виконується, можна використовувати як параметр lpApplicationName, так і параметр lpCommandLine. При цьому варто дотримуватися таких правил:

- покажчик lpApplicationName, якщо його значення не дорівнює NULL, вказує на рядок, що містить ім'я файла модуля для виконання, якщо ім'я модуля містить пробіли, його обов'язково треба брати у лапки;
- якщо ж значення покажчика lpApplicationName дорівнює NULL, то ім'я модуля визначається першої з лексем, переданих параметром lpCommandLine (найчастіше використовуване сполучення параметрів);
- параметр lpApplicationName, якщо він не дорівнює NULL, визначає модуль, що виконується, і, якщо вказується модуль з поточного диска й каталогу, то вказується тільки ім'я файла й розширення (наприклад, .EXE або .CMD чи .BAT) у противному випадку задається повний шлях доступу та ім'я файла;
- якщо значення параметра lpApplicationName дорівнює NULL і, якщо у

першій лексемі параметра lpCommandLine повний шлях доступу не зазначений, то здійснюється пошук файла в наступному порядку:

1) каталог модуля поточного процесу;
2) поточний каталог;
3) системний каталог Windows, інформацію про який можна одержати за допомогою функції GetSystemDirectory;
4) каталог Windows, що повертається функцією GetWindowsDirectory;
5) каталоги, перераховані в змінній оточення PATH.

Командний рядок у новий процес може бути переданий з головної програми через аргумент argv його функції main або бути отриманим ним шляхом виклику функції GetCommandLine у вигляді одиночного рядка символів.

3.1.2.3 Наслідувані дескриптори

Іноді дочірньому процесу буває потрібний доступ до об'єкта, дескриптор якого визначений у батьківському процесі. І, якщо цей дескриптор наслідуваний, то дочірній процес може одержати копію відкритого дескриптора батьківського процесу й використовувати його у своїх цілях. Найчастіше в дочірній процес так передаються дескриптори стандартного вводу й виводу.

Перетворення дескриптора в наслідуваний вимагає виконання декількох кроків.

Прапор bInheritHandles, переданий при виклику функції CreateProcess, обумовлює можливість одержання копії наслідуваних дескрипторів відкритих файлів, процесів та інше дочірнім процесом. Але крім передачі цього прапора, для спадкування будь-якого конкретного дескриптора, потрібно зробити додаткові дії тому, що самі по собі дескриптори не стають наслідуваними за замовчуванням. Наслідуваним дескриптор стає або в момент створення дескриптора правильним заповненням полів структури SECURITY_ATTRIBUTES, або шляхом копіювання вже існуючого дескриптора.

Поле bInheritHandle структури SECURITY_ATTRIBUTES повинне бути

встановлене у значення TRUE, а поле nLength повинне ініціюватися значенням, що повертає оператор sizeof(SECURITY_ATTRIBUTES).

У наступному фрагменті коду приведений типовий приклад створення наслідуваних файлових (або інших дескрипторів). У прикладі дескриптор захисту в структурі атрибутів захисту встановлений в NULL.

```
HANDLE h1, h2, h3;
SECURITY_ATTRIBUTES sa =
  {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
...
h1 = CreateFile(..., &sa, ...);  // Наслідуваний.
h2 = CreateFile(..., NULL, ...); // Ненаслідуваний.
h3 = CreateFile(..., &sa, ...);  // Наслідуваний.
// Застосоване повторне використання структури sa.
```

Але й цього ще не досить. Щоб дочірньому процесу стало відомо значення наслідуваного дескриптора, батьківський процес повинен передати це значення дочірньому процесу або через механізм міжпроцесної взаємодії, або шляхом призначення дескриптора стандартному пристрою вводу/виводу в структурі STARTUPINFO. На практиці частіше використовується другий метод, тому що він дозволяє перенаправляти ввод/вивід стандартним способом.

У випадку дескрипторів, які не є дескрипторами файлів, застосовується інший спосіб, при якому дескриптор перетворюється в текстовий формат і поміщається в командний рядок або у змінну оточення. Такий підхід можливий тільки, якщо дескриптор є наслідуваним.

Успадковані дескриптори являють собою окремі екземпляри. Тому батьківський і дочірній процеси можуть одержати доступ до того самого файла, використовуючи різні покажчики файлів, і кожний із процесів повинен самостійно закривати приналежний йому дескриптор.

GetProcessHandleCount

Незакриття дескрипторів після того, як необхідність у них відпала, найчастіше стає причиною витоку ресурсів, що призводить до зниження продуктив-

ності системи, збоям у програмі й навіть впливає на інші процеси. Тому в API ОС Windows існує функція, що дозволяє визначити кількість відкритих дескрипторів, що належать зазначеному процесу. Ця функція має наступний прототип:

```
BOOL GetProcessHandleCount(HANDLE hProcess,
  PDWORD pdwHandleCount)
```

3.1.2.4 Ідентифікатори процесів

GetCurrentProcess, *GetCurrentProcessId*

У випадку, якщо в батьківського процесу з'являється необхідність в одержані ідентифікатора або дескриптора дочірнього процесу, то це можна зробити, скориставшись структурою PROCESS_INFORMATION. Одержавши цю інформацію, процес, наприклад, може закрити дескриптор дочірнього процесу. Така операція не приводить до знищення дочірнього процесу, а тільки унеможливлює доступ до нього з боку батьківського. Інформацію про поточний процес повертають дві функції:

```
HANDLE GetCurrentProcess(VOID)
DWORD GetCurrentProcessId(VOID)
```

У дійсності функція GetCurrentProcess повертає псевдодескриптор, що не є наслідуваним. Це значення може використовуватися будь-яким процесом щораз, коли йому потрібний його власний дескриптор.

3.1.2.5 Дублювання дескрипторів

WinAPI передбачає створення копії дескрипторів у випадку коли батьківському й дочірньому процесам потрібен різний доступ до об'єкта, що ідентифікується наслідуваним дескриптором.

DuplicateHandle

Копія дескриптора з бажаними дозволами доступу й властивостями спадкування створюється функцією із прототипом:

```
BOOL DuplicateHandle(HANDLE hSourceProcessHandle,
  HANDLE hSourceHandle, HANDLE hTargetProcessHandle,
  LPHANDLE lphTargetHandle, DWORD dwDesiredAccess,
  BOOL bInheritHandle, DWORD dwOptions)
```

При вдалому завершенні функції покажчик lphTargetHandle указує на копію вихідного дескриптора – hSourceHandle, що є дескриптором об'єкта, що дублюється й повинен мати права доступу PROCESS_DUP_HANDLE. Якщо зазначеного дескриптора у вихідному процесі не існує, функція DuplicateHandle завершується помилкою. Новий дескриптор, на який указує покажчик lphTargetHandle, є дійсним у цільовому процесі, hTargetProcessHandle.

Параметр dwOptions може містити будь-яку комбінацію двох прапорів з наступними мнемонічними іменами:

1) DUPLICATE_CLOSE_SOURCE – викликає закриття вихідного дескриптора.
2) DUPLICATE_SAME_ACCESS – змушує ігнорувати параметр dwDesiredAccess.

Параметр dwDesiredAccess має дуже багато можливих значень і може бути скасований одним із прапорів параметра dwOptions.

Функція DuplicateHandle може застосовуватися до дескрипторів будь-якого типу.

3.1.2.6 Завершення й припинення виконання процесу

ExitProcess

Процес (точніше єдиний потік, що виконується в цьому процесі) завершує свою роботу викликом функції:

```
VOID ExitProcess(UINT uExitCode)
```

У загальному випадку ця функція не здійснює повернення у систему, а тільки завершує процес і всі його потоки. Виконання ж оператора return з кодом повернення в основній програмі рівносильне виклику функції ExitProcess з кодом завершення.

TerminateProcess

При необхідності, один процес може припинити виконання іншого процесу, якщо дескриптор останнього має права доступу PROCESS_TERMINATE. При виклику функції завершення процесу вказується код завершення.

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode)
```

При завершенні процесу, необхідно звільнити всі ресурси, які процес розділяв з іншими процесами, у тому числі ресурси синхронізації (мьютекси, семафори й події).

3.1.2.7 Очікування завершення процесу

Найпростішим, але й найбільш обмеженим методом синхронізації процесів є очікування одним з них завершення іншого.

WaitForSingleObject, *WaitForMultipleObject*

Стандартні функцій очікування Windows мають наступні властивості:

- функції очікування *застосовні до всіляких типів об'єктів* системи, дескриптори процесів – тільки окремий випадок їхнього застосування;
- функції можуть очікувати завершення одного процесу, першого з деякі-

лькох зазначених процесів або всіх процесів, що утворюють групу;

– існує можливість встановлювати кінцевий інтервал очікування.

Обидві розглянутих нижче функції очікують перехід об'єкта синхронізації в сигнальний стан. Наприклад, система переводить процес у сигнальний стан, коли він завершується або його виконання припиняється ззовні. Функціями очікування, котрі, як вже вказувалося, використовуються для всіляких об'єктів системи, є наступні функції:

```
DWORD WaitForSingleObject(HANDLE hObject,
  DWORD dwMilliseconds)
DWORD WaitForMultipleObjects(DWORD nCount,
  CONST HANDLE *lpHandles, BOOL fWaitAll,
  DWORD dwMilliseconds)
```

Значення, що повертається функціями визначає причину завершення очікування або, у випадку помилки, дорівнює 0xffffffff.

В аргументах цих функцій вказується або дескриптор одиночного процесу hObject, або дескриптори ряду окремих об'єктів, що зберігаються в масиві, на який вказує покажчик lpHandles. Значення параметра nCount, що обумовлює розмір масиву, у ніякому разі не повинне перевищувати значення MAXIMUM_WAIT_OBJECTS (за замовчуванням 64).

dwMilliseconds – інтервал очікування, що обчислюється у мілісекундах. Якщо значення цього параметра дорівнює 0, то повернення з функції здійснюється відразу ж після перевірки стану зазначеного об'єкта, що дозволяє програмі опитувати процеси для визначення їхнього стану завершення. Якщо ж значення цього параметра дорівнює INFINITE, то очікування триває доти, поки очікуваний процес не завершиться.

fWaitAll – параметр, що вказує на необхідність очікування завершення всіх процесів, а не тільки одного.

Значення, що можуть повертатися цією функцією у випадку її успішного завершення такі:

– WAIT_OBJECT_0 – зазначений об'єкт перейшов у сигнальний стан

(функція WaitForSingleObject) або одночасно всі nCount об'єктів перейшли в сигнальний стан (функція WaitForMultipleObject і параметр fWaitAll дорівнює TRUE);

- WAIT_OBJECT_0+n (де $0 \le n < nCount$) – для визначення дескриптора завершеного процесу потрібно від повернутого значення відняти значення WAIT_OBJECT_0, якщо в сигнальний стан перейшли кілька об'єктів – вертається дескриптор, що має найменше значення;
- WAIT_TIMEOUT – протягом зазначеного часу очікування об'єкт (об'єкти), для яких очікується перехід у сигнальний стан, не виконали умову очікування;
- WAIT_FAILED – невдале завершення функції, наприклад, у дескриптора відсутні права доступу SYNCHRONIZE.
- WAIT_ABANDONED_0 – це значення неможливо у випадку процесів, а тільки для об'єктів типу мьютексів.

Наведена нижче як приклад програма, використовуючи описані вище функції керування процесами, створює процеси для пошуку текстового шаблона у файлах, по одному процесу на кожний файл. Як дочірні процеси використовується програма grep, але аналогічна програма може бути використана з будь-якою іншою програмою, що використовує стандартний вивід. Тобто програму дочірнього процесу можна розглядати як програму, що виконується, і виконання якої контролюється батьківським процесом.

Програма виконує наступні дії:

- для пошуку зазначеного шаблона в кожному із вхідних файлів використовується окремий процес, що запускає на виконання той самий модуль, для чого кожний процес запускається командним рядком, у якій вказується ім'я процесу (grep), шаблон для розшуку і ім'я файла;
- полю hStdOut структури STARTUPINFO нового процесу привласнюється значення дескриптора тимчасового файла, котрий обов'язково визначається як наслідуваний;
- с використанням функції WaitForMultipleObjects організується очіку-

вання завершення всіх процесів пошуку;

- по завершенні всіх процесів пошуку, запуском утиліти cat (утиліта не є системною, а являє собою окремо написану програму) по черзі здійснюється вивід результатів роботи, котрі зберігаються у тимчасових файлах;
- успішність спроби знаходження шаблона процесом визначається по коду завершення процесу grep.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <io.h>
int _tmain(DWORD argc, LPTSTR argv[])
/* Для виконання пошуку в кожному з файлів, зазначених у командному рядку, створюється окремий процес. Кожному процесу надається тимчасовий файл у поточному каталозі, у якому зберігаються результати. */
  {
    HANDLE hTempFile;
// Атрибути захисту для наслідуваного дескриптора
    SECURITY_ATTRIBUTES StdOutSA =
      {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
    TCHAR CommandLine[MAX_PATH + 100];
    STARTUPINFO StartUpSearch, StartUp;
    PROCESS_INFORMATION ProcessInfo;
    DWORD iProc, ExCode;
// Покажчик на масив дескрипторів процесів
    HANDLE *hProc;
    typedef struct
      {TCHAR TempFile[MAX_PATH];} PROCFILE;
// Покажчик на масив імен тимчасових файлів
    PROCFILE *ProcFile;
    GetStartupInfo(&StartUpSearch);
    GetStartupInfo(&StartUp);
    ProcFile = malloc((argc - 2) * sizeof(PROCFILE));
    hProc = malloc((argc - 2) * sizeof(HANDLE));
// Створення для кожного файла окремого процесу "grep"
    for (iProc = 0; iProc < argc - 2; iProc++)
      {
```

```
        _stprintf(CommandLine, _T("%s%s %s"),
          _T("grep "), argv[1], argv[iProc + 2]);
// Тимчасові файли для зберігання результатів пошуку
        GetTempFileName(_T("."), _T("gtm"),
          0, ProcFile[iProc].TempFile);
// Цей дескриптор є наслідуваним
        hTempFile =
            CreateFile(ProcFile[iProc].TempFile,
                   GENERIC_WRITE,
                   FILE_SHARE_READ | FILE_SHARE_WRITE,
                   &StdOutSA, CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL, NULL);
        StartUpSearch.dwFlags = STARTF_USESTDHANDLES;
        StartUpSearch.hStdOutput = hTempFile;
        StartUpSearch.hStdError = hTempFile;
        StartUpSearch.hStdInput =
            GetStdHandle(STD_INPUT_HANDLE);
// Створюється процес для виконання командного рядка
        CreateProcess(NULL, CommandLine, NULL, NULL,
                      TRUE, 0, NULL, NULL,
                      &StartUpSearch, &ProcessInfo);
// Закрити непотрібні дескриптори
        CloseHandle(hTempFile);
        CloseHandle(ProcessInfo.hThread);
        hProc[iProc] = ProcessInfo.hProcess;
      }
/* Виконати всі процеси й дочекатися завершення
   кожного з них */
    for (iProc = 0; iProc < argc - 2;
         iProc += MAXIMUM_WAIT_OBJECTS)
/* Дозволити використання досить великої
   кількості процесів */
      WaitForMultipleObjects(min(MAXIMUM_WAIT_OBJECTS,
                     argc - 2 - iProc), &hProc[iProc],
                     TRUE, INFINITE);
/* Переслати результуючі файли на стандартний вивід
   с використанням утиліти cat */
    for (iProc = 0; iProc < argc - 2; iProc++)
      {
        if (GetExitCodeProcess(hProc[iProc],
            &ExCode) && ExCode==0)
          {
// Якщо шаблон виявлений - вивести результати
            if (argc > 3)
              _tprintf(_T("%s:\n"), argv[iProc + 2]);
```

```
/* Використання стандартного виводу декількома
   процесами */
            fflush(stdout);
            _stprintf(CommandLine, _T("%s%s"),
                _T("cat "), ProcFile[iProc].TempFile);
            CreateProcess(NULL, CommandLine,
                NULL, NULL, TRUE, 0, NULL,
                NULL, &StartUp, &ProcessInfo);
            WaitForSingleObject(ProcessInfo.hProcess,
                                INFINITE);
            CloseHandle(ProcessInfo.hProcess);
            CloseHandle(ProcessInfo.hThread);
          }
        CloseHandle(hProc [iProc]);
        DeleteFile(ProcFile[iProc].TempFile);
      }
     free(ProcFile);
     free(hProc);
     return 0;
  }
```

3.1.2.8 Часові характеристики процесу

API ОС Windows надає можливість одержання різних часових характеристик процесу:

- минулий час (elapsed time);
- час, витрачений ядром (kernel time);
- користувальницький час (user time).

GetProcessTimes

Всі ці параметри можна одержати, скориставшись функцією з таким прототипом:

```
BOOL GetProcessTimes(HANDLE hProcess,
  LPFILETIME lpCreationTime, LPFILETIME lpExitTime,
  LPFILETIME lpKernelTime, LPFILETIME lpUserTime)
```

Дескриптор процесу може посилатися як на процес, що продовжує вико-

нуватися, так і на процес, виконання якого припинилося. Минулий час обчислюється шляхом вирахування часу створення процесу із часу завершення процесу. Інші часові інтервали функція визначає самостійно й повертає через відповідні покажчики. Тип даних FILETIME є 64-бітовим і для обчислення зазначеної різниці в структурі типу union необхідно об'єднати змінну цього типу зі змінною типу LARGE_INTEGER.

Для потоків існує аналогічна функція GetThreadTimes, але їй як параметр передається дескриптор потоку.

Код програми, що наведений нижче, демонструє одержання часових характеристик процесів.

Одним з можливих застосувань цієї команди, наприклад, може бути здійснення порівняльного аналізу часу виконання й ефективності різних версій функцій копіювання й перетворення файлів з ASCII в Unicode.

У даній програмі використовується функція Windows GetCommandLine, що повертає повноцінний командний рядок, а не окремі складові рядка з масиву argv.

Крім того, програма використовує допоміжну функцію SkipArg, що переглядає командний рядок і встановлює в ній покажчик у позицію, що безпосередньо іде за ім'ям файла, що буде виконуватися. За для економії місця вихідний код функції SkipArg у прикладі не наводиться.

Для визначення версії ОС у програмі використовується функція GetVersionEx. В застарілих операційних системах такою програмою можна лише обчислити минулий час процесу. Програмний код для цих систем представлений з тією метою, щоб показати, що в деяких випадках працездатність програм, принаймні із частковим збереженням їхньої функціональності, вдається забезпечити для цілого діапазону різних версій Windows.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <malloc.h>
#include <io.h>
int _tmain(int argc, LPTSTR argv[]) {
  STARTUPINFO Startup;
  PROCESS_INFORMATION ProcInfo;
// Структура для виконання арифметичних операцій
  union {
    LONGLONG li;
    FILETIME ft;
    } CreateTime, ExitTime, ElapsedTime;
  FILETIME KernelTime, UserTime;
  SYSTEMTIME ElTiSys, KeTiSys, UsTiSys,
     StartTimeSys, ExitTimeSys;
  LPTSTR targv = SkipArg(GetCommandLine());
  OSVERSIONINFO OSVer;
  BOOL IsNT;
  HANDLE hProc;
  OSVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  GetVersionEx(&OSVer);
  IsNT = (OSVer.dwPlatformId == VER_PLATFORM_WIN32_NT);
  GetStartupInfo(&StartUp);
  GetSystemTime(&StartTimeSys);
// Виконати командний рядок;
// дочекатися завершення процесу
  CreateProcess (NULL, targv, NULL, NULL, TRUE,
                 NORMAL_PRIORITY_CLASS, NULL,
                 NULL, &StartUp, &ProcInfo);
//Перевірка наявності ВСІХ НЕОБХІДНИХ прав доступу
  DuplicateHandle(GetCurrentProcess(),
                  ProcInfo.hProcess,
                  GetCurrentProcess(), &hProc,
                  PROCESS_QUERY_INFORMATION |
                  SYNCHRONIZE, FALSE, 0);
  WaitForSingleObject(hProc, INFINITE);
  GetSystemTime (&ExitTimeSys);
// Обчислення часових інтервалів для процесу
  if (IsNT) {
    GetProcessTimes(hProc, &CreateTime.ft,
                    &ExitTime.ft, &KernelTime,
                    &UserTime);
  ElapsedTime.li = ExitTime.li - CreateTime.li;
  FileTimeToSystemTime(&ElapsedTime.ft, &ElTiSys);
  FileTimeToSystemTime(&KernelTime, &KeTiSys);
  FileTimeToSystemTime(&UserTime, &UsTiSys);
  _tprintf(_T("Минулий час: %02d:%02d:%02d:%03d\n"),
```

```
        ElTiSys.wHour, ElTiSys.wMinute,
        ElTiSys.wSecond, ElTiSys.wMilliseconds);
_tprintf(_T("Користув. час:%02d:%02d:%02d:%03d\n"),
        UsTiSys.wHour, UsTiSys.wMinute,
        UsTiSys.wSecond, UsTiSys.wMilliseconds);
_tprintf(_T("Системний час:%02d:%02d:%02d:%03d\n"),
        KeTiSys.wHour, KeTiSys.wMinute,
        KeTiSys.wSecond, KeTiSys.wMilliseconds);
} else {
_tprintf(_T("ОС не сімейства NT"));
}
CloseHandle(ProcInfo.hThread);
CloseHandle(ProcInfo.hProcess);
CloseHandle(hProc);
return 0;
}
```

Таку програму можна, наприклад, використовувати для аналізу продуктивності різних варіантів програм сортування даних, копіювання й перетворення файлів.

3.2 Завдання на самостійну роботу

По основній і додатковій літературі ознайомитися з функціями створення і роботи з процесами в ОС Linux і Windows. Знати принципи керування процесами в цих системах та відмінності в їх створенні та функціонуванні.

Підготувати відповіді на контрольні запитання до роботи.

Згідно до варіанта завдання, одержаного від викладача, підготувати вступну частину протоколу лабораторної роботи, де в ***обов'язковому порядку привести блок-схему роботи програми***.

3.3 Варіанти завдань до лабораторної роботи

У зв'язку з деякими складностями в створенні процесів у ОС Windows і рідкому їх вживанню у практичному програмуванні, завдання лабораторної роботи виконуються лише в середовищі ОС Linux із застосуванням відповідних засобів цієї системи.

До всіх варіантів завдання лабораторної роботи слід розробити допоміжну програму у окремому виконуваному файлі з ім'ям integral на диску. Програма повинна обчислювати визначений інтеграл $I = \int_a^b f(x)\,dx$, вигляд функції і межі інтегрування вказані у варіанті. У варіанті також вказані крок і спосіб чисельного інтегрування. Діапазон інтегрування та крок *обов'язково* передати до програми як параметри командного рядка системи. Зв'язування програми при компіляції необхідно проводити з використанням ключа -lm (підключення математичної бібліотеки). Ця програма потім буде запускатись як окремий процес для роботи у паралельному режимі.

Відповідно до варіанта завдання за допомогою виклику fork() необхідно створити дерево процесів згідно зі схемою наданою на рис. 3.3 а) або б). На схемі: P (Parent) – батьківський процес; C та C1 (Child) – дочірні процеси. Таким чином у системі у паралельному режимі одночасно повинні працювати три процеси P, C і C1.

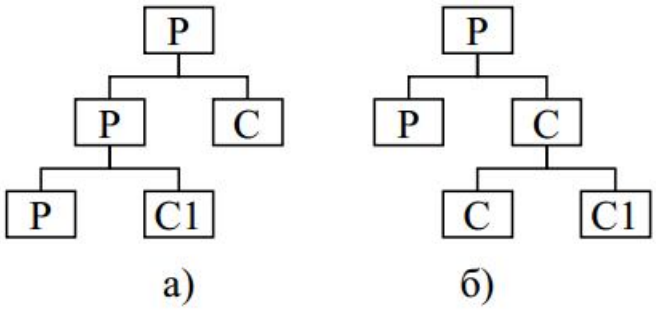

Рисунок 3.3 – Схеми створюваних дерев процесів

1) Програма integral методом Сімпсона обчислює інтеграл від функції $f(x) = x^2 \sin x$ у межах від a=0 до b=315 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execl підмінює дочірній процес C. Дочірній процес C1 з точністю 1e-8 обчислює і виводить на друк на-

ступну суму $\sum_{n=0}^{\infty}\frac{1}{2^n}$. Батьківський процес P очікує завершення процесу C1 і, якщо він завершився нормально із кодом завершення 0, змушує завершитись другий дочірній процес C шляхом посилки до нього сигналу переривання SIGINT після чого сам завершує роботу. Дерево процесів відповідає рис. 3.3а).

2) Програма integral методом трапецій обчислює інтеграл від функції $f(x)=\frac{\sin x\,\cos x}{x}$ у межах від a=0 до b=1000 з кроком h=0.1, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної *x*. Програма за допомогою системного виклику execlp підмінює дочірній процес C. Дочірній процес C1 з точністю 1e-8 обчислює і виводить на друк наступну суму $\sum_{n=0}^{\infty}(-1)^n\frac{1}{2^n}$ після чого генерує сигнал alarm з часом затримки у одну секунду. Батьківський процес P після створення дочірнього процесу C переходить у стан очікування сигналу будильника і по одержанні його відправляє до процесу C сигнал SIGINT після чого сам завершує роботу. Дерево процесів повинне відповідати рис. 3.3б).

3) Програма integral методом прямокутників обчислює інтеграл від функції $f(x)=\ln^2 x$ у межах від a=0 до b=100 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної *x*. Програма за допомогою системного виклику execle підмінює дочірній процес C1. Дочірній процес C з точністю 1e-4 обчислює і виводить на друк наступну суму $\sum_{n=0}^{\infty}(-1)^n\frac{1}{2n+1}$. Батьківський процес P після формування обох дочірніх процесів призупиняє своє виконання на дві секунди, а потім перевіряє їх закінчення. Якщо якійсь з процесів або обидва не закінчились, вони закінчуються примусово посилкою до них сигналу SIGINT. Дерево процесів відповідає рис. 3.3а).

4) Програма integral методом Сімпсона обчислює інтеграл від функції

$f(x)=\frac{\sin x}{x^2}$ у межах від a=0 до b=315 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної *x*. Програма за допомогою системного виклику execv дочірній процес C1. Дочірній процес C з точністю 1e-5 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty}\frac{1}{n(n+1)}$. Батьківський процес P очікує завершення будь-якого дочірнього процесу. Після цього три рази з інтервалом в одну секунду перевіряє закінчення другого дочірнього процесу і, якщо той не завершився, закінчує його примусово. Дерево процесів повинне відповідати рис. 3.3б).

5) Програма integral методом трапецій обчислює інтеграл від функції $f(x)=\frac{\cos x}{1+x^2}$ у межах від a=0 до b=1000 з кроком h=0.1 і за допомогою системного виклику execvp підмінює дочірній процес C. Дочірній процес C1 попередньо генерує сигнал alarm з часом затримки дві секунди, а потім з точністю 1e-5 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty}\frac{1}{(2n-1)(2n+1)}$. Батьківський процес P очікує одержання сигналу від процесу C1 потім перевіряє його завершення і, якщо він не завершився, завершує його посиланням сигналу SIGINT. Із затримкою в одну секунду теж саме робить з другим дочірнім процесом C. Дерево процесів відповідає рис. 3.3а).

6) Програма integral методом прямокутників обчислює інтеграл від функції $f(x)=\frac{\ln x}{x^2}$ у межах від a=0 до b=100 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execve підмінює дочірній процес C. Дочірній процес C1 попередньо генерує сигнал alarm з часом затримки три секунди, а потім з точністю 1e-5 обчислює і виводить на друк наступну суму $\sum_{n=2}^{\infty}\frac{1}{(n-1)(n+1)}$. Батьківський процес P очікує одержання сигналу від процесу C1 потім перевіряє його завершення і, якщо він не завершився, за-

вершує його посиланням сигналу SIGINT. Із затримкою в дві секунду теж саме робить з другим дочірнім процесом C1. Дерево процесів відповідає рис. 3.3б).

7) Програма integral методом Сімпсона обчислює інтеграл від функції $f(x) = x^3 \cos x$ у межах від a=0 до b=315 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execl підмінює дочірній процес C1. Дочірній процес C з точністю 1e-5 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty} \frac{1}{n(n-1)(n+1)}$. Батьківський процес P очікує завершення процесу C і, якщо він завершився нормально із кодом завершення 0, змушує завершитись другий дочірній процес C1 шляхом посилки до нього сигналу переривання SIGINT після чого сам завершує роботу. Дерево процесів відповідає рис. 3.3а).

8) Програма integral методом трапецій обчислює інтеграл від функції $f(x) = \frac{x}{e^x + 1}$ у межах від a=0 до b=1000 з кроком h=0.1, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execlp підмінює дочірній процес C1. Дочірній процес C з точністю 1e-6 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty} \frac{1}{n^2}$ після чого генерує сигнал alarm з часом затримки у одну секунду. Батьківський процес P після створення дочірнього процесу C переходить у стан очікування сигналу будильника і по одержанні його відправляє до процесу C сигнал SIGINT після чого сам завершує роботу. Дерево процесів відповідає рис. 3.3б).

9) Програма integral методом прямокутників обчислює інтеграл від функції $f(x) = x^2 \ln x$ у межах від a=0 до b=100 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execle підмінює дочірній процес C. Дочірній процес C1 з точністю 1e-6 обчислює і виводить на друк

наступну суму $\sum_{n=1}^{\infty}(-1)^n \frac{1}{n^2}$. Батьківський процес P після формування обох дочірніх процесів призупиняє своє виконання на дві секунди, а потім перевіряє їх закінчення. Якщо якійсь з процесів або обидва не закінчились, вони закінчуються примусово посилкою до них сигналу SIGINT. Дерево процесів відповідає рис. 3.3а).

10) Програма integral методом Сімпсона обчислює інтеграл від функції $f(x) = \frac{\cos x}{x^3}$ у межах від a=0 до b=315 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execv підмінює дочірній процес C. Дочірній процес C1 з точністю 1e-6 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty}\frac{1}{(2n-1)^2}$. Батьківський процес P очікує завершення будь-якого дочірнього процесу. Після цього три рази з інтервалом в одну секунду перевіряє закінчення іншого дочірнього процесу і, якщо той не завершився, закінчує його примусово. Дерево процесів відповідає рис. 3.3б).

11) Програма integral методом трапецій обчислює інтеграл від функції $f(x) = \frac{x}{e^x - 1}$ у межах від a=0 до b=1000 з кроком h=0.1, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execvp підмінює дочірній процес C1. Дочірній процес C попередньо генерує сигнал alarm з часом затримки дві секунди, а потім з точністю 1e-8 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty}(-1)\frac{1}{n^4}$. Батьківський процес P очікує одержання сигналу від процесу C потім перевіряє його завершення і, якщо він не завершився, завершує його посиланням сигналу SIGINT. Із затримкою в дві секунди теж саме робить з другим дочірнім процесом C1. Дерево процесів відповідає рис. 3.3а).

12) Програма integral методом прямокутників обчислює інтеграл від фун-

кції $f(x)=\frac{\ln^2 x}{x}$ у межах від a=0 до b=100 з кроком h=0.01, на кожному кроці інтегрування на термінал повинне виводитись поточне значення інтеграла та змінної x. Програма за допомогою системного виклику execve підмінює дочірній процес C1. Дочірній процес C з точністю 1e-8 обчислює і виводить на друк наступну суму $\sum_{n=1}^{\infty}(-1)^{n-1}\frac{1}{n^4}$ після чого генерує сигнал alarm з часом затримки у одну секунду. Батьківський процес P після створення дочірнього процесу C переходить у стан очікування сигналу будильника і по одержанні його відправляє до процесу C1 сигнал SIGINT після чого сам завершує роботу. Дерево процесів відповідає рис. 3.3б).

Інтегрування у варіантах необхідно виконувати за такими формулами:

- за методом прямокутників – $I=h\sum_{i=0}^{N} f\left(x_i+\frac{h}{2}\right),\ N=\frac{b-a}{h},\ x_i=a+hi;$
- за методом трапецій – $I=\frac{h}{2}\sum_{i=0}^{N} f(x_i)+f(x_{i+1}),\ N=\frac{b-a}{h},\ x_i=a+hi;$
- за методом Сімпсона – $I=\frac{h}{3}\left(f(a)+f(b)+\sum_{i=1}^{N-1}2((i \bmod 2)+2)f(x_i)\right),$ $N=\frac{b-a}{h},\ x_i=a+hi.$

3.4 Контрольні питання

1) Що таке процеси і сигнали, для чого вони потрібні?
2) Поясніть принцип сумісного використання процесами коду и бібліотек.
3) Що таке ідентифікатор процесу (PID)?
4) Яки ресурси процесів не можуть використовуватися спільно?
5) Опишіть організацію процесів у ОС Linux.
6) Чим обмежується кількість процесів, що підтримуються системою Linux?

7) Коли і як запускається і навіщо потрібен процес init?
8) Як працює планувальник процесів у Linux, що це за параметр nice?
9) Назвіть можливості створення нових процесів у Linux.
10) Поясніть роботу системного виклику system і опишіть які параметри до нього передаються.
11) Чому використання виклику system не зовсім зручно при запуску процесу?
12) Опишіть альтернативний спосіб створення процесу його заміною.
13) Скільки варіантів виклику exec існує і в чому вони відрізняються один від одного?
14) Чи існують які-небудь окремі рекомендації для запуску програми за допомогою функцій exec, або можна вибирати будь-яку функцію?
15) Пояснить різницю у роботі між викликами system і exec.
16) У чому полягає основний недолік застосування system і exec і як його усувають у ОС Linux?
17) Поясніть дії системного виклику fork при дублюванні процесу.
18) Опишіть типову схему використання виклику fork.
19) Як розподіляються простори імен до виклику fork і після його використання?
20) Як можна змусити батьківський процес призупинитись і дочекатися завершення будь-якого дочірнього процесу?
21) Що допомагає батьківському процесу визначити статус завершення дочірнього процесу?
22) Яки засоби дозволяють інтерпретувати інформацію про стан процесу? Приведіть декілька прикладів такої інтерпретації.
23) Як дочекатись завершення не будь-якого дочірнього процесу, а деякого певного?
24) Для чого слугує аргумент options у виклику waitpid? Яке значення його застосується найбільш часто?
25) Яка обов'язкова умова накладається на дочірні процеси, котрі створені

із застосуванням системного виклику fork?

26) Що таке процес-зомбі, як він виникає, як запобігти його виникненню і як від нього позбавитись якщо він все ж виник?
27) Що таке сигнали у UNIX-сумісних системами, як вони виникають?
28) Назвіть деякі імена (4, 5) основних сигналів, що задаються у заголовному файлі signal.h, і опишіть їх дію.
29) Розкажіть про функцію обробки сигналів у програмі – signal. Як понімати другий параметр у виклику signal?
30) Чому завжди необхідно користуватися іменами сигналів, а не їхніми числовими значеннями?
31) Опишіть функцію для обробки сигналів, котру рекомендують сучасні стандарти X/Open і специфікації UNIX.
32) Для чого застосовується структура sigaction і які елементи (як мінімум) в неї містяться? Опишіть ці елементи.
33) Назвіть та опишіть деякі засоби програмної відправки сигналів.
34) Для чого у багатокористувальницькому середовищі служить застосування сигналів і припинення виконання програми?
35) Що собою являє процес в операційному середовищі Windows, у чому його відмінність від процесів UNIX?
36) У якому смислі у ОС Windows говорять про процеси-предки і процеси-нащадки?
37) Які є механізми ідентифікації процесів у Windows?
38) Назвіть ресурси, що належать кожному процесу з точки зору програміста.
39) Якою функцією у Windows створюється новий процес? Розкажіть про цю функцію.
40) Яким чином прапорці-константи CREATE_SUSPENDED, DETACHED_PROCESS і CREATE_NEW_CONSOLE обумовлюють параметри нового процесу й потоку, що створюються?
41) Для чого потрібна структура PROCESS_INFORMATION? Розкажіть

про поля, котрі вона містить.

42) Як задається ім'я файла модуля, котрий буде виконуватися, як новий процес?

43) Яким чином у дочірній процес найзручніше передати дескриптори стандартних приладів вводу/виводу?

44) Опишіть крокі, котрі необхідно створити для перетворення дескриптора деякого ресурсу в наслідуваний.

45) Як у дочірній процес передаються наслідувані дескриптори у випадку якщо вони не є дескрипторами файлів?

46) Як батьківській процес може одержати ідентифікатор або дескриптор дочірнього процесу і як цю інформацію одержати для поточного процесу?

47) Навіщо і як створюється копія дескриптора з бажаними дозволами до доступу?

48) Як завершити і призупинити на деякий час виконання процесу?

49) Назвіть найпростіший метод синхронізації процесів, опишіть властивості функцій, що виконують таку синхронізацію.

50) Які часові характеристики процесу можна одержати при його виконанні? Назвіть та опишіть функцію API, котра дає можливість одержати ці характеристики.

51) Розкажіть, що необхідно зробити у лабораторній роботі згідно з вашим варіантом.

ЛАБОРАТОРНА РОБОТА № 4
ПОТОКИ І ЇХ СИНХРОНІЗАЦІЯ В ОС LINUX І WINDOWS

Література до роботи: [1] стор. 70 – 75, 82 – 85, 110 – 148
[3] стор. 106 – 149, [4] стор. 68 – 174

МЕТА РОБОТИ

Вивчення на практиці наступних питань:

- створення нових потоків у процесі;
- синхронізацію доступу до даних різних потоків одного процесу;
- керування в тому самому процесі одним потоком з іншого.

4.1 Огляд теоретичної частини. Поняття потоку, їх достоїнства й недоліки

Попередня лабораторна робота (№3) була присвячена роботі процесів, але накладні витрати на створення нового процесу як в ОС Linux, так і особливо в Windows занадто великі. Тому в обох ОС передбачене застосування ***потоків***, що дозволяє навіть одному процесу стати багатозадачним.

Як і в попередніх роботах, спочатку будуть розглянуті потоки POSIX в Linux (UNIX), а потім в Windows.

Потік – це послідовність або цикл керування в процесі.

Найчастіше створення нового потоку має явно виражені переваги в порівнянні зі створенням нового процесу. Це пов'язане з тим, що накладні витрати при створенні нового потоку у значній більшості випадків істотно менше, ніж при створенні нового процесу. Особливо це відноситься до ОС Windows, хоча створення нових процесів в Linux дуже ефективно в порівнянні з іншими операційними системами.

До достоїнств потоків можна віднести наступне.

1) Продуктивність додатка, у якому змішані ввід, обчислення й вивід, можна підвищити, запустивши ці операції як три окремих потоки.

2) Застосування множинних потоків усередині процесу може при наявності відповідного додатка дозволити одному процесу краще використовувати доступні апаратні ресурси в комп'ютерах із багатоядерними процесорами.
3) Перемикання між потоками вимагає від операційної системи набагато менше зусиль, чим перемикання між процесами, і з ними набагато зручніше виконувати програми з декількома потоками виконання в однопроцесорних системах.

У потоків є й недоліки.

1) Створення багатопоточної програми вимагає дуже ретельної розробки. Імовірність появи незначних часових збоїв або помилок, викликаних ненавмисним спільним використанням змінних, у такій програмі досить значна.
2) Налагодження багатопоточної програми є набагато більш сутужнім, ніж налагодження одного потоку виконання, оскільки взаємозв'язки потоків дуже важко контролювати.
3) Програма, у якій громіздкі обчислення розділені на дві частини, і ці дві частини виконуються як окремі потоки, необов'язково буде працювати швидше на машині з одним процесором.

4.1.1 Потоки в ОС Linux

ОС Linux, як і багато інших сучасних операційних систем, цілком здатна виконувати множинні процеси одночасно. Але в дійсності у всіх процесів є як мінімум один потік виконання.

Принциповим є розходження між системним викликом fork і створенням нових потоків. Коли процес виконує системний виклик fork, створюється нова копія процесу з її власними змінними й власним PID. Час виконання цього нового процесу планується незалежно й, частіш за все, він виконується незалежно від його батьківського процесу. Коли створюється новий потік, цей потік виконання одержує власний стек (й, відповідно, локальні змінні), але використовує

разом з його батьківським процесом глобальні змінні, файлові дескриптори, оброблювачі сигналів і поточний каталог.

Стандарт POSIX 1003.1C стандартизував процедури роботи з потоками, і в даний час вони реалізовані практично у всіх ОС сімейства Linux. Багатоядерні процесори вже є звичайними для персональних комп'ютерів, і в більшості машин є низькорівнева апаратна підтримка, котра дозволяє їм виконувати кілька потоків одночасно. Раніше при наявності одноядерних процесорів одночасне виконання потоків було лише просто ефективною ілюзією.

Існує ціла низка бібліотечних викликів, пов'язаних з потоками, більшість ідентифікаторів цих викликів починається із префікса pthread. Для застосування цих бібліотечних викликів у програмі необхідно включити заголовний файл pthread.h, а компонування програми робити з використанням опції -lpthread. У деяких ОС система програмування вимагає також визначити спеціальний макрос _REENTRANT (це повинне робитись до того, як будуть об'явлені будь-які директиви #include), а так само визначення _POSIX_C_SOURCE, але у більшості випадків в цьому немає необхідності. Таки вимоги обумовлені тим, що при роботі із процесами необхідно використовувати, так звані, реентерабельні функції бібліотеки С. Реентерабельний програмний код може викликатися кілька разів або з різних потоків, або викликами, котрі якимсь чином вкладені один до одного, й при цьому працювати коректно. Отже, реентерабельна частина програмного коду, як правило, повинна застосовувати локальні змінні таким чином, щоб кожний з будь-яких викликів коду одержував власну унікальну копію даних.

4.1.1.1 Створення потоку

Для успішної роботи з потоками конче необхідно розглянути функції, що виконують керування ними: створення, завершення й таке інше.

pthread_create

Новий потік створюється функцією pthread_create.

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Прототип функції більш складний, ніж у виклику fork, але застосування функції є досить простим. Перший аргумент – покажчик на змінну типу pthread_t. В області пам'яті на яку вказує цей покажчик, при створенні потоку міститься ***ідентифікатор потоку***, призначений для посилань на потік. Наступний аргумент задає атрибути потоку. Як що немає потреби в особливих атрибутах, можна просто передати в цьому аргументі null. У третьому формальному аргументі виклику – void *(*start_routine)(void *) потокові передається адреса функції, що як параметр повинна приймати не типізований покажчик void, і повертати теж покажчик на void. Цю функцію потік повинен почати виконувати після створення. Таким чином можна передати єдиний аргумент будь-якого типу й повернути покажчик на будь-який тип. У четвертому, останньому параметрі вказуються аргументи, які потрібно передати цій функції, якщо вони їй потрібні.

Функція створення потоку, у випадку успішного завершення, повертає значення, яке, як прийнято, дорівнює 0 і, у противному випадку, вертається номер помилки.

pthread_exit

При завершенні потік повинен викликати функцію pthread_exit – декотрий аналог функції exit для процесу. Ця функція завершує потік, котрий її викликав, і повертає покажчик на об'єкт. Функція pthread_exit оголошується в такий спосіб:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Ніколи не можна використовувати функцію з поверненням покажчика на локальну змінну тому, що така змінна перестає існувати коли потік завершується при виникненні серйозної помилки.

pthread_join

Функція pthread_join – для потоків є еквівалентом функції wait, котру застосовують процеси для очікування своїх дочірніх процесів. Вона оголошується так:

```
#include <pthread.h>
int pthread_join(pthread_t th, void** thread_return);
```

Перший параметр – це ідентифікатор потоку, який відслідковується. Ідентифікатор, як відомо, вертається через покажчик – перший параметр у функції pthread_create при створенні потоку. Другий аргумент – покажчик на покажчик, що вказує на значення, котре повертається з потоку. Функція повертає нуль у випадку успішного завершення й код помилки у разі аварійної ситуації.

Нижче наведена проста програма, що створює один додатковий потік, демонструє спільне використання змінних обома потоками (вже існуючим у процесі й щойно створеним) й змушує новий потік повернути результат вихідному потоку.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void *thread_function(void *arg);
char message[] = "Hello World";

int main() {
  int res;
  pthread_t a_thread;
  void *thread_result;
```

```
  res = pthread_create(&a_thread, NULL,
                       thread_function,
                       (void *)message);
  if (res ! = 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  printf("Waiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join-failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread-joined, it returned %s\n",
         (char *)thread_result);
  printf("Message is now %s\n", message);
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
  printf("thread_function is running. Argument was
           %s\n", (char *)arg);
  sleep(3);
  strcpy(message, "Bye!");
  pthread_exit("Thank you for the CPU time");
}
```

На початку цієї програми оголошується прототип функції, котра буде викликана потоком після його створення. Цей прототип повністю відповідає оголошенню третього параметра в прототипі функції pthread_create (приймає покажчик на void, а повертає також покажчик на void).

У функції main оголошується декілька змінних і потім здійснюється виклик функції pthread_create, щоб почати виконання нового потоку.

Після нормального створення потоку в процесі тепер виконуються два потоки. Вихідний потік (main) триває й виконує код, розташований слідом за функцією pthread_create, а новий потік починає виконання функції, котра має назву thread_function (звісно, що ім'я цієї функції може бути цілком довільним).

Вихідний потік викликає функцію pthread_join, що перш ніж повернути

керування, очікує завершення потоку з ідентифікатором a_thread, після чого виводиться значення, що повертається з потоку, а також вміст змінної message, котра була змінена у потоці, наприкінці програми робота процесу й двох його потоків припиняється.

Нижче наведений ще один приклад програми з потоками, але тут уже перевіряється одночасне виконання двох потоків. Логіка роботи програми заснована на спільному використанні змінних обома потоками в процесі (це не стосується локальних змінних у функції потоку).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
int run_now = 1;
char message[] = "Hello World";

int main() {
  int res;
  pthread_t a_thread;
  void *thread_result;
  int print_count1 = 0;
  res = pthread_create(&a_thread, NULL,
              thread_function, (void *)message);
  if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
  }
  while(print_count1++ < 20) {
    if (run_now == 1) {
      printf("1");
      run_now = 2;
    } else {
      sleep(1);
    }
  }
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join failed");
```

```
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
  int print_count2 = 0;
  while(print_count2++ < 20) {
    if (run_now == 2) {
      printf("2");
      run_now = 1;
    } else {
      sleep(1);
    }
  }
  sleep(3);
}
```

Програма являє собою незначно змінений приклад першої програми. У ній два потоки (один main, а другий у функції потоку), працюючи паралельно, змінюють значення змінної run_now, тим самим змушуючи її приймати значення, котре дорівнює 1 у першому потоці й 2 у другому.

Причому кожний з потоків, перш ніж змінити значення цієї змінної перевіряє його, і якщо воно не було змінено в паралельному потоці, то процес, котрий провадить перевірку, засипає на 3 секунди. Таким чином, здійснюється синхронізація роботи потоків. Цей прийом називається ***циклом активного*** або ***діяльного очікування*** (busy wait).

4.1.1.2 Синхронізація потоків

На жаль, такий процес синхронізації потоків є досить неефективним через цілу низку обставин. Тому на практиці для синхронізації потоків застосовують ряд функцій, спеціально розроблених для цих цілей.

До таких методів синхронізації належать використання ***семафорів***, котрі виступають у ролі сторожів, що охороняють фрагменти коду, а також ***м'ютексів*** (mutex – скорочення від mutual exclusions – взаємні виключення) – об'єктів,

котрі у дечому схожі на семафори і діють як спеціальні пристрої взаємного виключення для захисту фрагментів програмного коду. Ці методи досить схожі, і один може бути описаний у термінах іншого. Але вибір того чи іншого методу синхронізації визначається семантикою розв'язуваного завдання. В одних випадках більш ефективним є використання м'ютексів, наприклад, керування доступом до деякої області спільно використовуваної пам'яті, до якої у кожний окремий момент часу може звертатися тільки один потік. Але у задачах керування доступом до низки ідентичних об'єктів у цілому, наприклад, надання потокові одного USB-порту з набору, що включає п'ять доступних портів, більше підходить семафор.

Для семафорів існують два набори інтерфейсних функцій: один з них, в основному, застосовується для синхронізації процесів, другий набір включений до стандарту POSIX Real-time Extensions і застосовується для потоків. Тому надалі описаним є тільки другий набір і він повинен використовуватися при програмуванні синхронізації потоків.

Як це було показано в лекціях, семафор – це змінна особливого типу, що може змінюватися потоками з позитивним або негативним приростом, але звертання до змінної у відповідний момент завжди атомарно навіть у багатопоточних програмах. Це означає, що якщо два потоки (або декілька їх) у програмі намагаються змінити значення семафора, система гарантує, що всі операції будуть насправді виконуватися по черзі одна за іншою. У випадку звичайних змінних результат конфліктних операцій різних потоків в одній програмі є абсолютно довільним.

Найчастіше семафори використовуються для захисту фрагмента програмного коду, так щоб тільки один потік виконання міг змінити його в будь-який конкретний момент часу. Для вирішення цього завдання досить щоб семафор приймав тільки значення 0 і 1 (***двійковий*** або ***бінарний семафор***). Більше узагальнений тип семафора – ***семафор з лічильником*** застосовується тоді коли потрібно обмеженому числу потоків дозволити виконувати заданий фрагмент коду. На практиці у програмуванні склалась ситуація, що семафори з лічильником

є менш популярними ніж бінарні.

Згідно зі стандартом POSIX, імена функцій для роботи з семафорами починаються із префікса sem_. Для роботи з потоками застосовують чотири базові функції.

sem_init

Семафор створюється за допомогою функції sem_init, що оголошується в такий спосіб.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Ця функція ініціалізує об'єкт-семафор, на який указує параметр-покажчик sem, задає варіант його спільного використання pshared і привласнює йому початкове цілочисельне значення value. Параметр pshared управляє типом семафора. Якщо pshared дорівнює 0, семафор є локальним відносно поточного процесу. У противному випадку семафор може бути спільно використаний різними процесами. Не всі версії ОС Linux підтримують спільне використання семафорів, і передача ненульового значення параметру pshared призводить до аварійного завершення виклику.

sem_wait, *sem_post*

Наступна пара функцій управляє значенням семафора й оголошується вони в такий спосіб.

```
#include <semaphore.h>
int sem_wait(sem_t* sem);
int sem_post(sem_t* sem);
```

Обидві ці функції приймають покажчик на об'єкт-семафор, котрий був створений викликом функції sem_init.

Функція sem_post атомарно збільшує значення семафора на 1 (відповідає дії підняття семафора – up).

Функція sem_wait атомарно зменшує значення семафора на одиницю (дія опускання – down), але завжди чекає доти, поки спочатку лічильник семафора не одержить ненульове значення. Таким чином, якщо викликається sem_wait для семафора зі значенням 2, то потік продовжить своє виконання, а семафор буде зменшений до 1. Якщо sem_wait викликається для семафора зі значенням 0, функція буде чекати доти, поки який-небудь інший потік не збільшить значення, і воно стане ненульовим. Якщо на закритому семафорі очікують два або більше потоків, то тільки один з них одержить можливість продовжитися при відкритті семафора, інші залишаться в стані очікування.

sem_destroy

Остання з базових функцій для роботи з семафорами – sem_destroy. Вона оголошується в такий спосіб:

```
#include <semaphore.h>
int sem_destroy(sem_t* sem);
```

і очищає семафор, коли закінчується робота з ним.

Якщо робиться спроба знищити семафор, на якому чекає хоча б один потік, то вертається помилка. У випадку ж успішного завершення, всі вище вказані функції, як завжди, повертають значення 0.

Приклад наведений нижче демонструє застосування семафорів для синхронізації потоків.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
```

```
void *thread_function(void *arg);
sem_t bin_sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
  int res;
  pthread_t a_thread;
  void *thread result;
  res = sem_init(&bin_sem, 0, 0);
  if (res != 0) {
    perror("Semaphore initialization failed");
    exit(EXIT_FAILURE);
  }
  res = pthread_create(&a_thread, NULL,
                       thread_function, NULL);
  if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  printf("Input some text. Enter 'end' to finish\n");
  while (strncmp("end", work_area, 3) != 0) {
    fgets(work_area, WORK_SIZE, stdin);
    sem_post(&bin_sem);
  }
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  sem_destroy(&bin_sem);
  exit(EXIT_SUCCESS);
}

void *thread function(void *arg) {
  sem_wait(&bin_sem);
  while(strncmp("end", work area, 3) != 0) {
    printf("You input %d characters\n",
           strlen(work_area)-1);
    sem_wait(&bin_sem);
  }
  pthread_exit(NULL);
}
```

У прикладі оголошується масив для спільного використання і семафор, котрий перед створенням нового потоку, одержує значення 0 (закритий).

У функції main, після запуску нового потоку, у робочу область, поки семафор закритий, читається деякий текст із клавіатури й потім за допомогою sem_post відкривається семафор. Це дозволяє другому потоку порахувати символи перед тим, як перший потік почне знову зчитувати ввід із клавіатури. Як видно з вихідного коду, потоки спільно використовують той самий масив work_area.

У прикладі пропущені деякі перевірки помилок, але в робочому програмному коді *обов'язково завжди перевіряти й виявляти помилки*, що повертаються із системних викликів.

У програмах подібних до розглянутої, через недотримання часових умов роботи декількох потоків можливо не передбачене нарощування семафора кілька разів. У цьому випадку один з потоків одержує можливість виконати свій код кілька разів, зменшуючи значення семафора поки воно не стане нульовим. Очевидно, що таке явище порушує логіку роботи програми, і повинне бути ліквідованим. Зробити це можна шляхом застосування додаткового семафора таким чином, щоб той потік, який багаторазово відкриває семафор, на другому семафорі очікував закінчення коду іншого процесу (приклад розглянутий у курсі лекцій).

Але в таких випадках набагато ефективніше застосування не семафорів, а м'ютексів.

М'ютекси дають можливість програмістам «замикати» поділюваний об'єкт так, що тільки один потік може звернутися до нього.

_mutex_init, *lock*, *unlock*, *destoy*

Базові функції, необхідні для використання м'ютексів, дуже схожі на функції семафорів. Вони оголошуються в такий спосіб:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t* mutex,
            const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread mutex_t* mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Ну і як завжди, у випадку успішного завершення функцій вертається 0 і код помилки у випадку аварійного завершення, але ці *функції не встановлюють змінну* errno і потрібно використовувати тільки код повернення.

Як і функції семафорів, функції м'ютексів приймають покажчик на попередньо створений об'єкт типу pthread_mutex_t. Додатковий параметр атрибутів у функції pthread_mutex_init дозволяє задати атрибути м'ютекса, що призначені для керування його поведінкою.

При деяких атрибутах поведінки (у тому числі й при атрибутах за замовчуванням) програма може потрапити в тупикову ситуацію. Це трапляється коли, наприклад, здійснюється спроба виклику функції pthread_mutex_lock для раніше вже заблокованого м'ютекса. Зміна атрибутів м'ютекса дозволяє уникнути такої ситуації й змушує м'ютекс або перевіряти наявність такої ситуації й повертати помилку або діяти рекурсивно, дозволяючи множинні блокування тим самим потоком, але, при цьому, у подальшому очікується така ж кількість розблокувань. Якщо зміни атрибутів м'ютекса не передбачається, то у другому параметрі потрібно передавати значення NULL, і використовувати поведінку за замовчуванням.

Нижче наведений приклад програми, що аналогічна попередньої, але використовує м'ютекс для забезпечення доступу до поділюваних змінних тільки одному потоку в кожній з моментів часу. Для простоти в програмі знову пропущені дії по перевірці помилок при поверненнях з заблокованого й відкритого м'ютекса. Але слід ще раз підкреслити, що в реальних програмах *таке спрощення не припустиме*.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
  int res;
  pthread_t a_thread;
  void *thread_result;
  res = pthread_mutex_init(&work_mutex, NULL);
  if (res != 0) {
    perror("Mutex initialization failed");
    exit(EXIT_FAILURE);
  }
  res = pthread_create(&a_thread, NULL,
                       thread_function, NULL);
  if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  pthread_mutex_lock(&work_mutex);
  printf("Input some text. Enter 'end' to finish\n");
  while (!time_to_exit) {
    fgets (work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
      pthread_mutex_lock(&work_mutex);
      if (work_area[0] != '\0') {
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
      } else {
        break;
      }
    }
  }
  pthread_mutex_unlock(&work_mutex);
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
```

```
  if (res ! = 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  pthread_mutex_destroy(&work_mutex);
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
  sleep(1);
  pthread_mutex_lock(&work_mutex);
  while(strncmp("end", work_area, 3) ! = 0) {
    printf("You input %d characters\n",
           strlen(work_area)-1);
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while (work_area[0] == '\0') {
      pthread_mutex_unlock(&work_mutex);
      sleep(1);
      pthread_mutex_lock(&work_mutex);
    }
  }
  time_to_exit = 1;
  work_area[0] = '\0';
  pthread_mutex_unlock(&work_mutex);
  pthread_exit(0);
}
```

У програмі, як і раніше, оголошується поділюваний масив робочої області, а також додаткова змінна time_to_exit. Тут же оголошується, а в потоці main створюється об'єкт м'ютекса, після чого запускається новий потік. Перший потік робить читання даних у буфер доти, поки другий потік не встановить прапор завершення роботи time_to_exit. Другий потік підраховує кількість уведених до буферу символів, після чого очищає його вміст. Потік працює, поки в буфері не з'являться символи «end» після цього встановлює прапор time_to_exit. Під час роботи обидва процеси поперемінно за допомогою м'ютекса захоплюють у виняткове користування спільно поділюваний буфер. При

цьому кожний з них намагається заблокувати м'ютекс і, коли це вдається, перевіряє, чи закінчив паралельно працюючий потік своє завдання. Якщо ні, м'ютекс відкривається, виконується очікування на якийсь час. Після установки прапора виконується приєднання другого потоку, видалення об'єкта м'ютекса й процес завершується.

4.1.1.3 Скасування потоку

Іноді логіка виконання програми вимагає, щоб якийсь потік завершувався на вимогу з іншого потоку. Для цього призначені кілька функцій.

pthread_cancel

Функція для створення запиту на завершення потоку.

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Функція досить проста. Їй на вході передається ідентифікатор потоку, що підлягає завершенню. Функція стандартним образом повідомляє про вдале виконання або про помилку.

Але потік, що одержав повідомлення про закриття, може по-різному реагувати на цей сигнал.

pthread_setcancelstate

При запуску потоку він може за допомогою спеціальної функції pthread_setcancelstate із прототипом:

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

установити власну реакцію на скасування. Параметр state у виклику функції може приймати два значення:

- PHTREAD_CANCEL_ENABLE – дозволяє потоку одержувати й реагувати на запити на скасування;
- PTHREAD_CANCEL_DISABLE – змушує ігнорувати запити на завершення.

Через покажчик oldstate функція повертає попередній стан. Якщо його значення не представляє інтересу, то можна передати параметр NULL.

Коли дозволена обробка запитів на скасування, то знову є дві альтернативні можливості обробки запиту.

pthread_setcanceltype

Після встановлення власної реакції потоку на скасування, тип реакції потоку тепер установлюється функцією pthread_setcanceltype і залежить від параметра type, переданого до неї. Цей параметр так само може приймати два значення:

- PTHREAD_CANCEL_ASYNCHRONOUS – запит на скасування обробляється негайно;
- PTHREAD_CANCEL_DEFERRED, скасування потоку відкладається доти, поки він не звернеться до якій-небудь із наступних функцій: pthread_join, pthread_cond_wait, pthread_cond_timedwait, pthread_testcancel, sem_wait, sigwait, у деяких системах можливе використання sleep.

Функція pthread_setcanceltype має наступний прототип:

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

На закінчення опису роботи з потоками в ОС Linux, наведений приклад створення декількох (у прикладі шістьох) потоків у тому самому процесі з наступним видаленням їх у послідовності, що відрізняється від порядку створення (видалення робиться у зворотному порядку).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
  int res;
  pthread_t a_thread[NUM_THREADS];
  void *thread_result;
  int lots_of_threads;
  for (lots_of_threads = 0; lots_of_threads <
       NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_thread]),
                         NULL, thread_function,
                         (void*)lots_of_threads);
    if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
    }
  }
  printf("Waiting for threads to finish...\n");
  for (lots_of_threads = NUM_THREADS - 1;
       lots_of_threads >= 0; lots_of_threads-i) {
    res = pthread_join(a_thread[lots_of_threads],
                       &thread_result);
    if (res == 0) {
      printf("Picked up a thread\n");
    } else {
      perror("pthread_join failed");
    }
  }
  printf("All done\n");
  exit(EXIT_SUCCESS);
}

void* thread_function(void* arg) {
  int my_number = (int)arg;
  int rand_num;
  printf("thread_function is running. Argument was %d\n",
         my_number);
  rand_num = 1+(int)(9.0*rand()/(RAND_MAX+1.0));
```

```
    sleep(rand_num);
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

4.1.2 Потоки в ОС Windows

Основною одиницею виконання коду в Windows є потік – незалежна одиниця виконання в контексті процесу, кілька потоків одночасно можуть виконуватися в рамках одного процесу, розділяючи його адресний простір і інші ресурси.

Потоки, що належать до одного процесу, розділяють загальні дані й код, тому дуже важливо, щоб кожний потік мав ще й власну область пам'яті, що належить тільки йому. В Windows задоволення цієї вимоги забезпечується декількома способами.

1) У кожного потоку є власний стек, що він використовує при виклику функцій і обробці деяких даних;
2) при створенні потоку вихідний процес може передати йому аргумент, що є просто покажчиком. На практиці цей аргумент поміщається в стек потоку;
3) Кожний потік може розподіляти індекси власних локальних областей зберігання (Thread Local Storage, TLS). Одним з переваг TLS є те, що вони забезпечують захист даних, котрими володіє один поток, від впливу з боку інших потоків.

Хоча в ОС Windows потоки, як і процеси, є зовсім рівноправними, але іноді для зручності застосовуються терміни «батьківський» і «дочірній» потік. Це виправдано тим, що при запуску процес Windows містить як мінімум один потік і всі інші протоки запускаються з нього.

4.1.2.1 Керування потоками

У потоків, як і у будь-яких інших об'єктів операційної системи Windows,

є ***дескриптори***, за допомогою яких система і управляє потоками.

CreateThread

Для створення потоків, що виконуються в адресному просторі вихідного процесу, передбачений системний виклик CreateThread, прототип якого має вигляд:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpvThreadParm, DWORD dwCreationFlags,
  LPDWORD lpThreadId)
```

Параметри функції мають наступний смисл:

- lpsa – покажчик на структуру атрибутів захисту (таку ж, як і у процесів), при використанні за замовчуванням можна передати NULL;
- dwStackSize – розмір стека нового потоку в байтах. Значенню 0 цього параметра відповідає розмір стека за замовчуванням – дорівнює розміру стека основного потоку (найчастіше це 1 МіБ). Спочатку для стека приділяється одна сторінка віртуальної пам'яті. Нові сторінки стека виділяються в міру потреби доти, поки стек не досягне свого максимального розміру;
- lpStartAddr – покажчик на функцію, що повинна виконуватися в контексті процесу. Ця функція приймає єдиний аргумент у вигляді покажчика й повертає 32-бітовий код завершення. Цей аргумент може інтерпретуватися потоком або як змінна типу DWORD, або як покажчик. Функція потоку – ThreadFunc має наступну сигнатуру:

```
DWORD WINAPI ThreadFunc(LPVOID)
```

- lpvThreadParm – покажчик, котрий переданий потоку як аргумент, частіш за все інтерпретується потоком як покажчик на власну, керовану потоком, захищену від інших потоків того ж процесу область пам'я-

ті, що розподіляється їм незалежно, аргумент має унікальну для кожного потоку структуру;

- dwCreationFlags – якщо значення цього параметра встановлено таким, що дорівнює 0, то потік запускається відразу ж після виклику функції CreateThread. Установка цього прапорця у спеціальне значення CREATE_SUSPENDED призведе до запуску потоку в припиненому стані, з якого потік може бути переведений у стан готовності шляхом виклику функції ResumeThread;
- lpThreadId – покажчик на змінну типу DWORD, що одержує ідентифікатор нового потоку.

У випадку помилки функція повертає значення NULL.

ExitThread

Будь-який потік процесу може сам завершити своє виконання, викликавши функцію ExitThread з наступним прототипом:

```
VOID ExitThread(DWORD dwExitCode)
```

По завершенні виконання потоку пам'ять, котра займалася його стеком, звільняється. Коли завершується виконання останнього потоку, завершується й виконання самого процесу.

TerminateThread

Виконання потоку також може бути завершено іншим потоком за допомогою функції TerminateThread, однак звільнення ресурсів потоку при цьому не відбувається, не виконуються оброблювачі завершення, тому застосовувати функцію TerminateThread є *украй небажаним*.

Потік, виконання якого було завершено, продовжує існувати доти, поки за допомогою функції CloseHandle не буде закритий його дескриптор.

GetExitCodeThread

За допомогою функції:

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode)
```

будь-який потік, можливо й такий, котрий очікує завершення іншого потоку, може одержати код його завершення, lpExitCode містить код завершення потоку, що вказує на його стан. Якщо потік ще не завершений, значення цієї змінної буде дорівнювати STILL_ACTIVE.

Для ідентифікації потоків існує цілий ряд функцій, що дозволяють одержати ідентифікатори (ID) і дескриптори потоків, наприклад, такі:

- GetCurrentThread – повертає ненаслідуваний псевдодескриптор вихідного потоку;
- GetCurrentThreadId – дозволяє одержати ідентифікатор потоку, а не його дескриптор;
- OpenThread – створює дескриптор потоку по відомому ідентифікатору.

ResumeThread, *SuspendThread*

При створенні потоку, з ним зв'язується так званий ***лічильник припинень***. Коли потік створюється в припиненому стані, лічильнику привласнюється значення 1.

API підтримує збільшення й зменшення лічильника за допомогою функцій припинення й поновлення виконання потоку:

```
DWORD ResumeThread(HANDLE hThread);
DWORD SuspendThread(HANDLE hThread);
```

Перша з них зменшує на одиницю значення лічильника, прагнучи відновити виконання потоку, друга збільшує лічильник на одиницю й припиняє по-

тік. Виконання потоку може бути продовжено лише в тому випадку, якщо значення лічильника стане рівним 0 (у лекціях розглядався пример роботи потоків з використанням подібних функції – resume і suspend).

У випадку успішного виконання обидві функції повертають попереднє значення лічильника припинень, при помилці вертається – 0xffffffff.

WaitForSingleObject, *WaitForMultipleObjects*

Очікування завершення потоку здійснюється абсолютно так само, як і для процесів, але при виклику функцій очікування (WaitForSingleObject і WaitForMultipleObjects) замість дескрипторів процесів у функції передаються дескриптори потоків. Оскільки ці функції універсальні для всіх об'єктів ядра ОС, то в масиві, що передається до функції WaitForMultipleObjects, можуть бути одночасно зазначені дескриптори потоків, процесів і інших об'єктів.

Функція очікування чекає, поки об'єкт, зазначений дескриптором, не перейде в сигнальний стан. Об'єкт потоку переводиться в сигнальний стан за допомогою функцій ExitThread і TerminateThread, що призводить до звільнення всіх інших потоків, що чекають переходу даного об'єкта в сигнальний стан. Дескриптор потоку, що перейшов у сигнальний стан, не виходить із цього стану.

Чекати переходу в сигнальний стан того самого об'єкта можуть одночасно кілька потоків.

Потоки в Windows синхронізуються за допомогою таких об'єктів ядра як семафори, м'ютекси й події, а також критичні секції, останні створюються не у ядрі, а у потоці користувальницької програми.

Об'єкти синхронізації, можуть застосовуватися для синхронізації потоків, що належать як тому самому процесу, так і, крім критичних секцій, до різних процесів. Синхронізація потоків є однією з найважливіших і найцікавіших задач і відіграє істотну роль майже в будь-якому багатопоточному додатку.

4.1.2.2 Застосування критичних ділянок коду

При використанні поділюваної загальної пам'яті несинхронізованими потоками основна проблема полягає в тому, що в програмі є ***критична ділянка коду*** (critical section), що характеризується тим, що ця ділянка програмного коду, щораз повинна виконуватися тільки одним потоком. Паралельне виконання цієї ділянки декількома потоками може призвести до непередбачених або невірних результатів. Таким чином, якщо один з потоків приступив до виконання критичної секції, то ніякий інший потік не повинен входити в даний код доти, поки його не покине перший потік.

Як простий механізм реалізації й застосування на практиці концепції критичних ділянок коду Windows надає об'єкт CRITICAL_SECTION.

Об'єкти CRITICAL_SECTION (CS) можна ініціюовати і видаляти, але вони не мають дескрипторів (оскільки не є об'єктами ядра) і не можуть спільно використовуватися іншими процесами. Поділювані змінні, з якими працює критичний код, повинні оголошуватися як змінні типу CRITICAL_SECTION. Потоки входять в об'єкти CS і залишають їх, але виконання коду окремого об'єкта CS щораз дозволено тільки одному потоку. Разом з тим, той самий потік може входити в кілька окремих об'єктів CS і залишати їх, якщо вони розташовані в різних місцях програми.

InitializeCriticalSection, *DeleteCriticalSection*

Для ініціалізації й видалення змінної типу CRITICAL_SECTION використовуються, відповідно, дві спеціальні функції:

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION
                               lpCriticalSection)
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

EnterCriticalSection, *LeaveCriticalSection*

Дві інші спеціальні функції:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

дозволяють заблокувати потік, якщо на даній критичній ділянці коду присутній інший потік, і зняти блокування потоку, що очікує. Завжди необхідно стежити за своєчасною переуступкою прав володіння об'єктами CS, у противному випадку інші потоки будуть перебувати в стані очікування протягом невизначеного часу навіть після завершення виконання потоку-власника поділюваного ресурсу.

Об'єкти CRITICAL_SECTION є рекурсивними, тобто потік, що володіє об'єктом CS, може повторно ввійти в цей же CS без його блокування. Підтримується лічильник входжень в об'єкт CS, і тому, щоб розблокувати цей об'єкт для інших потоків потік повинен покинути даний CS стільки разів, скільки було входжень у нього. Це корисно для реалізації рекурсивних функцій і безпечного багатопоточного виконання функцій загальних (поділюваних) бібліотек.

Вихід з об'єкта CS, яким даний потік не володіє, може привести до непередбачених результатів, включаючи блокування самого потоку.

TryEnterCriticalSection

Ще одна функція:

```
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

дозволяє опитати CS і перевірити, чи не володіє їм інший потік.

Повернення TRUE означає, що визивний потік, котрий викликав функцію, може набути права володіння критичною ділянкою, а повернення FALSE говорить про те, що критична ділянка коду вже зайнята іншим потоком.

Об'єкти CRITICAL_SECTION не є об'єктами ядра й підтримуються в користувальницькому просторі. Це може привести до збільшення продуктивності.

Використання об'єктів CRITICAL_SECTION є досить простим прийо-

мом, і одним з найпоширеніших способів їхнього використання є забезпечення доступу потоків до поділюваних глобальних змінних.

Нижче наведений приклад системи «виробник/споживач». Вимоги до програми і її дії можна визначити в такий спосіб.

- У програмі повинні бути два потоки, виробник (producer) і споживач (consumer), що працюють у повністю асинхронному режимі.
- Виробник періодично створює повідомлення, що містять таблицю чисел, наприклад, таблицю біржових котирувань, що періодично оновлюється.
- На вимогу користувача споживач відображає поточні дані. Необхідно, щоб відображувані дані являли собою самий останній повний набір і ніякі дані не повинні відображатися двічі.
- Дані не повинні відображатися в ті проміжки часу, коли вони обновляються виробником; застарілі дані також не повинні відображатися. Використання об'єктів CS гарантує, що ніколи не відбудеться одержання даних у момент відновлення таблиці. Цей приклад є окремим випадком конвеєрної моделі, у якій дані передаються від одного потоку до іншого.
- Як засіб контролю цілісності даних виробник обчислює просту контрольну суму, що далі порівнюється з аналогічною сумою, котра обчислена споживачем,
- Обома потоками підтримується статистика сумарної кількості відправлених, отриманих і загублених повідомлень.

```
#include "EvryThing.h"
#include <time.h>
#define DATA_SIZE 256
//Блок повідомлення.
typedef struct msg_block_tag {
// Прапори готовності й припинення повідомлень.
volatile DWORD f_ready, f_stop;
```

```
// Порядковий номер блоку повідомлення.
volatile DWORD sequence;
volatile DWORD nCons, nLost;
time_t timestamp;
// Структура захисту блоку повідомлення.
CRITICAL_SECTION mguard;
// Контрольна сума вмісту повідомлення.
DWORD checksum;
// Уміст повідомлення.
DWORD data[DATA_SIZE];
} MSG_BLOCK;
// Блок, підготовлений до заповнення новим повідомленням.
MSG_BLOCK mblock = {0, 0, 0, 0, 0};
DWORD WINAPI produce(void*);
DWORD WINAPI consume(void*);
void MessageFill(MSG_BLOCK*);
void MessageDisplay(MSG_BLOCK*);

DWORD _tmain(DWORD argc, LPTSTR argv[]) {
  DWORD Status, ThId;
  HANDLE produce_h, consume_h;
// Ініціалізація критичної ділянки блоку повідомлення.
InitializeCriticalSection (&mblock.mguard);
// Створення двох потоків.
  produce_h = (HANDLE)_beginthreadex(NULL, 0,
                          produce, NULL, 0, &ThId);
  consume_h = (HANDLE)_beginthreadex (NULL, 0,
                          consume, NULL, 0, &ThId);
// Очікування завершення обох потоків.
  WaitForSingleObject(consume_h, INFINITE);
  WaitForSingleObject(produce_h, INFINITE);
  DeleteCriticalSection(&mblock.mguard);
  _tprintf(_T("Потоки завершили виконання\n"));
  _tprintf(_T("Відправлено: %d, Отримано: %d, \
              Відомі втрати: %d\n"),
              mblock.sequence, mblock.nCons,
              mblock.nLost);
  return 0;
}

/* Потік виробника - створення нових повідомлень через
   випадкові інтервали часу. */
DWORD WINAPI produce(void *arg){
  srand((DWORD)time(NULL));
  while (!mblock.f_stop) {
```

```
    Sleep(rand() / 100);
// Одержати й заповнити буфер. */
    EnterCriticalSection(&mblock.mguard);
    __try {
      if (!mblock.f_stop) {
        mblock.f_ready = 0;
        MessageFill(&mblock);
        mblock.f_ready = 1;
        mblock.sequence++;
      }
    } __finally {
        LeaveCriticalSection (&mblock.mguard); }
  }
  return 0;
}

/* Потік споживача - одержання повідомлення по запиту
   користувача. */
DWORD WINAPI consume (void *arg) {
  DWORD ShutDown = 0;
  CHAR command, extra;
// Прийняти ЧЕРГОВЕ повідомлення
  while (!ShutDown) {
    _tprintf(_T("\n**Уведіть 'c' для прийому; 's' \
                для припинення роботи: "));
    _tscanf("%c%c", &command, &extra);
    if (command == 's') {
      EnterCriticalSection(&mblock.mguard);
      ShutDown = mblock.f_stop = 1;
      LeaveCriticalSection(&mblock.mguard);
    } else /* Одержати новий буфер для прийнятих
              повідомлень. */
    if (command == 'c'){
      EnterCriticalSection(&mblock.mguard);
      __try {
      if (mblock.f_ready == 0)
       _tprintf(_T("Нові повідомлення відсутні. \
                    Повторіть спробу.\n"));
      else {
        MessageDisplay(&mblock);
        mblock.nCons++;
        mblock.nLost = mblock.sequence -
                                        mblock.nCons;
// Нові повідомлення відсутні.
        mblock.f_ready = 0;
```

```
      }
      } __finally {
         LeaveCriticalSection (&mblock.mguard); }
    } else {
        tprintf(_T("Така команда відсутня. /
                    Повторіть спробу.\n"));
      }
  }
  return 0;
}
// Заповнення буфера повідомлення вмістом
void MessageFill(MSG_BLOCK *mblock) {
  DWORD i;
  mblock->checksum = 0;
  for (i = 0; i < DATA_SIZE; i++) {
    mblock->data[i] = rand();
    mblock->checksum ^= mblock->data[i];
  }
  mblock->timestamp = time(NULL);
  return;
}

/* Відображення буфера повідомлення
void MessageDisplay(MSG_BLOCK *mblock) {
  DWORD i, tcheck = 0;
  for (i = 0; i < DATA_SIZE; i++)
    tcheck ^= mblock->data[i];
  _tprintf(_T("\nЧас генерації повідомлення № %d: %s"),
              mblock->sequence,
              _tctime(&(mblock->timestamp)));
  _tprintf(_T("Перший і останній записи: %x %x\n"),
              mblock->data[0],
              mblock->data[DATA_SIZE - 1]);
  if (tcheck == mblock->checksum)
    _tprintf(_T("УСПІШНА ОБРОБКА ->Контрольна сума \
                 збігається.\n"));
  else
   _tprintf(_T("ЗБІЙ ->Розбіжність контрольної суми \
                Повідомлення зазіпсоване.\n"));
  return;
}
```

4.1.2.3 Робота з м'ютексами

Об'єкти взаємного виключення – м'ютекси забезпечують більш універса-

льну функціональність у порівнянні з CRITICAL_SECTION. Оскільки м'ютекси можуть мати імена й дескриптори, їх можна використовувати також для синхронізації потоків, що належать різним процесам. Наприклад, два процеси, що поділяють загальну пам'ять за допомогою відображення файлів, можуть використовувати м'ютекси для синхронізації доступу до поділюваних областей пам'яті.

М'ютекси можуть спільно використовуватися різними процесами, оскільки вони мають дескриптор, що може бути переданий в інший процес. Крім того м'ютекси допускають кінцеві періоди очікування. Звертання до м'ютекса й робота з ним, як і з будь-яким іншим об'єктом ОС, що має дескриптор, провадиться за допомогою звертання до цього дескриптора. Так потік здобуває права володіння м'ютексом шляхом виклику однієї з функцій очікування WaitForSingleObject або WaitForMultipleObjects, передачею їм як параметр дескриптора цього м'текса (ці функції використовуються з усіма об'єктами, тому в масиві дескрипторів переданому функції WaitForMultipleObjects можуть утримуватися дескриптори зовсім різних об'єктів). Звільнення м'ютекса й передача прав користування одному з потоків, що очікують, здійснюється за допомогою виклику функції ReleaseMutex з дескриптором м'ютекса.

CreateMutex

Дескриптор (handle) об'єкта м'ютекс створюється функцією

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa,
  BOOL bInitialOwner, LPCTSTR lpMutexName)
```

Як видно із прототипу функції, їй передаються три параметри.

Перший параметр – lpsa є покажчиком на структуру, що містить атрибути м'ютекса. Якщо використовуються атрибути за замовчуванням, то цей параметр установлюється рівним NULL.

Другий параметр – bInitialOwner визначає можливість (значення TRUE) для потоку, що викликав функцію створення м'ютекса негайно дістати права

володіння новим м'ютексом. Ця атомарна операція дозволяє запобігти придбанню прав володіння м'ютексом іншими потоками, перш ніж це зробить потік, що створює м'ютекс.

Останній третій параметр – lpMutexName указує на рядок, що містить ім'я м'ютекса. Імена м'тексов чутливі до регістра, мають довжину до 260 символів і повинні бути унікальними у своєму просторі імен (події, м'ютексы, семафори, відображення файлів і інші об'єкти ядра). Якщо параметр дорівнює NULL, то м'ютекс створюється без ім'я.

Значення, що повертається функцією, має тип HANDLE, і значення NULL указує на невдале завершення функції.

ReleaseMutex

Прототип функції звільнення м'ютекса має такий вигляд:

```
BOOL ReleaseMutex(HANDLE hMutex),
```

і особливих пояснення не вимагає

Потік може заволодівати тим самим м'ютексом кілька разів, і при цьому не буде блокуватися навіть у тих випадках, коли вже володіє їм. У цьому випадку потік повинен звільнити м'ютекс стільки разів, скільки він його захоплював. Це, так само як і у випадку об'єктів CS, може виявитися корисним для обмеження доступу до рекурсивних функцій, а також у додатках, що реалізують вкладені транзакції.

При роботі з м'ютексами потоки повинні програмуватися таким чином, щоб ресурси завжди звільнялися, перш ніж потік завершить своє виконання.

Нижче наведений приклад використання м'ютексів для синхронізації двох потоків, які працюють спільно, заповнюючи поділюваний масив значеннями по зростанню. Якщо в цьому випадку не передбачити засобів синхронізації потоків, то масив може бути заповнений не по зростанню значень, а довільним чином.

У програмі спочатку оголошуються глобальні поділювані змінні (сам масив з 1000 чисел, індекс поточної позиції в масиві й сам дескриптор м'ютекса). У функції WinMain створюється об'єкт м'ютекса й два робочих потоки, після чого програма переходить в очікування завершення обох потоків, і, по їхньому закінченні, закриває дескриптори як потоків, так і м'ьютекса.

Функції потоків на початку своєї роботи виконують перевірку на завершення роботи і якщо робота не завершена, очікують можливості захопити м'ютекс. Після одержання прав монопольного користування поділюваним масивом і поточним покажчиком позиції елемента масиву, змінюють їх відповідним чином. По закінченні роботи з ресурсом м'ютекс звільняється.

```
const int MAX_TIMES=1000;
//Поділювані змінні
int g_nIndex=0;
DWORD g_dwTimes [MAX_TIMES];
HANDLE g_hMutex=NULL;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
                   hPrevInstance, LPSTR lpCmdLine,
                   int nCmdShow)
  {
     HANDLE hThreads[2];
//Створюється об'єкт mutex
    g_hMutex=CreateMutex (NULL, FALSE, NULL);
/* Запускаються два потоки з параметрами за
   замовчуванням */
    hThreads[0]=CreateThread(NULL, 0, FirstThread,
                             NULL, 0, NULL);
    hThreads[1]=CreateThread(NULL, 0, SecondThread,
                             NULL, 0, NULL);
//Очікування завершення обох потоків
    WaitForMultipleObjects (2, hThreads, TRUE,
                            INFINITE);
//Закриваються дескриптори потоків
    CloseHandle(hThreads[0]);
    CloseHandle(hThreads[1]);
// Закривається об'єкт mutex
    CloseHandle (g_hMutex);
  }
```

```
DWORD WINAPI FirstThread(LPVOID lpvThreadParm)
  {
    BOOL fDone=FALSE;
    DWORD dw;
    while (!fDone)
      {
//Очікування звільнення об'єкта mutex
        dw=WaitForSingleObject(g_hMutex, INFINITE);
        if(dw==WAIT_OBJECT_0)
          {
//Об'єкт mutex звільнився
            if(g_nIndex>=MAX_TIMES) fDone=TRUE;
            else
              {
                g_dwTimes[g_nIndex]=GetTickCount();
                g_nIndex++;
              }
//Звільняється mutex
            ReleaseMutex(g_hMutex);
          }
        else break; //Вихід із циклу
      }
    return (0);
  }

DWORD WINAPI SecondThread (LPVOID lpvThreadParm)
  {
    BOOL fDone=FALSE;
    DWORD dw;
    while (!fDone)
      {
//Очікування звільнення об'єкта mutex
        dw=WaitForSingleObject(g_hMutex, INFINITE);
        if(dw==WAIT_OBJECTEC_0)
          {
//Об'єкт mutex звільнився
            if(g_nIndex>=MAX_TIMES) fDone=TRUE;
            else
              {
                g_nIndex++;
                g_dwTimes[g_nIndex-1]=GetTickCount();
              }
//Звільняється mutex
            ReleaseMutex (g_hMutex);
          }
```

```
        else break; //Вихід із циклу
      }
    return(0);
  }
```

4.1.2.4 Семафори та їх застосування

Об'єкти синхронізації ядра – семафори, підтримують лічильники, і коли значення цього лічильника більше 0, об'єкт семафора перебуває в сигнальному стані. Якщо ж значення лічильника стає нульовим, об'єкт семафора переходить у несигнальний стан.

Потоки й процеси організують очікування на семафорі звичайним способом, використовуючи для цього одну або кілька функцій з сімейства Wait. Коли потік розблоковується значення лічильника зменшується на 1.

CreateSemaphore, *ReleaseSemaphore*

Для керування семафорами використовуються функції аналогічні тим, які призначені для керування м'ютексами:

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpsa,
  LONG lSemInitial, LONG lSemMax, LPCTSTR lpSemName);
BOOL ReleaseSemaphore(HANDLE hSemaphore,
  LONG cReleaseCount, LPLONG lpPreviousCount);
```

Параметр lSemMax функції CreateSemaphore обумовлює максимально припустиме значення лічильника семафора, і його значення повинне бути рівним, принаймні, 1. Параметр lSemInitial – початкове значення цього лічильника, його завдання обов'язкове і воно завжди повинне задовольняти наступній умові: $0 \le lSemInitial \le lSemMax$ і ніколи не виходити за межі діапазону. Параметри lpsa та lpSemName такі ж самі, як і в м'ютексів. Повернення функцією значення NULL,як завжди вказує на її невдале виконання.

Кожна окрема операція очікування може зменшити значення лічильника тільки на 1, але за допомогою функції ReleaseSemaphore значення його лічи-

льника може бути змінене на будь-яке значення (мова йде про другий параметр – cReleaseCount), таке, що більше 0 і може сягати аж максимально припустимого значення.

За допомогою третього параметру – покажчика lpPreviousCount функція повертає попереднє значення лічильника. Це значення буває потрібним у виняткових випадках, тому у звичайній практиці вказується NULL. Функція повертає FALSE, якщо значення лічильника семафора перевищить lSemMax, при цьому лічильник семафора не змінюється.

Іноді м'ютекс розглядають як окремий випадок семафора, значення лічильника якого задано рівним 1. Але такий розгляд не є повністю вірним, тому що відсутнє таке поняття, як права володіння семафором і семафор, на відміну від м'ютекса, може бути звільнений будь-яким потоком, а не тільки тим, що очікує на ньому.

Найбільше часто семафори застосовуються у випадку керування розподілом кінцевого числа ресурсів, коли значення лічильника семафора асоціюється з кількістю доступних ресурсів, наприклад, кількістю об'єктів у черзі. Тоді максимальне значення лічильника відповідає максимальному розміру черги.

Семафори мають низку переваг, але їм властиві й деякі недоліки. Більш за те, семафори в певному розумінні є зайвими елементами у системі синхронізації, оскільки спільне використання м'ютексів і таких елементів синхронізації як події надає набагато більш широкі можливості керування, чим семафори.

4.1.2.5 Синхронізація потоків з допомогою подій

Останнім з об'єктів синхронізації ядра є ***події*** (events). Об'єкти події використовуються для того, щоб сигналізувати іншим потокам про настання якої-небудь події, наприклад, про появу нового повідомлення.

Важливою додатковою можливістю, яка забезпечується об'єктами подій, є те, що перехід у сигнальний стан єдиного об'єкта події здатний вивести зі стану очікування одночасно кілька потоків.

CreateEvent

Об'єкти події діляться на такі, що скидаються вручну й такі, що скидаються автоматично, і ця їхня властивість встановлюється при створенні події викликом функції CreateEvent.

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpsa,
   BOOL bManualReset, BOOL bInitialState,
   LPTCSTR lpEventName)
```

- Події, що скидаються вручну, (manual-reset events) можуть сигналізувати одночасно всім потокам, що очікують настання цієї події, і переводяться в несигнальний стан програмно.
- Події, котрі скидаються автоматично (auto-reset event) скидаються самостійно після звільнення одного з потоків, що очікують на їх появу, тоді як інші потоки продовжують очікувати переходу події в сигнальний стан.

Щоб створити подію, що скидається вручну, необхідно встановити значення параметра bManualReset у значення TRUE. Точно так само, щоб зробити початковий стан події сигнальним, таким, що дорівнює TRUE встановлюється значення параметра bInitialState.

SetEvent, *ResetEvent*, *PulseEvent*

Для керування об'єктами подій використовуються наступні три функції:

```
BOOL SetEvent(HANDLE hEvent)
BOOL ResetEvent(HANDLE hEvent)
BOOL PulseEvent(HANDLE hEvent)
```

Потік може встановити подію в сигнальний стан, використовуючи функцію SetEvent. Якщо подія є такою, що скидається автоматично, то вона вертається в несигнальний стан уже після звільнення тільки одного з потоків, котрі

очікують на неї. Під час відсутності потоків, котрі очікують настання цієї події, вона залишається в сигнальному стані доти, поки такий потік не з'явиться, після чого цей потік відразу ж звільняється. Аналогічним чином поводиться й семафор, максимальне значення лічильника якого було встановлене рівним 1.

Якщо ж подія така, що скидається вручну, то вона залишається в сигнальному стані доти, поки який-небудь потік не викличе функцію ResetEvent, указавши дескриптор цієї події як аргумент. У цей час всі потоки, що очікують, звільняються, але до виконання такого скидання події, інші потоки можуть як переходити в стан її очікування, так і звільнятися.

Функція PulseEvent звільняє всі потоки, що очікують настання події, котра скидається вручну, але після цього подія відразу ж скидається автоматично. У випадку ж використання події, що скидається автоматично, функція PulseEvent звільняє тільки один потік, що очікує, якщо такі є.

Документи Microsoft рекомендують уникати використання функції PulseEvent. Така рекомендація, очевидно, пов'язана з тим фактом, що функція PulseEvent стає корисної лише після того, подія, яка скидається вручну, установлена в сигнальний стан за допомогою функції SetEvent. Крім того, необхідно дотримуватися особливої обережності коли для очікування переходу в сигнальний стан всіх подій використовується функція WaitForMultipleObjects. У цьому випадку потік, що очікує, звільниться тільки тоді, коли одночасно всі події будуть перебувати в сигнальному стані, але деякі з подій, що перебувають у сигнальному стані, можуть бути скинуті, перш ніж потік звільниться.

Нижче наведена модернізація програми «виробник/споживач», що була розглянута для випадку синхронізації за допомогою критичних секцій. У цьому новому варіанті програми важливим є саме використання подій, оскільки усувається одна із проблем, пов'язана з тим, що в старому варіанті програми споживач повинен був безупинно повторювати спроби одержання нових повідомлень. Використання в прикладі м'ютексів замість критичних секцій не є принциповою відмінністю, тому що м'ютекс активізує критична ділянка коду для

доступу до об'єкта структури даних, а от подія використовується для повідомлення про те, що з'явилося нове повідомлення.

```
/* Підтримуються два потоки - виробник і споживач. Виробник пе-
  ріодично створює буферні дані з контрольними сумами, або "бло-
  ки повідомлень", що сигналізують споживачеві про готовність
  повідомлення. Потік споживача відображає інформацію у відпо-
  відь на запит користувача. */

#include "EvryThing.h"
#include <time.h>
#define DATA_SIZE 256
/* Блок повідомлення. */
typedef struct msg_block_tag {
/* Прапори готовності й припинення повідомлень. */
  volatile DWORD f_ready, f_stop;
/* Порядковий номер блоку повідомлення. */
  volatile DWORD sequence;
  volatile DWORD nCons, nLost; time_t timestamp;
/* М'ютекс, що захищає структуру блоку повідомлення. */
  HANDLE mguard;
/* Подія "Повідомлення готове". */
  HANDLE mready;
/* Контрольна сума повідомлення. */
  DWORD checksum;
/* Уміст повідомлення. */
  DWORD data[DATA_SIZE];
} MSG_BLOCK;

DWORD _tmain(DWORD argc, LPTSTR argv[]) {
  DWORD Status, ThId;
  HANDLE produce_h, consume_h;
/* Инициализировать м'ютекс і подію (скидається
   автоматично) у блоці повідомлення. */
  mblock.mguard = CreateMutex(NULL, FALSE, NULL);
  mblock.mready = CreateEvent(NULL, FALSE, FALSE,
                              NULL);
/* Створити потоки виробника й споживача; очікувати
   їхнього завершення.*/
/* ... Код як у попередній програмі ... */
  CloseHandle(mblock.mguard);
  CloseHandle(mblock.mready);
  _tprintf(_T("Потоки виробника й споживача \
              завершили виконання\n"));
```

```
  _tprintf(_T("Відправлено: %d, Отримано: %d, \
              Відомі втрати: %d\n"),
           mblock.sequence, mblock.nCons,
           mblock.nLost);
  return 0;
}
DWORD WINAPI produce(void *arg) {
/* Потік виробника - створення нових повідомлень через
   випадкові інтервали часу. */
  srand((DWORD)time(NULL)); /* Створити початкове
                               число для генератора
                               випадкових чисел. */
  while(!mblock.f_stop) {
/* Випадкова затримка. */
    Sleep(rand() / 10); /* Тривалий період очікування
                           наступного повідомлення. */
/* Одержати й заповнити буфер. */
    WaitForSingleObject(mblock.mguard, INFINITE);
    __try {
      if (!mblock.f_stop) {
        mblock.f_ready = 0;
        MessageFill(&mblock);
        mblock.f_ready = 1;
        mblock.sequence++;
        SetEvent(mblock.mready); /* Сигнал
                                    "Повідомлення готове". */
      }
    }
    __finally { ReleaseMutex(mblock.mguard); }
  }
  return 0;
}
DWORD WINAPI consume (void *arg) {
  DWORD ShutDown = 0;
  CHAR command, extra;
/* Прийняти ЧЕРГОВЕ повідомлення по запиту
   користувача. */
  while (!ShutDown) { /* Єдиний потік, що одержує
                         доступ до стандартних
                         пристроїв вводу/виводу. */
    _tprintf(_T("\n** Уведіть 'c' для прийому; \
                's' для припинення роботи: "));
    _tscanf("%c%c", &command, &extra);
    if (command == 's') {
      WaitForSingleObject(mblock.mguard, INFINITE);
```

```
      ShutDown = mblock.f_stop = 1;
      ReleaseMutex(mblock.mguard);
    } else if (command == 'c') {
/* Одержати новий буфер прийнятих повідомлень. */
       WaitForSingleObject(mblock.mready, INFINITE);
       WaitForSingleObject(mblock.mguard, INFINITE);
       __try {
         if (!mblock.f_ready) _leave;
/* Очікувати настання події, що вказує на
   готовність повідомлення. */
         MessageDisplay(&mblock);
         mblock.nCons++;
         mblock.nLost = mblock.sequence -
                        mblock.nCons;
         mblock.f_ready = 0; /* Нові готові
                             повідомлення відсутні. */
       } __finally { ReleaseMutex (mblock.mguard); }
       } else {
         _tprintf(_T("Неприпустима команда. \
                     Повторить спробу.\n"));
    }
  }
  return 0;
}
```

4.2 Завдання на самостійну роботу

При підготовці до виконання лабораторної роботи не обходимо по основній і додатковій літературі вивчити й знати механізми створення, управління, синхронізації, та припинення роботи потоків у операційних системах Linux та Windows. За результатами вивчення цих питань підготувати розгорнуті відповіді на контрольні запитання.

Підготувати ввідну частину протоколу лабораторної роботи згідно до варіанта завдання, одержаного від викладача.

4.3 Варіанти завдань до лабораторної роботи

В програмах для початкового заповнення масивів випадкових цілих чисел *необхідно використовувати* файл int.bin, котрий був створений в одному з ка-

талогів згідно до варіанта лабораторної роботи №2 і містить у себе 1000 випадкових цілих чисел.

Засоби синхронізації потоків обираються студентом самостійно відповідно до семантики варіанта завдання.

1) Три паралельних потоки обробляють цілочисельний масив M[100] (спочатку масив містить нулі). Перший потік пише в масив на місце нулів позитивні числа. Другий потік пише в масив негативні числа. Третій потік читає з масиву відмінні від нуля числа. Причому, читати інформацію з масиву він може лише тоді, коли хоча б один потік записав в масив число. Для контролю роботи кожний раз при читанні інформації третій потік виводить весь масив на екран. В кожний момент часу лише один потік може працювати з масивом.

2) Два паралельних потоки P1 і P2 обробляють масив M[100] натуральних випадкових чисел. P1 вилучає з масиву поруч розташовані парні числа, P2 – поруч розташовані непарні числа. Кожний раз при вилученні потоки друкують масив і елементи, що вилучаються.

3) Два паралельних потоки P1 і P2 обробляють масив натуральних випадкових чисел M[100]. P1 вилучає з масиву найбільші числа, P2 – найменші. Кожний раз при вилученні потоки друкують масиві і елементи, що вилучаються.

4) Три паралельних потоки P1, P2 і P3 обробляють три цілочисельних масиви випадкових чисел M1[100], M2[100], M3[100]. P1 вилучає з масивів M1 і M3 найменші числа. P2 вилучає з масивів M1 і M2 найбільші числа. P3 вилучає з масивів M2 і M3 число, що зустрічається в обох масивах. Кожний раз при вилученні потоки друкують масив і елементи, що вилучаються.

5) Три паралельних потоки P1, P2 і P3 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 пише на місце нулів підряд натуральні числа від 1 до 100 по одному за кожний сеанс, P2 пише на місце нулів підряд негативні цілі числа від -100 до -1 по одному за кожний сеанс, P3 читає з масиву числа, замінюючи їх нулями, якщо масив заповнений.

6) Чотири паралельних потоки P1, P2, P3 і P4 обробляють чотири цілочисельних масиви M1[100], M2[100], M3[100], M4[100], що містять спочатку

нулі. P1 вставляє в масиви M4 і M1 натуральні парні числа від 1 до 100 по одному за кожний сеанс. P2 вставляє в масиви M1 і M2 натуральні непарні числа від 1 до 100 по одному за кожний сеанс. P3 вставляє в масиви M2 і M3 негативні парні числа від -1 до -100 по одному за кожний сеанс. P4 вставляє в масиви M3 і M4 негативні непарні числа від -1 до -100 по одному за кожний сеанс. Потоки закінчують роботу, коли в масиві немає нульових елементів.

7) Два паралельних потоки P1 і P2 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 записує в масив числа від 1 до 100 (до 5 чисел за сеанс), якщо в масиві є нулі. P2 читає з масиву числа, замінюючи їх нулями (читати можна, якщо в масиві немає нулів).

8) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю випадкових чисел M[20,20]. P1 замінює нулями стовпчик з максимальним елементом. P2 замінює нулями рядок з мінімальним елементом.

9) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю випадкових чисел M[20,20]. P1 замінює нулями всіх сусідів максимального елементу. P2 замінює нулями всіх сусідів мінімального елементу.

10) Три паралельних потоки P1, P2 і P3 обробляють цілочисельний масив M[100] випадкових чисел. Потік P1 копіює до свого власного масиву парні числа по три числа за один раз. Потік P2 теж саме робить з числами котрі кратні трьом. Останній потік після кожної операції виводить на термінал всі три масиви.

11) Два паралельних потоки P1 і P2 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 пише на місце нулів підряд натуральні числа від 1 до 100 по одному за кожний сеанс, P2 замінює ці числа їхнім квадратом по одному за кожний сеанс. Після операції кожен потік друкує свій ідентифікатор і зміненій їм масив.

12) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю M[20,20], котра містить нулі. Спочатку обидва потоки по черзі, по п'ять чисел за один сеанс, замінюють нулі випадковими числами. По заповненню матриці, потік P1 замінює своїм ідентифікатором стовпчик з максимальною кількістю

парних елементів, Р2 замінює своїм ідентифікатором рядок з максимальною кількістю непарних елементів. У перехресті стовпчика і рядка записується сума ідентифікаторів потоків.

4.4 Контрольні питання

1) Чому у сучасних ОС крім процесів передбачене застосування потоків?
2) Сформулюйте, що таке потік і назвіть їх достоїнства у порівнянні з процесами.
3) Назвіть недоліки, що властиві потокам.
4) У чому полягає принципове розходження між системним викликом fork, котрим створюється новий процес і створенням нових потоків?
5) Чим відрізняється обробка потоків на багатоядерних процесорах від їх оброки на одноядерних?
6) Що таке реентерабельний код і яка основна вимога пред'являється до функцій які є реентерабельними?
7) Яка системна функція створює новий потік в процесі, яким чином ця функція повертає ідентифікатор створеного потоку?
8) Поясніть, що означає третій формальний параметр – void *(*start_routine)(void *) у прототипі функції створення потоку.
9) Чи може четвертій фактичний параметр у виклику функції створення потоку мати значення null і якщо так, то чому?
10) Поясніть, чому не можна повертати з функції pthread_exit покажчик на локальну змінну цієї функції.
11) За допомогою якої функції процес може дочекатись закінчення потоку і потім завершити його? Опишіть прототип функції.
12) Опишіть алгоритм синхронізації роботи потоків, так званим, циклом активного очікування.
13) Які методи синхронізації потоків пропонуються замість циклів акти-

вного очікування?

14) Чим і яким чином визначається вибір методу синхронізації за допомогою семафорів або м'ютексів?
15) Дайте визначення семафора.
16) Як слід розуміти атомарність функцій для роботи з семафорами?
17) Які два типи семафорів відомі і в яких випадках застосовується той чи іншій тип?
18) Яка вимога накладається стандартом POSIX на імена функцій управління семафорами? Назвіть ці функції.
19) Опишіть прототип функції за допомогою якої створюються семафори в ОС Linux?
20) Що можна сказати про другий фіктивний параметр, котрий визначає спільне використання семафорів, у функції їх створення?
21) Назвіть та опишіть пару функцій, що управляють значенням семафора у ОС Linux, тобто інкрементують й декрементують його значення.
22) Що трапляється коли функція декременту викликається для семафора зі значенням 0?
23) Яка функція стандарту POSIX знищую семафор після роботи з ним? Чи можна цією функцією знищити семафор на якому чекає потік?
24) Чи є необхідність перевіряти й виявляти в програмах помилки, що повертаються із системних викликів, що працюють з семафорами?
25) Що треба робити, щоб один з потоків не мав можливості виконати свій код кілька разів через недотримання часових умов?
26) Який тип мають м'ютекси у операційній системі Linux?
27) Чого, на відміну від переважної більшості функцій POSIX, на роблять функції управління м'ютексами?
28) Назвіть та опишіть функції сімейства роботи з м'ютексами.
29) Чи може й якщо так, то при яких умовах створитися тупикова ситуація в програмі у разі використання м'ютексів?
30) Яка функція створює запит на завершення потоку і чи можна й як

змінити реакцію потоку на цю функцію?

31) Для якої цілі слугує функція pthread_setcanceltype і які параметри функції вам відомі?
32) Приведіть визначення основної одиниці виконання коду в операційній системі Windows.
33) Яким чином у Windows забезпечується вимога володіння потоком власної області пам'яті, що належить тільки йому?
34) Опишіть прототип системного виклику CreateThread для створення нових потоків в адресному просторі вихідного процесу.
35) Якого розміру і як виконується виділення стеку для нового потоку у процесі?
36) Опишіть функцію, що повинна виконуватися в контексті процесу на котру вказує третій параметр функції CreateThread.
37) Як частіш за все інтерпретується потоком покажчик, котрий передається потоку у четвертому аргументі функції CreateThread.
38) Як затримати, а потім відновити безпосередній запуск потоку після виклику функції CreateThread.
39) За допомогою якої функції потік може сам завершити своє виконання, що при цьому трапляється?
40) Чому є украй небажаним застосовувати функцію TerminateThread.
41) Опишіть функцій, котрі існують в API для ідентифікації потоків.
42) За допомогою яких функцій потоки можуть надсилати один до одного повідомлення про призупинення и продовження, що при цьому трапляється?
43) Як здійснюється очікування завершення потоку і як працюють функції, що призначені для цього?
44) Назвіть механізми за допомогою яких синхронізуються потоки в Windows. Які з них є об'єктами ядра, а які – користувальницької програми?
45) Що це таке критична ділянка коду, які функції API управляють цими

об'єктами?

46) Як виконується блокування та визволення критичної секції коду?
47) Чи може той самий потік багаторазово повторно ввійти в одну й ту ж критичну секцію коду без її блокування?
48) Поясніть чому м'ютекси забезпечують більш універсальну функціональність у порівнянні з критичними ділянками коду.
49) Які переваги мають м'ютекси завдяки тому, що вони мають дескриптор?
50) Розкажіть про параметри, що передаються у функцію API CreateMutex і яке значення вона повертає?
51) Може або ні потік заволодівати тим самим м'ютексом кілька разів без блокування?
52) Яким чином повинні програмуватися потоки при роботі з м'ютексами?
53) Чи є обов'язковим завдання початкового значення лічильника у функції CreateSemaphore і якщо так, то якій умові воно завжди повинне задовольняти?
54) Чому не можна м'ютекс розглядати просто як окремий випадок семафора?
55) У яких випадках Найчастіше за все використовуються семафори?
56) Чому семафори у певному розумінні вважають зайвими елементами у системі програмування?
57) У чому полягає призначення об'єктів, котрі мають назву події?
58) Яку важливу додаткову можливість забезпечують об'єкти подій?
59) У чому полягає різниця між подіями, що скидаються вручну й такими, що скидаються автоматично?
60) Як довго в сигнальному стані залишаються події різного типу?
61) Чому у документах фірми Microsoft рекомендують уникати використання функції PulseEvent.

ПЕРЕЛІК ПОСИЛАНЬ

Основні джерела:

1. Шеховцов В.А. Операційні системи – К.: Видавнича група BHV, 2005. – 567 с.
2. Рольщиков В.Б. Конспект лекцій з курсу «Операційні системи» – Одеса: ОГЭКУ, 2015. – 89 с.
3. Рольщиков В.Б. Методичні вказівки до виконанні лабораторних робіт з курсу «Операційні системи» / Частина перша – Одеса: ОГЭКУ, 2013. – 89 с.

Додаткові джерела:

4. Таненбаум Э. Сучасні операційні системи. 2-е изд. – Спб.: Питер, 2002. – 1040 с.
5. Таненбаум Э., Вудхалл А. Операційні системи: розробка й реалізація – Спб.: Питер, 2006. – 576 с.

МЕТОДИЧНІ ВКАЗІВКИ

до виконанні лабораторних робіт з курсу
«Операційні системи»

(Частина друга)

для студентів денної форми навчання.
Напрям підготовки – 050101 Комп'ютерні науки,
рівень підготовки "бакалавр"

Укладач: Рольщиков В.Б.

Підписано до друку Формат Папір офсетний
Ум.друк.арк. Тираж Замовлення
Видавництво та друкарня

Одеський державний екологічний університет
65016, Одеса 16, вул. Львівська, 1