**Обмін даними між паралельними процесами у Windows**

Автора перекладу: Мирончук Олександр

Власний сайт: <http://oleksandr-myronchuk.blogspot.com>

github: <https://github.com/OleksandrMyronchuk>

sourceforge: <https://sourceforge.net/u/mr-oleksandr/profile>

Вміст деяких розділів було відредаговано.

Всі програми були створені в середовищі Visual Studio 2017.

**Цей посібник є безкоштовний та вільним для поширення.**

Зміст

[Вступ 6](#_Toc482137071)

[Обмін інформацією між процесами і потоками 7](#_Toc482137072)

[Обмін повідомленнями 7](#_Toc482137073)

[Додаток-приймач для демонстрації між процесної зв'язку 9](#_Toc482137074)

[Додаток-відправник для демонстрації зв'язку між процесами 11](#_Toc482137075)

[Передача даних за допомогою механізму повідомлень 16](#_Toc482137076)

[Передача даних процесу за допомогою повідомлення WM\_COPYDATA 16](#_Toc482137077)

[Прийом даних з процесу за допомогою повідомлення WM\_COPYDATA 17](#_Toc482137078)

[Обмін даними через файли 20](#_Toc482137079)

[Спільне використання файлів даних 20](#_Toc482137080)

[Спільне використання файлу декількома процесами 21](#_Toc482137081)

[Обмін даними через проекцію файлу в пам'яті 23](#_Toc482137082)

[Передача проекції файлу в інший процес 24](#_Toc482137083)

[Отримання проекції файлу з іншого процесу 26](#_Toc482137084)

[Обмін даними через сторінковий файл 29](#_Toc482137085)

[Обмін даними через сторінковий файл 30](#_Toc482137086)

[Читання даних з сторінкового файлу 31](#_Toc482137087)

[Передача даних 33](#_Toc482137088)

[Способи передачі даних між процесами 33](#_Toc482137089)

[Зв'язки між процесами 35](#_Toc482137090)

[Передача повідомлень 36](#_Toc482137091)

[Синхронний і асинхронний обмін даними 38](#_Toc482137092)

[Буферизація 38](#_Toc482137093)

[Робота з анонімними каналами в Windows 39](#_Toc482137094)

[Анонімні канали 39](#_Toc482137095)

[Створення каналів анонімних 40](#_Toc482137096)

[З'єднання клієнтів з анонімним каналом 41](#_Toc482137097)

[Обмін даними по анонімному каналу 42](#_Toc482137098)

[Приклади роботи з анонімними каналами 43](#_Toc482137099)

[Програма процесу-клієнта анонімного каналу 43](#_Toc482137100)

[Програма процесу-сервера анонімного каналу 44](#_Toc482137101)

[Процес-клієнт анонімного каналу 46](#_Toc482137102)

[Програма процес-сервера анонімного каналу 47](#_Toc482137103)

[Перенаправлення стандартного вводу-виводу 50](#_Toc482137104)

[Процес-клієнт, який записує дані в анонімний канал 51](#_Toc482137105)

[Процес-клієнт, читання даних з анонімного каналу 52](#_Toc482137106)

[Процес-сервер, анонімного каналу 53](#_Toc482137107)

[Робота з іменованими каналами в Windows 55](#_Toc482137108)

[Іменовані канали 55](#_Toc482137109)

[Створення іменованих каналів 55](#_Toc482137110)

[З'єднання сервера з клієнтом 57](#_Toc482137111)

[З'єднання клієнтів з іменованим каналом 58](#_Toc482137112)

[Обмін даними по іменованого каналу 61](#_Toc482137113)

[Процес-сервер іменованого каналу 61](#_Toc482137114)

[Процес клієнт, іменованого каналу 63](#_Toc482137115)

[Сервер іменованого каналу, який відповідатиме на повідомлення клієнта 64](#_Toc482137116)

[Клієнт іменованого каналу, який відсилає повідомлення серверу 66](#_Toc482137117)

[Сервер, який створює загальнодоступний іменований канал 68](#_Toc482137118)

[Копіювання даних з іменованого каналу 70](#_Toc482137119)

[Зразок копіювання повідомлення клієнтом іменованого каналу 72](#_Toc482137120)

[Передача транзакцій по іменованого каналу 73](#_Toc482137121)

[Приклад передачи транзакції по іменованому каналу 74](#_Toc482137122)

[Приклад передачи транзакцій по іменованому каналу 77](#_Toc482137123)

[Визначення та зміна стану іменованого каналу 78](#_Toc482137124)

[Приклад визначення стану іменованого каналу 80](#_Toc482137125)

[Приклад зміни стану іменованого каналу 82](#_Toc482137126)

[Отримання інформації про іменованому каналі 84](#_Toc482137127)

[Приклад отримання інформації по іменований канал 85](#_Toc482137128)

[Робота з поштовими скриньками в Windows 88](#_Toc482137129)

[Концепція поштових скриньок 88](#_Toc482137130)

[Створення поштових скриньок 89](#_Toc482137131)

[З'єднання клієнтів з поштовою скринькою 90](#_Toc482137132)

[Обмін даними через поштову скриньку 91](#_Toc482137133)

[Процес-сервер поштової скриньки 91](#_Toc482137134)

[Процес-клієнт поштової скриньки 93](#_Toc482137135)

[Отримання інформації про поштовій скриньці 94](#_Toc482137136)

[Процес-сервер поштової скриньки 95](#_Toc482137137)

[Процес-клієнт поштової скриньки 97](#_Toc482137138)

[Зміна часу очікування повідомлення 98](#_Toc482137139)

[Зміна часу очікування повідомлення 99](#_Toc482137140)

[Висновки 101](#_Toc482137141)

[Перелік використаних джерел 102](#_Toc482137142)

[Додатки 103](#_Toc482137143)

[Додаток-приймач для демонстрації між процесної зв'язку 103](#_Toc482137144)

[Додаток-відправник для демонстрації зв'язку між процесами 105](#_Toc482137145)

[Передача даних процесу за допомогою повідомлення WM\_COPYDATA 107](#_Toc482137146)

[Прийом даних з процесу за допомогою повідомлення WM\_COPYDATA 108](#_Toc482137147)

[Спільне використання файлу декількома процесами 110](#_Toc482137148)

[Передача проекції файлу в інший процес 111](#_Toc482137149)

[Отримання проекції файлу з іншого процесу 113](#_Toc482137150)

[Обмін даними через сторінковий файл 114](#_Toc482137151)

[Читання даних з сторінкового файлу 115](#_Toc482137152)

[Програма процесу-клієнта анонімного каналу 115](#_Toc482137153)

[Програма процесу-сервера анонімного каналу 116](#_Toc482137154)

[Процес-клієнт анонімного каналу 118](#_Toc482137155)

[Програма процес-сервера анонімного каналу 119](#_Toc482137156)

[Процес-клієнт, який записує дані в анонімний канал 121](#_Toc482137157)

[Процес-клієнт, читання даних з анонімного каналу 122](#_Toc482137158)

[Процес-сервер, анонімного каналу 123](#_Toc482137159)

[Процес-сервер іменованого каналу 125](#_Toc482137160)

[Процес клієнт, іменованого каналу 126](#_Toc482137161)

[Сервер іменованого каналу, який відповідатиме на повідомлення клієнта 128](#_Toc482137162)

[Клієнт іменованого каналу, який відсилає повідомлення серверу 129](#_Toc482137163)

[Сервер, який створює загальнодоступний іменований канал 131](#_Toc482137164)

[Зразок копіювання повідомлення клієнтом іменованого каналу 133](#_Toc482137165)

[Приклад передачи транзакції по іменованому каналу 135](#_Toc482137166)

[Приклад передачи транзакцій по іменованому каналу 136](#_Toc482137167)

[Приклад визначення стану іменованого каналу 138](#_Toc482137168)

[Приклад зміни стану іменованого каналу 139](#_Toc482137169)

[Приклад отримання інформації по іменований канал 141](#_Toc482137170)

[Процес-сервер поштової скриньки 142](#_Toc482137171)

[Процес-клієнт поштової скриньки 143](#_Toc482137172)

[Процес-сервер поштової скриньки 144](#_Toc482137173)

[Процес-клієнт поштової скриньки 146](#_Toc482137174)

[Зміна часу очікування повідомлення 148](#_Toc482137175)

# Вступ

Взаємодія між процесами (англ. inter-process communication, IPC) — обмін даними між потоками одного або різних процесів. Реалізується за допомогою механізмів, що надаються ядром ОС або процесом, що використовує механізми ОС та реалізує нові можливості IPC. Може здійснюватися як на одному комп'ютері, так і між декількома комп'ютерами мережі.

З механізмів, що надаються ОС і використовуються для IPC, можна виділити:

* механізми обміну повідомленнями;
* механізми синхронізації;
* механізми розподілу пам'яті;
* механізми віддалених викликів (RPC).

Для оцінки продуктивності різних механізмів IPC використовують наступні параметри:

* пропускна здатність (кількість повідомлень в одиницю часу, яке ядро ОС або процес здатний обробити);
* затримки (час між відправленням повідомлення одним потоком і його отриманням іншим потоком).

IPC може називатися термінами межпотоковое взаємодія (англ. inter-thread communication) і міжпрограмний взаємодія (англ. inter-application communication). Взаємодія між процесами, поряд з механізмами адресації пам'яті, є основою для розмежування адресного простору між процесами.

# Обмін інформацією між процесами і потоками

Як правило, потоки, що співіснують в рамках одного процесу, виконують взаємопов'язані завдання і робота одного потоку може залежати від ходу або результатів роботи іншого. В цьому випадку необхідно організувати передачу інформації з потоку в потік. В якості переданої інформації часто виступають дані, підготовлені, наприклад, в одному потоці, але використовувані в іншому. Однак з потоку в потік (і з процесу в процес) можна передавати не тільки дані, але і накази на виконання заданих дій. Зупинимося спочатку на міжпоточної передачі даних. У самій передачі даних немає ніякої складності, так як всі потоки процесу працюють в єдиному адресному просторі і всім їм доступні глобальні змінні і функції процесу. Тому для передачі деяких даних з одного потоку в інший досить ці дані оголосити глобальними. Реально, однак, міжпоточної обмін даними часто вимагає вирішення завдань синхронізації. Дійсно, якщо один потік, наприклад, формує дані для обробки їх іншим потоком, то інший потік повинен бути повідомлений про момент готовності даних і не повинен намагатися приймати їх раніше цього моменту. Питання синхронізації паралельно працюючих потоків, як що належать одному процесу, так і виконуваних в контексті різних процесів, будуть розглянуті в наступному розділі. Поточний розділ присвячений способам передачі інформації між процесами. Цих способів не так вже й багато. До них відносяться:

* передача в дочірній процес при його запуску параметрів командного рядка;
* використання механізму повідомлень;
* спільне використання файлів даних;
* спільне використання сторінкового файлу.

Тут ми зупинимося на інших трьох перед останніх способах.

## Обмін повідомленнями

Як видно з численних прикладів, наведених в цій книзі, механізм повідомлень відноситься до числа основних концепцій системи Windows. Заповнення вікон програми будь-якими зображеннями, відгук на вибір пунктів меню, управління діалоговими вікнами - все це здійснюється за допомогою ініціювання та обробки тих чи інших повідомлень Windows. При цьому до цих пір ми стикалися з повідомленнями або які надходять з Windows у вікно нашого застосування, або ініційованих в додатку з метою впливу на елементи управління (наприклад, для занесення нового рядка в елемент управління - список). Як буде показано в цьому розділі, повідомлення можна надсилати також з однієї програми в іншу (з процесу в процес), створюючи між процесами певний зв'язок.

Посилка повідомлення в інший процес здійснюється викликом функції *SendMessage()*, в числі параметрів якого вказується дескриптор вікна, якому надсилається дане повідомлення, код повідомлення і відповідні конкретного повідомлення параметри. Всередині розроблювального додатка не складає труднощів зберегти дескриптори усіх створених вікон; дескриптори вікон інших процесів ми, звичайно, не знаємо. однак для їх визначення передбачена функція Windows *FindWindow()* з наступним прототипом:

*HWND FindWindow (*

*LPCTSTR lpClassName, // Ім'я класу шуканого вікна*

*LPCTSTR lpWindowName // Тема шуканого вікна*

*);*

Таким чином, отримати дескриптор вікна активного додатка Windows не становить труднощів - треба лише знати ім'я класу цього вікна або його заголовок. Якщо функція *FindWindow()* знаходить серед активних потрібне, вона повертає значення дескриптора цього вікна, якщо вікно не знайдено, повертається *FALSE*.

Отримавши дескриптор вікна програми, з яким ми хочемо встановити зв'язок, можна посилати в нього будь-які повідомлення, які мають для програми-приймача сенс (такі, які в ньому обробляються, хоча б і за замовчуванням). Зрозуміло, посилка повідомлення в додаток має на увазі, що в додатку-приймачі існує черга повідомлень і цикл обробки повідомлень, тобто що воно є інтерфейсним.

Розробимо програмний комплекс, що складається з двох додатків – «перший додаток», «другий додаток».

Перше буде являти собою процес-відправник, друге - процес-приймач. Нехай обидві програми мають звичайним інтерфейсом з головним вікном і меню. У додатку-приймачі за допомогою меню можна відкрити модальний діалог з інформацією про програму; в додатку-відправника передбачимо можливість виконання наступних дій:

* створення дочірнього процесу-приймача і визначення дескриптора його головного вікна;
* посилка в процес-приймач повідомлення *WM\_SETTEXT*, що змінює заголовок його головного вікна;
* посилка в процес-приймач повідомлення *WM\_COMMAND*, що імітує вибір в цьому додатку певного пункту меню;
* посилка в процес-приймач повідомлення *WM\_DESTROY* «насильницького» завершення цього додатка.

Зауважимо, що поки ми розглядаємо тільки передачу в процес-приймач стандартних повідомлень Windows; пересиланню прикладних даних буде присвячений наступний приклад.

Наведемо спочатку текст програми-приймача «перший додаток». Вона являє собою цілком закінчений автономне додаток, який запускається з Робочого столу Windows або через меню «Пуск», після чого за допомогою команд меню можна керувати його роботою - викликати вікно модального діалогу з інформацією про програму або завершити її роботу.

### Додаток-приймач для демонстрації між процесної зв'язку

header.h

#pragma once

#include <Windows.h>

#define MI\_ABOUT 100//Ідентифікатори пунктів меню

#define MI\_EXIT 101

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void OnCommand(HWND, int);

void OnPaint(HWND);

void OnDestroy(HWND);

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

BOOL DlgOnInitDialog();

void DlgOnCommand(HWND, int);

resourse.rc

#include «header.h»

/\* Сценарій меню \*/

Main MENU

{

POPUP «Файл»

{

MENUITEM «Про програму», MI\_ABOUT

MENUITEM «Вихід», MI\_EXIT

}

}

/\* Сценарій діалогу \*/

About DIALOG 65, 0, 100, 50

STYLE WS\_POPUP | WS\_CAPTION | WS\_SYSMENU | DS\_MODALFRAME

CAPTION «Про програму»

{

CTEXT «Демонстрація зв'язку процесів», -1, 12, 4, 72, 25

PUSHBUTTON «Закрити», IDOK, 10, 24, 80, 12

}

main.cpp

#include «header.h»

wchar\_t szClassName[] = L»WinRcv»;//Ім'я класу вікна-приймача

wchar\_t szTitle[] = L»Перший додаток»;//Заголовок вікна-приймача

HINSTANCE hlnstance;//Дескриптор додатку

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE hlnst, HINSTANCE, LPSTR, int)

{

MSG Msg;

WNDCLASS wc;

ZeroMemory(&wc, sizeof(WNDCLASS));//обнулити структуру

wc.lpfnWndProc = WndProc;//зворотня функція головного вікна

wc.hInstance = hlnst; //дескриптор додатку

wc.hIcon = LoadIcon(nullptr, IDI\_APPLICATION);//іконка

wc.hCursor = LoadCursor(nullptr, IDC\_ARROW);//курсор

wc.hbrBackground = CreateSolidBrush(RGB(255, 255, 255));//фонова заливка

wc.lpszMenuName = L»Main»;//назва меню

wc.lpszClassName = szClassName;//назва класу вікна

RegisterClass(&wc);//реєструє клас вікна для подальшого використання у викликах функції CreateWindow

HWND hwnd = CreateWindow(//Створимо вікно

szClassName, // ім'я класу

szTitle, //назва вінка

WS\_OVERLAPPEDWINDOW,//стиль

210, 10, 200, 100,//x, y, width, height

HWND\_DESKTOP,//батько вікна

nullptr,//дескриптор меню (не знайшов як взяти дескриптор .rc файлу !)

hlnst, //дескриптор додатку

nullptr//lpParam

);

ShowWindow(hwnd, SW\_NORMAL); //Покажемо вікно

while (GetMessage(&Msg, nullptr, 0, 0)) //цикл обробки повідомлень

DispatchMessage(&Msg);//відправляє повідомлення в процедуру вікна

return 0;

}

/\*віконна функція головного вікна\*/

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_PAINT: OnPaint(hwnd); break;//вивести текст

case WM\_COMMAND: OnCommand(hwnd, wParam); break;//обробити команду від пунктів меню

case WM\_DESTROY: OnDestroy(hwnd); break;//закрити

default://стандартна обробка повідомлень

return(DefWindowProc(hwnd, msg, wParam, lParam));

}

}

/\*функція обробки повідомлень WM\_PAINT\*/

void OnPaint(HWND hwnd)

{

PAINTSTRUCT ps;

wchar\_t szText[] = L»Процес-приймач»;

HDC hdc = BeginPaint(hwnd, &ps);

TextOutW(hdc, 5, 10, szText, wcslen(szText) + 1);

EndPaint(hwnd, &ps);

}

/\*функція обробки повідомлень WM\_COMMAND від пунктів меню\*/

void OnCommand(HWND hwnd, int id)

{

switch (id)

{//Ідентифікатор пункту меню

case MI\_ABOUT: //Відкрити діалог

DialogBoxW(hlnstance, L»About», hwnd, DlgProc);

break;

case MI\_EXIT: //Завершення процесу

DestroyWindow(hwnd);

}

}

/\*функція обробки повідомлення WM\_DESTROY\*/

void OnDestroy(HWND)

{

PostQuitMessage(0);

}

/\*віконна функція діалогу\*/

BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_INITDIALOG: DlgOnInitDialog(); break;

case WM\_COMMAND: DlgOnCommand(hwnd, wParam); break;

default:

return FALSE;

}

}

/\*функція обробки повідомлення WM\_INITDIALOG для діалогового вікна\*/

BOOL DlgOnInitDialog()

{

return TRUE;

}

/\*функція обробки повідомлення WM\_COMMAND для діалогового вікна\*/

void DlgOnCommand(HWND hwnd, int id)

{

switch (id)

{//Код елемента керування(кнопки)

case IDOK://Натиснута кнопка «Закрити»

case IDCANCEL://Дана команда закрити діалог через його системне меню

EndDialog(hwnd, 0);//Закриття діалогового вікна

}

}

Наведемо спочатку текст програми-приймача. Вона являє собою цілком закінчений автономне додаток, який запускається з Робочого столу Windows або через меню «Пуск», після чого за допомогою команд меню можна керувати його роботою - викликати вікно модального діалогу з інформацією про програму або завершити її роботу.

### Додаток-відправник для демонстрації зв'язку між процесами

header.h

#pragma once

#include <Windows.h>

#define MI\_ABOUT 100

#define MI\_NEW 100

#define MI\_LINK1 101

#define MI\_LINK2 102

#define MI\_LINK3 103

#define MI\_EXIT 104

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void OnCommand(HWND, int);

void OnDestroy(HWND);

BOOL OnCreate(HWND, LPCREATESTRUCT);

void OnPaint(HWND);

void OnCopyData(HWND, HWND, COPYDATASTRUCT\*);

resours.rc

#include «header.h»

Main MENU

{

POPUP «Файл»

{

MENUITEM «Запуск процесу», MI\_NEW

MENUITEM «Зміна заголовка», MI\_LINK1

MENUITEM «Вибір пункту меню», MI\_LINK2

MENUITEM «Завершення процесу», MI\_LINK3

MENUITEM SEPARATOR

MENUITEM «Вихід», MI\_EXIT

}

}

main.cpp

#include «header.h»

wchar\_t szClassName[] = L»WinSnd»;//Ім'я класу головного вікна

wchar\_t szTitle[] = L»Другий додаток»;//Заголовок головного вікна

HWND hwndRcv;//Дескриптор головного вікна процесу-приймача

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int) {

MSG Msg;

WNDCLASS wc;

ZeroMemory(&wc, sizeof(wc));

wc.lpfnWndProc = WndProc;

wc.hInstance = hInst;

wc.hIcon = LoadIcon(nullptr, IDI\_APPLICATION);

wc.hCursor = LoadCursor(nullptr, IDC\_ARROW);

wc.hbrBackground = CreateSolidBrush(RGB(255, 255, 255));

wc.lpszMenuName = L»Main»;

wc.lpszClassName = szClassName;

RegisterClass(&wc);

HWND hwnd = CreateWindowW(//Створюємо головне вікно

szClassName,

szTitle,

WS\_OVERLAPPEDWINDOW,

10, 10, 200, 100,

HWND\_DESKTOP,

nullptr,

hInst,

nullptr

);

ShowWindow(hwnd, SW\_NORMAL);//Показуємо вікно

while (GetMessage(&Msg, nullptr, 0, 0))//Цикл обробки повідомлень

DispatchMessage(&Msg);

return 0;

}

/\*віконна функція головного вікна\*/

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_PAINT: OnPaint(hwnd); break;

case WM\_COMMAND: OnCommand(hwnd, wParam); break;

case WM\_DESTROY: OnDestroy(hwnd); break;

default:

return(DefWindowProc(hwnd, msg, wParam, lParam));

}

}

/\*функція обробки повідомлень WM\_COMMAND від пунктів меню\*/

void OnCommand(HWND hwnd, int id)

{

switch (id)

{

case MI\_NEW:

{//Запустимо процес-приймач

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

hwndRcv = FindWindowW(L»WinRcv», nullptr);

if (!hwndRcv)

{

if (CreateProcessW(L»PR\_1\_win32PB\_page\_354.exe», nullptr, nullptr, nullptr, FALSE, 0, nullptr, nullptr, &si, &pi))

{

Sleep(250);

while (!hwndRcv)//Дочекаємося запуску процесу

hwndRcv = FindWindowW(L»WinRcv», nullptr);//Отримаємо дескриптор вікна

}

}

else

MessageBoxW(nullptr, L»Процес вже запущений», L»Info», MB\_OK);

break;

}//Кінець case MI\_NEW

case MI\_LINK1://Змінимо заголовок процесу-приймача

SendMessageW(hwndRcv, WM\_SETTEXT, 0, (LPARAM)L»Новий заголовок»);

break;

case MI\_LINK2://Виберемо пункт меню процесу-приймача

SendMessage(hwndRcv, WM\_COMMAND, (WPARAM)MI\_ABOUT, (LPARAM)0);

break;

case MI\_LINK3://Завершимо процес-приймач

SendMessage(hwndRcv, WM\_DESTROY, 0, 0);

break;

case MI\_EXIT://Завершимо процес-відправник

DestroyWindow(hwnd);

}

}

/\*функція обробки повідомлень WM\_PAINT\*/

void OnPaint(HWND hwnd)

{

wchar\_t szT[] = L»Процес-відправник»;

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hwnd, &ps);

TextOut(hdc, 5, 10, szT, wcslen(szT) + 1);

EndPaint(hwnd, &ps);

}

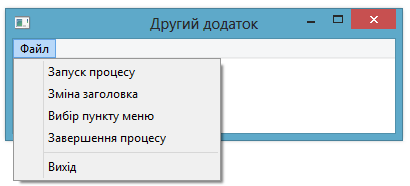
/\*функція обробки повідомлення WM\_DESTROY\*/

void OnDestroy(HWND)

{

PostQuitMessage(0);

}



Головне вікно процесу-відправника

відкритим меню

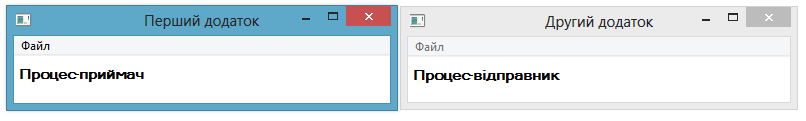
рис. 1

При запуску програми «Другий додаток» а на екран виводиться його головне вікно зі рядком тексту, яка на рис. 1 затулилася відкритим меню.

Всі змістовні дії програми зосереджені в функції *OnCommand()*. При виборі пункту меню «Запуск процесу» (ідентифікатор *MI\_NEW*) перш за все здійснюється спроба знайти серед активних процесів вікно з ім'ям класу «WinRcv».

Якщо таке вікно знайдено (функція *FindWindow()* повертає ненульовий дескриптор), на екран виводиться попередження і обслуговування даного пункту меню завершується; якщо вікна немає, створюється процес «Перший додаток». Зазначена перевірка захищає програмний комплекс від запуску повітряних примірників одного і того ж процесу-приймача.

Після створення процесу необхідно отримати дескриптор його головного вікна. Тут ми знову стикаємося з проблемою синхронізації множинних процесів одного програмного комплексу. Справа в тому, що фактичне створення процесу вимагає часу; якщо після виклику функції *CreateProcess()* відразу ж спробувати отримати дескриптор його вікна, то функція *FindWindow()* поверне *FALSE*, так як до цього моменту новий процес ще не буде створено. Тому в програмі виконуються циклічні спроби отримати дескриптор вікна процесу-приймача до тих пір, поки функція *FindWindow()* не поверне нульове значення. Отримане значення дескриптора зберігається в глобальній змінної *hwndRcv* для подальшого використання. На рис. 2 показано зображення на Робочому столі після запуску процесу-приймача.



З батьківського процесу запущений процес-приймач

Рис. 2

Подальші дії користувача полягають у виборі різних пунктів меню батьківського процесу. При виборі пункту «Зміна заголовка» (ідентифікатор *MI\_LINK1*) в процес-приймач надсилається повідомлення *WM\_SETTEXT*, яке дозволяє змінити заголовок його вікна (не затверджувати, що це дуже корисна дія, проте воно наочно демонструє пересилання повідомлень між процесами).

При виборі пункту «Завершення процесу» (ідентифікатор *MI\_LINK3*) в процес-приймач надсилається повідомлення *WM\_DESTROY*, яка природним чином наводить до завершення дочірнього процесу.

Найбільш цікава команда «Вибір пункту меню» (ідентифікатор *MI\_LINK2*). У цьому випадку в процес-приймач надсилається повідомлення *WM\_COMMAND*, яке імітує вибір в цьому процесі будь-якого пункту його меню. Для того щоб правильно сформувати повідомлення *WM\_COMMAND*, треба подивитися довідку про це повідомленні в інтерактивному довіднику:

*WM\_COMMAND*

*wNotifyCode = HIWORD(wParam);//* *Код повідомлення*

*wID = LOWORD(wParam);//* *Ідентифікатор пункту АБО елемента керування*

*hwndCtl = (HWND) lParam;//* *Дескриптор елемента керування*

і зіставити параметри повідомлення з параметрами функції *SendMessage()*, призначення яких випливає з прототипу функції:

*LRESULT SendMessage(*

*HWND hwnd, //Дескриптор вікна-приймача повідомлення*

*UINT uMsg, //Код повідомлення (в нас WM\_COMMAND)*

*WPARAM wParam, //Перший параметр повідомлення*

*LPARAM Iparam //Другий параметр повідомлення*

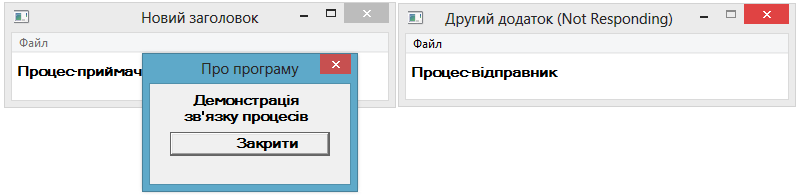
*) ;*

З наведених довідок видно, що в старшій половині параметра wParam функції *SendMessage()* повинен міститися код повідомлення, а в молодшій половині - ідентифікатор пункту меню (згадаємо, що в 32-розрядних додатках обидва параметри, і *wParam* і *lParam*, мають розмірність 32 біта). Коду повідомлення в нашому випадку немає, а ідентифікатор активізованого нами пункту меню «Про програму» процесу-приймача дорівнює *MI\_ABOUT*. У файл header.h нами був завбачливо включений макрос *#define* з визначенням значення цього ідентифікатора, і ми можемо скористатися цим символічним позначенням при виконанні функції *SendMessage()*:

*SendMessage (hwndRcv, WM\_COMMAND, (WPARAM) MI\_ABOUT, (LPARAM) 0);*

Зрозуміло, важливо тут не символічне позначення пункту меню *MI\_ABOUT*, а його значення (100), яке повинно бути таким же, як і в процесі-приймачі. Функція *SendMessage()* очікує отримати в якості третього і четвертого параметрів змінні типу *WPARAM* і *LPARAM* відповідно. Тому ми для наочності перетворимо символічну константу *MI\_ABOUT* в тип *WPARAM*, а останній 0 - в *LPARAM*, хоча в даному випадку явні перетворення необов'язкові.

На рис. 3 показано зображення на Робочому столі після посилки в процес-приймач повідомлень про зміну заголовка вікна і про вибір пункту меню «Про програмі».



Стан процесу-приймача після заміни заголовка його вікна і активізації модального діалогу з процесу-відправника

Мал. 3

## Передача даних за допомогою механізму повідомлень

Розглянемо тепер питання про передачу в інший процес масиву даних. Безпосередньо це зробити не можна, так як, хоча всі процеси працюють в одному і тому ж віртуальному (і лінійному) адресному просторі, їх фізичні адреси не перекриваються і один процес не має доступу до змінних іншого. Однак для передачі даних можна скористатися спеціально для цього призначених повідомленням *WM\_COPYDATA*, як непрямого параметра якого може виступати адреса будь-якої колекції даних.

Розробимо програмний комплекс, в якому батьківський процес-відправник («Третій додаток») готує значний за обсягом масив даних і передає його запущеному їм же процесу-приймача («Четвертий додаток»), а той займається пересиланням цього масиву за призначенням (на віддалений комп'ютер, принтер або просто диск; в нашому прикладі масив записується на диск). Для простоти в батьківському процесі не будемо зв'язуватися з меню або навіть з головним вікном; нехай цей процес носить безінтерфейсний, чисто обчислювальний характер. З дочірнім процесом-приймачем так вчинити не вдасться, так як для прийому повідомлення йому потрібні і віконна функція з циклом обробки повідомлень, і саме вікно.

### Передача даних процесу за допомогою повідомлення WM\_COPYDATA

main.cpp

#include <Windows.h>

#define sizeArray 1048576

HWND hwndRcv;//Дескриптор вікна процесу-приймача повідомлення

int nData[sizeArray];//Пересылаемый масив даних

/\*Головна функція програми WinMain\*/

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

/\*Створимо дочірній процес і отримаємо його дескриптор вікна\*/

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

hwndRcv = FindWindowW(L"WinRcv", nullptr);

if (!hwndRcv)

{

CreateProcess(L" WinRcv.exe", nullptr, nullptr, nullptr, FALSE, 0, nullptr, nullptr, &si, &pi);

while (!hwndRcv)

{

Sleep(250);

hwndRcv = FindWindowW(L"WinRcv", nullptr);

}

}

else

MessageBoxW(nullptr, L"Процес вже запущений", L"Info", MB\_OK);

/\*Підготуємо масив що передається в дочірній процес\*/

for (int i = 0; i < sizeArray; i++)

nData[i] = i \* 10;

/\*Передамо дані в інший процес\*/

COPYDATASTRUCT cds;

cds.dwData = 0;

cds.cbData = (sizeArray + 1) \* sizeof(int);

cds.lpData = (PVOID)nData;

SendMessage(hwndRcv, WM\_COPYDATA, (WPARAM)nullptr, (LPARAM)&cds);

return 0;//Завершимо додаток

}

### Прийом даних з процесу за допомогою повідомлення WM\_COPYDATA

header.h

#pragma once

#include <Windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void OnDestroy();

BOOL OnCopyData(HWND, COPYDATASTRUCT\*);

main.cpp

#include "header.h"

wchar\_t szClassName[] = L"WinRcv";

wchar\_t szTitle[] = L"Четвертий додаток";

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int)

{

MSG Msg;

WNDCLASS wc;

ZeroMemory(&wc, sizeof(wc));

wc.lpfnWndProc = WndProc;

wc.hInstance = hInst;

wc.hIcon = LoadIcon(nullptr, IDI\_APPLICATION);

wc.hCursor = LoadCursor(nullptr, IDC\_ARROW);

wc.hbrBackground = CreateSolidBrush(RGB(255, 255, 255));

wc.lpszClassName = szClassName;

RegisterClass(&wc);

HWND hwnd = CreateWindowW(

szClassName,

szTitle,

WS\_OVERLAPPEDWINDOW,

210, 10, 200, 100,

HWND\_DESKTOP,

nullptr,

hInst,

nullptr

);

ShowWindow(hwnd, SW\_NORMAL);

while (GetMessage(&Msg, nullptr, 0, 0))

DispatchMessage(&Msg);

return 0;

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_COPYDATA:

{

COPYDATASTRUCT\* pcds = (COPYDATASTRUCT\*)lParam;

OnCopyData(hwnd, pcds);

break;

}

case WM\_DESTROY: OnDestroy(); break;

default:

return (DefWindowProc(hwnd, msg, wParam, lParam));

}

}

/\*функція обробки повідомлення WM\_COPYDATA \*/

BOOL OnCopyData(HWND hwnd, COPYDATASTRUCT\* pcds)

{

DWORD dwCnt;

HANDLE hFile = CreateFileW(//Створимо файл

L"1.1",

GENERIC\_WRITE,

0,

nullptr,

CREATE\_ALWAYS,

0,

nullptr

);

WriteFile(hFile, pcds->lpData, pcds->cbData, &dwCnt, nullptr); // Запис даних

CloseHandle(hFile);//Закриємо дескриптор файлу

DestroyWindow(hwnd);//Завершимо додаток

return TRUE;//Після обробки WM\_COPYDATA слід повернути TRUE

}

/\*функція обробки повідомлення WM\_DESTROY\*/

void OnDestroy()

{

PostQuitMessage(0);

}

Як вже зазначалося, додаток-відправник даних складається з єдиної функції *WinMain()* - В ній робиться спроба знайти вікно активного процесу з ім'ям класу *«WinRcv»* і, якщо такого вікна немає, створюється процес-приймач Четвертий додаток (запускається додаток «Четвертий додаток»). У будь-якому випадку змінної hwndRcv зберігається дескриптор вікна процесу-приймача.

Далі готуються дані для передачі в процес-приймач (на прикладі цілочисельний масив *nData* заповнюється послідовним рядом чисел, кратних 10), після чого виконуються операції по пересиланню даних у складі повідомлення *WM\_COPYDATA* вікна з дескриптором *hwndRcv*.

Використання повідомлення *WM\_COPYDATA* вимагає наявності в програмі структурної змінної типу *COPYDATASTRUCT*. У файлі *WINUSER.H* визначено склад цієї структури:

*typedef struct tagCOPYDATASTRUCT {*

*DWORD dwData;//Довільний параметр, переданий додатком-приймача*

*DWORD cbData;//Розмір в байтах переданих даних*

*PVOID ipData;//Адреса даних, що передаються додатком-приймача*

*} COPYDATASTRUCT;*

Видно, що через елемент *IpData* процесу-приймача можна передати адресу будь-якої колекції даних (оскільки цей елемент має тип узагальненого адреси *PVOID*), через елемент *cbData* - обсяг переданих даних, а через елемент *dwData* - будь-який додатковий параметр. Ініціалізувавши ці елементи структурної змінної належним чином, ми викликом функції *SendMessage()* посилаємо в процес-приймач повідомлення *WM\_COPYDATA*. В якості параметрів цього повідомлення вказується дескриптор вікна передавального процесу (але у нас вікна в передавальному процесі немає, і цей параметр має значення *nullptr*) та адреса структурної змінної типу *COPYDATASTRUCT*.

Якщо функція *SendMessage()* використовується для передачі даних в інший процес, тобто передає повідомлення *WM\_SETTEXT* або *WM\_COPYDATA*, то вона працює досить складним чином. Для того щоб дані процесу-відправника були доступні процесу-приймача, функція виділяє в адресному просторі процесу-приймача область лінійних адрес, що накладаються на дані, що пересилаються. Базовий адреса цієї області передається в процес-приймач в якості елемента *IpData* структури *COPYDATASTRUCT*, що дає можливість процесу-приймача читати дані, що пересилаються.

Далі функція *SendMessage()* здійснює синхронну пересилання повідомлення. Це означає, що управління повернеться в процес-відправник (на пропозицію програми, наступне за викликом функції *SendMessage()*) тільки після того, як повідомлення буде повністю оброблено у процесі-приймачі, тобто коли у функції *OnCopyData()* буде виконаний останній оператор

*return TRUE;*

(згідно з документацією Microsoft, якщо повідомлення *SendMessage()* обробляється в прикладній програмі, вона повинна повернути значення *TRUE*). Таким чином, процес-відправник, відправивши повідомлення *WM\_COPYDATA*, зупиняється до повернення з функції, що оброблює це повідомлення в процесі-приймачі. Тому бажано, щоб обробка повідомлення не займала дуже багато часу.

Нарешті, останнє. Адреса даних що пересилаються, передається в процес-приймач разом зі структурою *COPYDATASTRUCT*, лише поки триває обробка повідомлення *WM\_COPYDATA*. Як тільки функція, що обробляла це повідомлення, завершується, система знищує відображення виділеного ділянки адрес на дані, що пересилаються. Враховуючи попередню замітку, розумно в функції обробки повідомлення *WM\_COPYDATA* не обробляти дані, а лише скопіювати їх в спеціально передбачений для цього (або динамічно виділений) буфер. Повернувши управління процесу-відправнику, процес-приймач може тепер вже без поспіху зайнятися обробкою отриманих даних, що вимагатиме введення в нього якого-небудь механізму синхронізації.

У функції *OnCopyData()* виконується створення нового файлу *1.1* і запис у нього всього масиву даних, отриманого через структуру *COPYDATASTRUCT*. Своїм останнім дією функція *OnCopyData()* завершує додаток-приймач.

# Обмін даними через файли

## Спільне використання файлів даних

У тих випадках, коли швидкість обробки даних не має вирішального значення, обмін даними між процесами можна виконувати через колективні файли. Для того щоб зробити файл подільним, досить при його відкритті вказати прапори *FILE\_SHARE\_READ* або *FILE\_SHARE\_WRITE* (або їх комбінацію). Розглянемо приклад спільного використання файлу двома процесами. Найпростіше в якості двох процесів просто використовувати два примірники одного додатка. Для того щоб не ускладнювати собі життя процедурами синхронізації, будемо вважати, що процеси звертаються до загального файлу тільки з метою читання з нього окремих ділянок. В якості робочого файлу скористаємося файлом *STOCK.DAT*. У цьому файлі розміром 8000000 байт ніби записана інформація про номери і вартостях якихось зразків, причому значення вартості завжди в 10 разів більше номера зразка, що зручно для контролю правильності читання файлу.

Організуємо програму «п’ятий додаток» на базі модального діалогу з елементом керування списком. Нехай додавання в список черговий вибірки з файлу здійснюється за натисканні кнопки, а номер вибірки визначається випадковим чином за допомогою датчика випадкових чисел.

### Спільне використання файлу декількома процесами

header.h

#pragma once

#include <Windows.h>

#define ID\_LIST 100

#define ID\_CHOICE 101

struct GOODS

{ //Структура, що відображає вміст файлу

int Nmb;

int Price;

};

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

void DlgOnCommand(HWND, int);

BOOL DlgOnlnitDialog();

resource.rc

#include "header.h"

Files DIALOG 41, 23, 84, 115

STYLE WS\_POPUP | WS\_CAPTION | WS\_SYSMENU

CAPTION L"Вибірки з файлу"

{

LISTBOX ID\_LIST, 5, 4, 74, 88, WS\_VSCROLL | WS\_BORDER

PUSHBUTTON L"Вибірка", ID\_CHOICE, 12, 98, 50, 14, BS\_PUSHBUTTON

}

main.cpp

#include "header.h"

HANDLE hFile;//Дескриптор файлу

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE hlnst, HINSTANCE, LPSTR, int)

{

DialogBoxW(hlnst, L"Files", nullptr, DlgProc);

return 0;

}

/\*Віконна функція діалогового вікна\*/

BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_INITDIALOG: DlgOnlnitDialog(); break;

case WM\_COMMAND: DlgOnCommand(hwnd, wParam); break;

default:

return FALSE;

}

}

/\*функція обробки повідомлення WM\_INITDIALOG\*/

BOOL DlgOnlnitDialog()

{

hFile = CreateFileW(

L"stock.dat",

GENERIC\_READ, //відкриємо файл для читання

FILE\_SHARE\_READ,//c поділяється доступом

nullptr,

OPEN\_EXISTING,

0,

nullptr);

return TRUE;

}

/\*функція обробки повідомлень від елементів управління діалогом\*/

void DlgOnCommand(HWND hwnd, int id)

{

switch (id)

{

case ID\_CHOICE:

{

GOODS Stock;//Змінна для читання з файлу

DWORD dwCount;//Число прочитаних байтів

wchar\_t szUnit[20];//Поле для формування рядків списку

int numb = rand() % 15;//Отримаємо випадкове число

SetFilePointer(hFile, numb \* sizeof(GOODS), 0, FILE\_BEGIN);//Зрушимо вказівник

ReadFile(hFile, &Stock, sizeof(Stock), &dwCount, nullptr);//Прочитаємо

wsprintfW(szUnit, L"%d %d", Stock.Nmb, Stock.Price);

SendDlgItemMessage(hwnd, ID\_LIST, LB\_ADDSTRING, 0, (LPARAM)szUnit);//У список

break;

}

case IDCANCEL:

EndDialog(hwnd, 0);

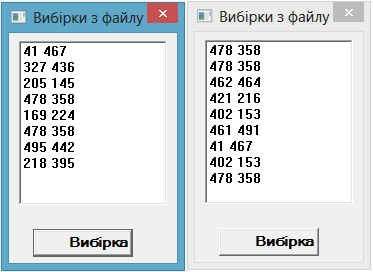
}

}

У функції *DlgOnlnitDialog()* файл *STOCK.DAT* відкривається для читання з прапором доступу *FILE\_SHARE\_READ*. Для того, щоб при запуску декількох примірників додатків, кожен з них здійснював вибірку з файлу випадковим чином, викликом функції *C++* *rand()* датчик випадкових чисел ініціалізується випадковим значенням, отриманим як деяка похідна від поточного часу).

У відповідь на кожне натискання кнопки "Вибірка" випадкове число, отримане від генератора випадкових чисел, множиться на розмір структури *GOODS* і використовується в якості зсуву для встановлення файлового вказівника. З файлу читається ділянку, яка відповідна за розміром структурі *GOODS* (фактично два цілих числа). Далі ці числа перетворюються в символьну форму і отриманий рядок додається в список.

На рис. 4 наведено результати сеансу роботи з двома примірниками цього додатка. При бажанні можна було запустити і більшу кількість примірників.



Мал. 4. Два примірники програми працюють з одним файлом

Легко переконатися в тому, що спільна робота двох процесів з одним файлом стало можливо тільки тому, що файл відкрито в режимі поділу. Досить замінити при відкритті файлу прапор *FILE\_SHARE\_READ* на 0 - й наступні екземпляри програми звертатися до файлу не зможуть.

# Обмін даними через проекцію файлу в пам'яті

Вище ми познайомилися з методикою проектування файлів даних в пам'ять. Використання проекцій файлів дозволяє спростити програму і підвищити ефективність її роботи за рахунок перекладання на Windows завдань завантаження в пам'ять необхідних ділянок файлів і, при необхідності, модифікації їх на диску. сутність методики полягає в тому, що після створення у фізичній пам'яті спеціального об'єкта «проекція файлу» виконується відображення цієї проекції на адресний простір процесу, що і забезпечує звернення до файлу за адресами пам'яті, як якщо б файл був звичайною програмної змінної.

Використання проекцій файлів в пам'яті для передачі даних між процесами ґрунтується на тому, що об'єкт "проекція файлу" після свого створення виявляється доступним для всіх процесів, що виконуються в системі. Вихідний процес створює проекцію файлу за допомогою функції *CreateFileMapping()*. Процедура доступу до створеної проекції з боку інших процесів включає в себе два етапи:

* відкриття наявної проекції за допомогою функції *CreateFileMapping()*;
* відображення відкритої проекції на адресний простір процесу за допомогою функції *MapViewOfFile()*.

Модифікуємо попередній приклад, залишивши його змістовну частину – випадкові вибірки записів з файлу *STOCK.DAT* і приміщення їх в список, але замість роботи з файлом на диску використовуємо проекцію файлу в пам'яті. Для демонстрації передачі даних з процесу на процес організуємо програмний комплекс з двох дуже близьких за алгоритмом додатків. Програма «Шостий додаток» відкриває файл з даними і створює його проекцію в пам'яті, після чого запускає дочірній процес (програму «Сьомий додаток»). Дочірній процес отримує доступ до тієї ж проекції файлу. Після цього обидва процесу паралельно і незалежно працюють з одним і тим же файлом.

### Передача проекції файлу в інший процес

header.h

#include <Windows.h>

#define ID\_LIST 100

#define ID\_CHOICE 101

struct GOODS //Структура, що описує запису файлу

{

int Nmb;//Номер зразка

int Price;//Ціна зразка

};

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

void DlgOnCommand(HWND, int);

BOOL DlgOnlnitDialog();

resource.rc

#include "header.h"

#include <Windows.h>

Files DIALOG 41, 23, 84, 115

STYLE WS\_POPUP | WS\_CAPTION | WS\_SYSMENU

CAPTION "Шостий додаток"

{

LISTBOX ID\_LIST, 5, 4, 74, 88, WS\_VSCROLL | WS\_BORDER

PUSHBUTTON "Вибірка", ID\_CHOICE, 5, 98, 70, 14, BS\_PUSHBUTTON

}

main.cpp

#include "header.h"

HANDLE hFile;//Дескриптор файлу

GOODS \*ptr;//Покажчик на змінну типу GOODS

HANDLE hMem;//Дескриптор проекції файлу в пам'яті

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE hlnst, HINSTANCE, LPSTR, int)

{

DialogBoxW(hlnst, L"Files", nullptr, DlgProc);//Відкриємо модальний діалог

return 0;

}

/\*Віконна функція модального діалогу\*/

BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM IParam)

{

switch (msg)

{

case WM\_INITDIALOG: DlgOnlnitDialog();

case WM\_COMMAND: DlgOnCommand(hwnd, wParam);

default:

return FALSE;

}

}

//Функція обробки повідомлення WM\_INITDIALOG\* /

BOOL DlgOnlnitDialog()

{

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

hFile = CreateFileW(L"stock.dat", GENERIC\_READ, 0, //Откроем имеющийся файл

nullptr, OPEN\_EXISTING, 0, nullptr);

/\*Створення в пам'яті іменованого об'єкта "проекція файлу"\*/

hMem = CreateFileMappingW(hFile, nullptr, PAGE\_READONLY, 0, 0, L"MyFile");

/\*Відображення проекції файлу на адресного простору процесу\*/

ptr = (GOODS\*)MapViewOfFile(hMem, FILE\_MAP\_READ, 0, 0, 0);

/\*Створимо дочірній процес - "Сьомий додаток"\*/

CreateProcessW(L"Сьомий додаток.exe", nullptr, nullptr, nullptr, FALSE, 0, nullptr, nullptr, &si, &pi);

return TRUE;

}

/\*функція обробки повідомлень WM\_COMMAND від діалогу\*/

void DlgOnCommand(HWND hwnd, int id)

{

switch (id)

{

case ID\_CHOICE: //Натиснута кнопка "Вибірка"

{

wchar\_t szUnit[20]; //Поле для формування рядків списку

int numb = rand() % 15;//Отримаємо випадкове число

/\*Читаємо вибірково з проекції файлу і заповнюємо список\*/

wsprintfW(szUnit, L"%d %d", ptr[numb].Nmb, ptr[numb].Price);

SendDlgItemMessage(hwnd, ID\_LIST, LB\_ADDSTRING, 0, (LPARAM)szUnit);

break;

}

case IDCANCEL://Закриваємо діалог через системне меню

UnmapViewOfFile(ptr); //Звільнимо адресний простір процесу

CloseHandle(hMem);//Закриємо дескриптор проекції файлу

CloseHandle(hFile);//Закриємо дескриптор файлу

EndDialog(hwnd, 0);//Закриємо діалог

}

}

### Отримання проекції файлу з іншого процесу

header.h

#include <Windows.h>

#define ID\_LIST 100

#define ID\_CHOICE 101

struct GOODS //Структура, що описує запису файлу

{

int Nmb;//Номер зразка

int Price;//Ціна зразка

};

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

void DlgOnCommand(HWND, int);

BOOL DlgOnInitDialog();

resource.rc

#include "header.h"

#include <Windows.h>

Files DIALOG 128, 23, 84, 115 //Другие координаты для удобства

STYLE WS\_POPUP | WS\_CAPTION | WS\_SYSMENU

CAPTION "Сьомий додаток" //Другой заголовок для ясности

{

LISTBOX ID\_LIST, 5, 4, 74, 88, WS\_VSCROLL | WS\_BORDER

PUSHBUTTON "Вибірка", ID\_CHOICE, 5, 98, 70, 14, BS\_PUSHBUTTON

}

main.cpp

#include "header.h"

#include <time.h>

HANDLE hFile;//Дескриптор файлу

HANDLE hMem;//Дескриптор проекції файлу в пам'яті

GOODS \*ptr;//Покажчик на змінну типу GOODS

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE hlnst, HINSTANCE, LPSTR, int)

{

DialogBoxW(hlnst, L"Files", nullptr, DlgProc); //Откроем модальный диалог

return 0;

}

/\*Віконна функція модального діалогу\*/

BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

switch (msg)

{

case WM\_INITDIALOG: DlgOnInitDialog();

case WM\_COMMAND: DlgOnCommand(hwnd, wParam);

default:

return FALSE;

}

}

/\*функція обробки повідомлення WM\_INITDIALOG\*/

BOOL DlgOnInitDialog()

{

srand(time(0));//Ініціалізуємо датчик випадкових чисел випадковим числом

/\*Відображення наявної вже проекція файлу

на адресний простір процесу\*/

hMem = OpenFileMappingW(FILE\_MAP\_READ, FALSE, L"MyFile");

ptr = (GOODS\*)MapViewOfFile(hMem, FILE\_MAP\_READ, 0, 0, 0);

return TRUE;

}

/\*функція обробки повідомлень WM\_COMMAND від діалогу\*/

void DlgOnCommand(HWND hwnd, int id)

{

switch (id)

{

case ID\_CHOICE: //Натиснута кнопка "Вибірка"

{

wchar\_t szUnit[20];//Поле для формування рядків списку

int numb = rand() % 15;//Отримаємо випадкове число

/\*Читаємо вибірково з проекції файлу і заповнюємо список\*/

wsprintfW(szUnit, L"%d %d", ptr[numb].Nmb, ptr[numb].Price);

SendDlgItemMessage(hwnd, ID\_LIST, LB\_ADDSTRING, 0, (LPARAM)szUnit);

break;

}

case IDCANCEL://Закриваємо діалог через системне меню

UnmapViewOfFile(ptr);//Звільнимо адресний простір процесу

CloseHandle(hMem);//Закриємо дескриптор проекції файлу

EndDialog(hwnd, 0);//Закриємо діалог

}

}

Обидва додатки розглянутого програмного комплексу побудовані на основі модального діалогу. У програмі «Шостий додаток» в процесі ініціалізації діалогового вікна виконуються підготовчі дії для роботи з проекцією файлових даних: відкриття файлу *STOCK.DAT*, створення фізичної пам'яті об'єкта «проекція файл» і відображення її на віртуальний адресний простір процесу з поверненням покажчика *ptr* на виділений регіон пам'яті. Після цього викликом функції *CreateProcess()* створюється дочірній процес - запускається додаток «сьомий додаток».

Для того щоб об'єкт «проекція файлу» міг використовуватися іншим процесом, йому призначається ім'я («*MyFile*» в даному прикладі), яке повинно бути відомо процесу-приймача. Ім'я об'єкта є такою ж принциповою характеристикою об'єкта, як і його дескриптор (значення типу *HANDLE*).

*Дескриптор об'єкта* повертається функцією створення даного об'єкта і використовується надалі в тому ж процесі для взаємодії з об'єктом. Так, отриманий у прикладі дескриптор файлу *hFile* передається як параметр функції *CreateFileMapping()*, а отриманий з цієї функції дескриптор проекції файлу *hMem* далі використовується при виклику функції *MapViewOfFile()* відображення проекції на адресний простір процесу.

На відміну від дескриптора, ім'я об'єкта використовується головним чином для передачі інформації про об'єкт іншого процесу, який, працюючи у власному адресному просторі, не може знати значень змінних першого процесу. Таким чином, ім'я об'єкта, на відміну від його дескриптора, є характеристикою загальносистемної об'єкта, видимої не тільки в процесі-творця об'єкта, але і у всіх інших.

Функція *MapViewOfFile()* повертає покажчик на створений регіон з відображенням проекції файлу. Оскільки дані, складові файл, можуть мати саму різну організацію, функція повертає покажчик узагальненого типу *LPVOID*, який в кожному конкретному випадку слід явно перетворювати в потрібний тип. Наш приклад характерний в тому відношенні, що вміст файлу становлять записи структурного типу *GOODS*. Тому що повертається функцією *MapViewOfFile()* значення слід перетворити в вказівник на цю структуру:

*ptr = (GOODS\*)MapViewOfFile(hMem, FILE\_FILE\_MAP\_READ, 0, 0, 0);*

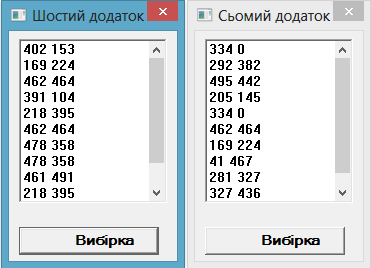
(покажчик *ptr* вже було оголошено раніше як *GOODS\**, тобто як покажчик на структуру *GOODS*).

Подальші дії в програмі «Шостий додаток» ініціюються натисканням кнопки «Вибірка». У цьому випадку віконна функція діалогового вікна передає управління на блок *case ID\_CHOICE* функції *DlgOnCommand()*. Тут викликом функції *C++ rand()* генерується випадкове число, використовуване далі як індекс запису у відображеному на пам'ять масиві даних. Запис, відповідна цим індексом, перетворюється в символьну форму і заноситься в елемент управління - список, наявний в діалоговому вікні.

При закритті діалогу (через системне меню або клацанням по кнопці «X» ) відбувається перехід на блок *case IDCANCEL*. Тут викликом відповідних функцій звільняється регіон адресного простору процесу, на який була відображена проекція файлу, далі закривається дескриптор самої проекції і, нарешті, закривається дескриптор дискового файлу. Всі ці дії в даному прикладі не обов'язкові, так як при завершенні програми система звільняє всі надані йому ресурси.

Необхідно мати на увазі, що, якщо деякий об'єкт використовується декількома процесами (як у нашому випадку об'єкт «проекція файлу»), він не буде знищений, поки не закриються всі його дескриптори. Тому, незважаючи на закриття дескриптора проекції файлу при завершенні програми «Шостий додаток», другий додаток, «сьомий додаток», буде продовжувати правильно працювати до свого завершення.

Друга складова нашого програмного комплексу, додаток «Сьомий додаток», відрізняється від першої лише парою деталей. Для того щоб у цьому додатку генерувалася інша послідовність випадкових чисел, функції ініціалізації діалогу виконується виклик функцій C++ *srand(time(0))*, які задають випадкове значення (залежне від поточного часу). Інша відмінність носить принциповий характер. Для отримання доступу до створеного вже об'єкту «проекція файлу» він відкривається повторно викликом функції OpenFileMapping() із зазначенням (що дуже важливо) від імені цього об'єкта. На рис. 5 наведено результат одного з сеансів роботи програмного комплексу.



Мал. 5. Робота програмного комплексу

## Обмін даними через сторінковий файл

Сторінковий файл входить у загальну фізичну пам'ять обчислювальної системи, помітно збільшуючи її обсяг. Ідея цього методу полягає в тому, що у фізичній пам'яті створюється проекція сторінкового файлу, що потім відображається на віртуальний адресний простір процесу. Оскільки іменовану проекцію файлу можуть використовувати інші процеси, попередньо відкривши її, створюється зручна можливість мати ділянку фізичної пам'яті, відображений на віртуальні простори декількох процесів (рис. 6). До речі, згідно документації Microsoft це єдиний спосіб передати фізичну пам'ять процесу в процес.

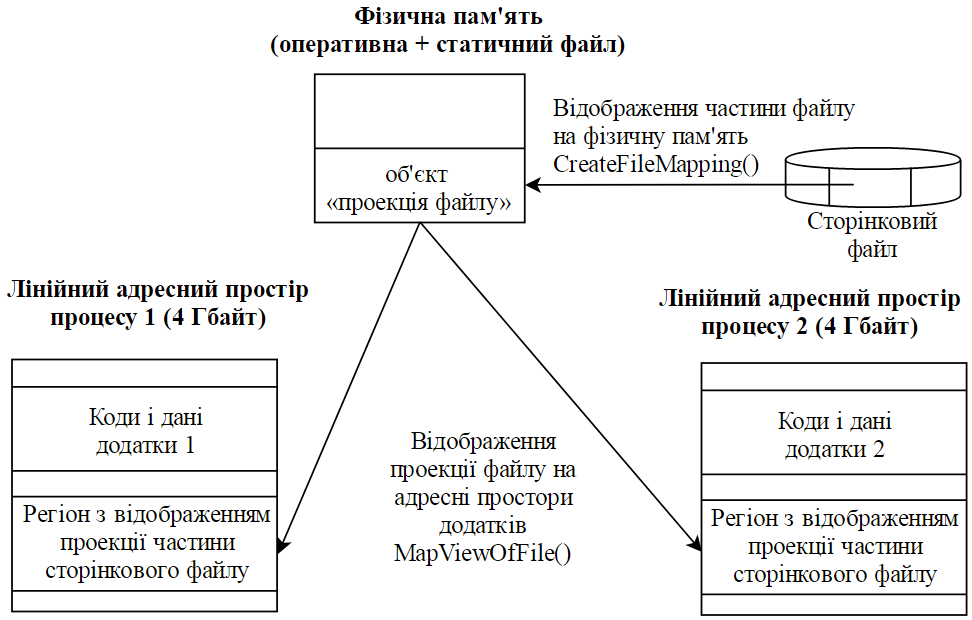


Рис. 6. Відображення частини сторінкового файлу на адресні простори двох процесів

Фізична пам'ять складається з оперативної пам'яті і сторінкового файлу. Виходить, що сторінковий файл відображає самого себе. Насправді створення об'єкта «проекція файлу» для сторінкового файлу означає лише те, що фізичну пам'ять для цього об'єкту слід виділити не з пойменованого файлу, а з сторінкового.

Розглянемо формальний приклад обміну даними через сторінковий файл. Нехай перший процес «Восьмий додаток», створивши в пам'яті проекцію сторінкового файлу і відобразивши її на свій адресний простір, записує в цю пам'ять деякі дані. Другий процес «Дев’ятий додаток», відобразивши ту ж проекцію сторінкового файлу на свій адресний простір, читає ці дані. Зрозуміло, в задачах такого роду необхідна синхронізація процесів, щоб другий процес не почав читати з пам'яті раніше, ніж перший процес запише туди щось корисне, але тут заради простоти ми обійдемося без синхронізації процесів.

### Обмін даними через сторінковий файл

main.cpp

#include <windows.h>

const int nSize = 0x4000000;//Розмір масиву даних в байтах = 64 Мбайт

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

/\*Створення в пам'яті об'єкта «проекція файлу» для 16 М цілих чисел\*/

wchar\_t szMapName[] = L"Mapping";//Довільне ім'я проекції

HANDLE hMem = CreateFileMapping((HANDLE)-1, nullptr, PAGE\_READWRITE, 0, nSize, szMapName);

/\*Відображення виділеної фізичної пам'яті на віртуальні адреси додатки\*/

int \*ptr = (PINT)MapViewOfFile(hMem, FILE\_MAP\_ALL\_ACCESS, 0, 0, nSize);

for(int i = 0; i < nSize / 4; i++)//Створимо тестові дані

ptr[i] = i;

/\*3апустим дочірній процес\*/

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

CreateProcessW(L"Дев’ятий додаток.exe", nullptr, nullptr, nullptr, FALSE, 0, nullptr, nullptr, &si, &pi);

/\*Виведемо вікно повідомлення і тим самим призупинимо програму\*/

wchar\_t szTxt[80];

wsprintfW(

szTxt,

L"Елементи з початку,\nсередини і кінця:\n0х%x\n0х%x\n0х%x",

ptr[0],

ptr[nSize / 8],

ptr[nSize / 4 - 1]

);

MessageBoxW(nullptr, szTxt, L"Запис", MB\_OK);

return 0;

}

### Читання даних з сторінкового файлу

main.cpp

#include <windows.h>

const int nSize = 0x4000000;//Розмір масиву даних в байтах = 64 Мбайт

/\*Головна функція WinMain\*/

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

wchar\_t lpszName[] = L"Mapping";//Довільне ім'я проекції

/\*Відкриємо вже створену проекцію і відобразимо її на адреси додатки\*/

HANDLE hMem = OpenFileMappingW(PAGE\_READWRITE, FALSE, lpszName);

int \*ptr = (PINT)MapViewOfFile(hMem, FILE\_MAP\_ALL\_ACCESS, 0, 0, nSize);

/\*Виведемо вікно повідомлення для контролю даних\*/

wchar\_t szTxt[80]{0};

wsprintfW(

szTxt,

L"Останні елементи:\n0x%x\n0x%x\n0x%x",

ptr[nSize / 4 - 3],

ptr[nSize / 4 - 2],

ptr[nSize / 4 - 1]

);

MessageBoxW(NULL, szTxt, L"Читання", MB\_OK);

return 0;

}

При створенні проекції файлу в попередньому прикладі ми в якості першого параметра функції CreateFileMapping() вказували дескриптор відкритого дискового файлу. Для того щоб створити проекцію сторінкового файлу, перший параметр цієї функції повинен бути дорівнює -1 (і мати до того ж тип HANDLE). Передостанній параметр цієї функції задає розмір проекції, а останній - довільне ім'я об'єкта, який, очевидно, буде використано і в другій програмі комплексу. Для переконливості приклад розмір створюваного масиву даних і, відповідно, проекції файлу для його зберігання заданий досить великим - 16 М цілих чисел, тобто 64 Мбайт.

За допомогою функції MapViewOfFile() в віртуальної пам'яті процесу виділяється регіон того ж розміру (останній параметр цієї функції), і проекція сторінкового файлу відображається на цей регіон. Як ми вже бачили в попередньому прикладі, функція MapViewOfFile() повертає покажчик на даний узагальненого типу LPVOID, який ми перетворимо в тип "покажчик на ціле" (PINT).

Виділена фізична пам'ять заповнюється тестовими даними (натуральний ряд чисел від 0 до OxFFFFFF), після чого створюється дочірній процес «Дев’ятий додаток». Для контролю створеного тестового масиву, а також для запобігання передчасного завершення програми на екран виводиться вікно повідомлення з трьома елементами масиву (першим, середнім і останнім).

В програмі «Восьмий додаток» відкривається проекція файлу з відомим ім'ям, виконується її відображення на віртуальний простір процесу і читаються 3 останніх елемента масиву з виведенням результату у вікно повідомлення.

На рис. 13.7 показаний результат виконання програмного комплексу.

рис

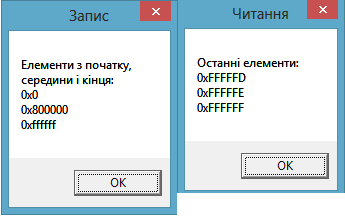


Рис. 7. Використання сторінкового файлу для межпроцессной передачі даних

# Передача даних

## Способи передачі даних між процесами

Під обміном даними між паралельними процесами розуміється пересилання даних від одного потоку до іншого потоку, припускаючи, що ці потоки виконуються в контекстах різних процесів. Потік, який посилає дані іншому потоку, називається відправником. Потік, який отримує дані від іншого потоку, називається адресатом або одержувачем.

Якщо потоки виконуються в одному процесі, то для обміну даними між ними можна використовувати глобальні змінні і засоби синхронізації потоків. Справа йде складніше у тому випадку, якщо потоки виконуються в різних процесах — потоки не можуть звертатися до загальних змінних і для обміну даними між ними існують спеціальні засоби операційної системи. Якщо говорити концептуально, то для обміну даними між процесами створюється канал передачі даних, організація якого схематично показана на рис. 8.

Коротко пояснимо пристрій і роботу каналу передачі даних. Канал даних включає вхідний і вихідний буфер пам'яті, потоки ядра операційної системи та загальну пам'ять, доступ до якої мають обидва потоку ядра. Працює канал передачі даних наступним чином:

* перший потік ядра операційної системи читає дані з вхідного буфера В1 і записує їх у загальну пам'ять М;
* другий потік ядра читає дані з загальної пам'яті М і записує їх в буфер В2;

Користувальницькі потоки Т1 і Т2 допомогою виклику функцій ядра операційної системи мають доступ до буферам В1 та В2 відповідно. Тому пересилання даних з потоку Т1 в потік Т2 відбувається наступним чином:

* користувальницький потік Т1 записує дані в буфер В1, використовуючи спеціальну функцію ядра операційної системи;
* потік K1 ядра операційної системи читає дані з буфера B1 і записує їх у загальну пам'ять M;
* потік К2 ядра операційної системи читає дані з загальної пам'яті М і записує їх в буфер B2;
* користувальницький потік Т2 читає дані з буфера В2.

канали передачі даних

B1

B2

M

T1,T2 — користувальницькі потоки

B1, B2 — буфери

K1, K2 — потоки ядра операційної системи

М — загальна пам'ять

Рис. 8. Схема каналу передачі даних

Звідси видно, що в будь-якому випадку обмін даними може бути організований тільки через ланцюжок взаємодіючих потоків, які обмінюються між собою даними через загальну, тільки для них, пам'ять.

На схему каналу, показану на рис. 8, можна поглянути і ширше. Вона також підходить для організації каналу передачі даних по мережі. Тільки в цьому випадку загальна пам'ять M, може розглядатися як передавальне середовище, пристрій якої аналогічно пристрою каналу передачі даних.

Тепер скажемо про те, як організувати канал передачі даних між процесами програмним чином. Для цього потрібно потоки ядра операційної системи та загальну пам'ять, яка використовується для обміну даними, замінити файлом. І ми одержимо найпростіший канал передачі даних між потоками, який виконується в контекстах різних процесів. Таким чином, обмін даними між процесами через загальний файл являє собою організацію найпростішого каналу передачі даних між процесами. Іноді операційна система може спростити доступ до поділюваного файла, щоб прискорити обмін даними. Наприклад, операційні системи Windows для цих цілей можуть відображати, або іншими словами, проектувати файл на адресний простір процесу.

При обміні даними між паралельними процесами розрізняють два способи передачі даних:

* потоком;
* повідомленням.

Якщо дані передаються безперервною послідовністю байтів, то така пересилка даних називається передача даних потоком. У цьому випадку загальна пам'ять М, доступна потокам ядра операційної системи, може бути відсутньою, а пересилання даних виконуватиметься одним потоком ядра безпосередньо з буфера В1 в буфер В2.

Якщо дані пересилаються групами байтів, то така група байтів, називається повідомленням, а сама пересилання даних називається передачею даних повідомленнями.

## Зв'язки між процесами

Перш ніж передавати дані між процесами, треба встановити зв'язок між цими процесами. Зв'язок між процесами може встановлюватися як на фізичному (або апаратному), так і на логічному (або програмному) рівнях. З точки зору напряму передачі даних розрізняють наступні види зв'язків:

* напівдуплексний зв'язок, тобто дані за цією зв'язку можуть передаватися тільки в одному напрямку;
* дуплексний зв'язок, тобто дані за цією зв'язку можуть передаватися в обох напрямках.

Тепер, припускаючи, що розглядаються тільки напівдуплексні зв’язки, визначимо можливі топології зв'язків. Під топологією зв'язку будемо розуміти конфігурацію зв'язків між процесами-відправниками та адресатами. З точки зору топології розрізняють наступні види зв'язків:

* напівдуплексний зв'язок, тобто дані за цією зв'язку можуть передаватися тільки в одному напрямку;
* дуплексний зв'язок, тобто дані за цією зв'язку можуть передаватися в обох напрямках.

Тепер, припускаючи, що розглядаються тільки полудуплексные зв'язку, визначимо можливі топології зв'язків. Під топологією зв'язку будемо розуміти конфігурацію зв'язків між процесами-відправниками та адресатами. З точки зору топології розрізняють наступні види зв'язків:

* 1 → 1 — між собою пов'язані тільки два процесу;
* 1 → N — один процес пов'язаний з N процесами;
* N → 1 — кожен з N процесів пов'язаний з одним процесом;
* N → M — кожен з N процесів пов'язаний з кожним з М процесів.

Ці види зв'язків схематично показані на рис. 9, де процеси зображені колами, зв'язки дугами, а прямокутники позначають засоби передачі даних. Тут потрібно було б сказати-не засоби, а канали передачі даних, але для назви каналів передачі даних, які забезпечують різні види зв'язків, використовуються спеціальні терміни. Причому ці терміни можуть відрізнятися в різних системах і, крім того, часто один канал передачі даних може підтримувати різні види зв'язків між процесами.

При розробці систем з обміном даних між процесами, перш за все, повинна бути обрана топологія зв'язків і напрямку передачі даних по цих зв'язків. Після цього в програмах реалізуються вибрані зв'язки між процесами з використанням спеціальних функцій операційної системи, які призначені для встановлення зв'язку між процесами. Ці функції, а також функції для обміну даними між процесами забезпечує система передачі даних, яка зазвичай є частиною ядра операційної системи.

а) 1 — 1

б) 1 — N

в) N — 1

г) N — M

Рис. 9. Топологія зв'язків між процесами

## Передача повідомлень

Концептуально обмін повідомленнями між процесами виконується за допомогою двох функцій:

* send — надсилання повідомлення;
* receive — отримати повідомлення.

Саме повідомлення складається з двох частин: заголовка і тіла повідомлення. У заголовку повідомлення знаходиться така службова інформація, як:

* тип повідомлення;
* ім'я адресата повідомлення;
* ім'я відправника повідомлення;
* довжина повідомлення (контрольна інформація).

Тіло повідомлення містить саме повідомлення.

При передачі повідомлень може використовуватися пряма або непряма адресація процесів. При прямій адресації процесів у функціях send і receive явно вказуються процеси відправник і адресат. У цьому випадку функції обміну даними мають наступний вигляд:

send(Process Р, повідомлення); // відіслати повідомлення процесу Р

receive(Process Q, повідомлення); // отримати повідомлення від процесу Q

При непрямій адресації функції send і receive вказуються не адреси, а ім'я зв'язку, по якому передається повідомлення. У цьому випадку функції обміну даними мають наступний вигляд:

send(Connection S, повідомлення); // надіслати повідомлення на зв'язку S

receive(Connection R, повідомлення); // отримати повідомлення по зв'язку R

Адресація процесів може бути симетричною і асиметричною. Якщо при обміні повідомленнями між процесами використовується тільки пряма або тільки непряма адресація, то така адресація процесів називається симетричною. Якщо ж при обміні повідомленнями між процесами використовується як пряма, так і непряма адресація, то така адресація процесів називається асиметричною. Асиметрична адресація процесів використовується в системах "клієнт-сервер". В цьому випадку клієнти знають адресу сервера та надсилають йому повідомлення, використовуючи функцію:

send(Process Сервер, повідомлення);

Сервер «слухає» канал зв'язку і приймає повідомлення від усіх клієнтів, використовуючи функцію:

receive(Connection S, повідомлення);

Часто ця функція так і називається listen (слухати).

На закінчення цього розділу дамо важливе визначення. Набір правил, за якими встановлюються зв'язки і передаються дані між процесами, називається протоколом. Можна трохи пояснити, в чому полягають правила протоколу з точки зору програмування передачі даних. Ці правила включають набір процедур або функцій, які використовуються для передачі даних та порядку використання цих функцій. Тобто при використанні функцій потрібно враховувати певну послідовність їх викликів. Наприклад, не можна починати передачу даних, не встановивши зв'язок між відправником і адресатом.

## Синхронний і асинхронний обмін даними

При передачі даних розрізняють синхронний і асинхронний обмін даними. Якщо потік-відправник, відправивши повідомлення функцією send, блокується до цього повідомлення потоком-адресатом, то таке відправлення повідомлення називається синхронним. В іншому випадку відправлення повідомлення називається асинхронним. Якщо потік-адресат, викликав функцію receive, блокується до тих пір, поки не отримає повідомлення, то таке повідомлення називається синхронним. В іншому випадку отримання повідомлення називається асинхронним.

Обмін повідомленнями називається синхронним, якщо потік-відправник синхронно передає повідомлення, а потік-адресат синхронно приймає ці повідомлення. В іншому випадку обмін повідомленнями називається асинхронним. Передбачається, що обмін даними потоком завжди відбувається синхронним чином, оскільки в цьому випадку між відправником і адресатом встановлюється безпосередній зв'язок.

Синхронний обмін даними в разі прямої адресації процесів називається рандеву (rendezvous, франц. "зустріч"). Такий механізм обміну повідомленнями використовується в мові програмування Ада.

Враховуючи можливості синхронного обміну даними, цей механізм може використовуватися для вирішення завдань синхронізації процесів. При цьому слід зазначити, що в операційних системах Windows синхронізація процесів, що виконуються на різних комп'ютерах в локальній мережі, може здійснюватися тільки за допомогою механізму синхронного обміну даними.

## Буферизація

Буфером називається місткість зв'язку між процесами, тобто кількість повідомлень, які можуть одночасно пересилатися по цьому зв'язку. Розрізняються три типи буферизації:

* нульова місткість зв'язку (немає буфера) — в цьому випадку можливий тільки синхронний обмін даними між процесами;
* обмежена місткість зв'язку (обмежений буфер) — в цьому випадку, якщо буфер повний, то відправник повідомлення повинен чекати очищення буфера хоча б від одного повідомлення;
* необмежена місткість зв'язку (необмежений буфер) — у цьому випадку відправник ніколи не чекає при надсиланні повідомлення.

Як видно з цих визначень, типи буферизації тісно пов'язані з синхронізацією передачі даних, тому також повинні враховуватися при розробці систем, які містять обмін даними між процесами.

# Робота з анонімними каналами в Windows

Так як робота з анонімними каналами вимагає спільного використання цілого ряду функцій, то даний розділ організований наступним чином. Спочатку дано визначення анонімних каналів та розглянуто всі функції, які призначені для роботи з ними. Потім наведено приклади, що ілюструють використання цих функцій. В останньому розділі показано, як за допомогою анонімних каналів можна перенаправити стандартний ввід-висновок.

## Анонімні канали

Анонімним каналом називається об'єкт ядра операційної системи, який забезпечує передачу даних між процесами, що виконуються на одному комп'ютері. Процес, який створює анонімний канал, називається сервером анонімного каналу. Процеси, які зв'язуються з анонімним каналом, називаються клієнтами анонімного каналу. Іншими словами можна сказати, що анонімний — це такий канал передачі даних між процесами, який не має імені. Отже, доступ до такого каналу мають тільки батьківський процес-сервер і дочірні процеси-клієнти цього каналу.

Перерахуємо характеристики анонімних каналів, які необхідно враховувати при їх використанні для обміну даними між паралельними процесами:

* не мають імені;
* напівдуплексні;
* передача даних потоком;
* синхронний обмін даними;
* можливість моделювання будь-якої топології зв'язків.

Трохи пояснимо характеристику напряму передачі даних по анонімному каналу. Так як анонімний канал напівдуплексний, то може скластися враження, що з цього каналу можна передавати дані від процесу до процесу лише в одному напрямку. Насправді це не так. Так, дійсно, за анонімним каналу можна передавати дані тільки в одному напрямку. Але кожен кінець анонімного каналу має свій дескриптор, який можна передати кожному дочірньому процесу. Тому кожен процес може як записувати дані в анонімний канал, так і читати дані звідти.

Зауваження

В операційних системах Windows NT анонімний канал реалізований за допомогою іменованого каналу з унікальним ім'ям. Звідси випливає, що у цих операційних системах функції, які працюють з іменованими каналами, можуть працювати також і з анонімними каналами.

Тепер наведемо порядок роботи з анонімними каналами, якого і будемо дотримуватися в подальшому:

* створення анонімного каналу сервером;
* з'єднання клієнтів з каналом;
* обмін даними по каналу;
* закриття каналу.

Докладно ці пункти роботи з анонімними каналами будуть розглянуті в наступних розділах цієї глави.

## Створення каналів анонімних

Анонімні канали створюються процесом-сервером за допомогою функції CreatePipe, яка має наступний прототип:

BOOL CreatePipe(

PHANDLE hReadHandle, // дескриптор для читання з каналу

PHANDLE hWriteHandle, // дескриптор для запису в канал

LPSECURITY\_ATTRIBUTES IpPipeAttributes, // атрибути захисту

DWORD dwSize // розмір буфера в байтах

);

При вдалому завершенні функція *CreatePipe* повертає ненульове значення, а у випадку невдачі — *false*. Розглянемо коротко призначення параметрів цієї функції.

У разі успішного завершення функція *CreatePipe* створює два дескриптора анонімного каналу — один для читання даних з каналу, а другий для запису даних в канал. Ці дескриптори повертаються в змінних, на які вказують параметри *hReadPipe* і *hWritePipe*. Дескриптор, на який вказує параметр *hReadPipe*, в подальшому використовується у функціях читання даних з каналу, а дескриптор *hWritePipe* — у функціях запису даних в канал. Параметр *lpPipeAttributes* визначає атрибути захисту анонімного каналу. Встановлення значення цього параметра *lpPipeAttributes* ми розглянемо в наступному розділі. Параметр *dwSize* визначає розмір буфера вводу-виводу анонімного каналу. Зазначимо, що операційні системи Windows автоматично визначають розмір буфера і тому значення параметра *dwSize* є тільки побажанням операційній системі при виборі розміру буфера. Значення цього параметра можна встановити рівним *0*, тоді операційна система вибере розмір буфера за замовчуванням.

## З'єднання клієнтів з анонімним каналом

Так як анонімні канали не мають імені, то для з'єднання процесу-клієнта з таким каналом необхідно передати йому один з дескрипторів анонімного каналу. При цьому дескриптор що передається повинен бути наслідуваним, а сам процес-клієнт повинен бути дочірнім процесом процесу сервера анонімного каналу і успадковувати успадковані дескриптори процесу-сервера.

Успадкування дескрипторів анонімного каналу визначається значенням поля *bInheritHandle* у структурі типу *SECURITY\_ATTRIBUTES*, на який вказує параметр *lpPipeAttributes* функції *CreatePipe*. Якщо значення цього поля, яке має тип *bool*, рівне *true*, то дескриптори анонімного каналу створюються успадкованими, в іншому випадку — дескриптори не успадковується. Якщо процесу-клієнту передаються два дескриптора анонімного каналу, то природно, що обидва вони повинні бути зроблені успадкованими при створенні анонімного каналу процесом-сервером. Якщо ж процесу-клієнту передається тільки один з дескрипторів, то можливі два варіанти дій. Або створити успадковані дескриптори анонімного каналу, а потім той дескриптор, який не передається клієнту, зробити не успадкованим. Або, навпаки, створити дескриптори що успадковує анонімного каналу, а потім той дескриптор, який передається клієнту, зробити наслідуваним. В операційній системі Windows 98 обидві ці завдання вирішуються шляхом створення дескриптора що не успадковується або успадковує дубліката вихідного дескриптора, використовуючи функцію *DuplicateHandle*. Після цього вихідний дескриптор закривається. В операційній системі Windows 2000 ця задача може бути вирішена за допомогою функції *SetHandleInformation*, яка змінює властивість спадкування дескриптора. Для того щоб процес-клієнт успадковував дескриптори анонімного каналу він повинен бути створений функцією *CreateProcess* в процесі-сервері анонімного каналу і параметр *bInheritHandles* цієї функції повинен бути встановлений в *true*.

Явна передача спадкового дескриптора процесу-клієнту анонімного каналу може виконуватись одним з наступних способів:

* через командний рядок;
* через поля *hStdInput*, *hStdOutput* і *hStdError* структури *STARTUPINFO*;
* за допомогою повідомлення *WM\_COPYDATA*;
* через файл.

В даній главі ми будемо використовувати тільки перші два способи передачі дескрипторів процесу-клієнту, що найбільш природно для анонімних каналів.

## Обмін даними по анонімному каналу

Для обміну даними по анонімному каналу в операційних системах Windows використовуються ті ж функції, що для запису/читання даних у файл.

Для запису даних у анонімний канал використовується функція *WriteFile*, яка має наступний прототип:

BOOL WINAPI WriteFile(

\_In\_        HANDLE       hFile, //дескриптор анонимного канала

\_In\_        LPCVOID      lpBuffer, //буфер данных

\_In\_        DWORD        nNumberOfBytesToWrite, //количество байтов для записи

\_Out\_opt\_   LPDWORD      lpNumberOfBytesWritten, //количество записанных байтов

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped //асинхронный ввод

);

Функція *WriteFile* записує в анонімний канал кількість байтів, заданих параметром *nNumberOfBytesToWrite*, з буфера даних, на який вказує параметр *lpBuffer*. Дескриптор виведення цього анонімного каналу повинен бути поставлене першим параметром функції *WriteFile*. При успішному завершенні функція *WriteFile* повертає ненульове значення, а у випадку невдачі — *false*. Кількість байт, записаних функцією *WriteFile* у анонімний канал, повертається в змінній, на яку вказує параметр *nNumberOfBytesToWrite*. Параметр *lpOverlapped* призначений для виконання асинхронної операції виводу, так як анонімні канали піддержують тільки синхронну передачу даних, то в нашому випадку цей параметр завжди буде дорівнювати *NULL*.

Для читання даних з анонімного каналу використовується функція ReadFile, яка має наступний прототип:

BOOL WINAPI ReadFile(

\_In\_        HANDLE       hFile, //дескриптор анонімного каналу

\_Out\_       LPVOID       lpBuffer, //буфер даних

\_In\_        DWORD        nNumberOfBytesToRead, //кількість байт для запису

\_Out\_opt\_   LPDWORD      lpNumberOfBytesRead, //кількість записаних байтів

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped //асинхронний ввід

);

Функція *ReadFile* читає з анонімного каналу кількість байт, заданих параметром *nNumberOfBytesToRead*, В буфер даних, на який вказує параметр *lpBuffer*. Дескриптор введення цього анонімного каналу повинен бути поставлене першим параметром функції *ReadFile*. При успішному завершенні функція *ReadFile* повертає ненульове значення, а у випадку невдачі — *false*. Кількість байт, Прочитаних функцією *WriteFile* з анонімного каналу, повертається в змінній, на яку вказує параметр *nNumberOfBytesToRead*. Також як і у випадку запису в анонімний канал параметр *lpOverlapped* повинен дорівнювати *NULL*. Зазначимо, що обмін даними по анонімному каналу здійснюється тільки у відповідності з призначенням дескриптора цього каналу. Дескриптор для запису в анонімний канал повинен бути параметром функції *WriteFile*, а дескриптор для читання з анонімного каналу повинен бути параметром функції *ReadFile*. В цьому і полягає сенс передачі даних по анонімному каналу тільки в одному напрямку. Однак це не означає, що один процес може використовувати анонімний канал тільки для запису або тільки для читання. Один і той же процес може як писати в анонімний канал, так і читати дані з нього, належним чином використовуючи дескриптори цього каналу.

На завершення цього розділу відзначимо, що після завершення обміну даними по анонімному каналу потоки повинні закрити дескриптори запису і читання анонімного каналу, використовуючи функцію CloseHandle.

## Приклади роботи з анонімними каналами

Спочатку розглянемо приклад, в якому процес-сервер створює анонімний канал і дочірній процес, якому передає один з дескрипторів цього анонімного каналу. Для передачі дескриптор використовується командний рядок і, в цьому випадку, дочірній процес є клієнтом анонімного каналу. Для визначеності передамо клієнту дескриптор для запису в анонімний канал і залишимо сервера дескриптор для читання.

У лістингу наведена програма процесу-клієнта анонімного каналу.

### Програма процесу-клієнта анонімного каналу

#include <windows.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

HANDLE hWritePipe;

// перетворимо символьне уявлення дескриптора в число

hWritePipe = (HANDLE)atoi(argv[1]);

// чекаємо команди про початок запису в анонімний канал

printf("Press any key to start communication.\n");

getchar();

// пишемо в анонімний канал

for (int i = 0; i < 10; i++)

{

DWORD dwBytesWritten;

if (

!WriteFile

(

hWritePipe,

&i,

sizeof(i),

&dwBytesWritten,

nullptr

)

)

{

printf("Write to file failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

printf("The number %d is written to the pipe.\n", i);

Sleep(500);

}

// закриваємо дескриптор каналу

CloseHandle(hWritePipe);

printf("The process finished writing to the pipe.\n");

printf("Press any key to exit.\n");

getchar();

return 0;

}

В наступному лістингу наведена програма процесу-сервера анонімного каналу, який запускає клієнта і передає йому дескриптор запису в анонімний канал через командний рядок.

### Програма процесу-сервера анонімного каналу

#include <windows.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

wchar\_t lpszComLine[500]; // для командного рядка

STARTUPINFO si;

PROCESS\_INFORMATION pi;

HANDLE hWritePipe, hReadPipe, hInheritWritePipe;

// створюємо анонімний канал

if (

!CreatePipe(

&hReadPipe, // дескриптор для читання

&hWritePipe, // дескриптор для запису

nullptr, // атрибути захисту за замовчуванням, в цьому випадку

// дескриптори hReadPipe і hWritePipe не успадковуються

0 // розмір буфера за замовчуванням

)

)

{

printf("Create pipe failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// робимо наслідуваний дублікат дескриптора hWritePipe

if (

!DuplicateHandle(

GetCurrentProcess(), // дескриптор поточного процесу

hWritePipe, // вихідний дескриптор каналу

GetCurrentProcess(), // дескриптор поточного процесу

&hInheritWritePipe, // новий дескриптор каналу

0, // цей параметр ігнорується

true, // новий дескриптор наслідуваний

DUPLICATE\_SAME\_ACCESS // доступ не змінюємо

)

)

{

printf("Duplicate handle failed %d.\n", GetLastError());

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// закриваємо непотрібний дескриптор

CloseHandle(hWritePipe);

// встановлюємо атрибути нового процесу

ZeroMemory(&si, sizeof(STARTUPINFO));

si.cb = sizeof(STARTUPINFO);

// формуємо командний рядок

wsprintfW(lpszComLine, L" десятий додаток.exe %d", (int)hInheritWritePipe);

// запускаємо новий консольний процес

if (

!CreateProcess(

nullptr,// ім'я процесу

lpszComLine, // командний рядок

nullptr,// атрибути захисту процесу за замовчуванням

nullptr,// атрибути захисту первинного потоку за замовчуванням

true,// успадковані дескриптори поточного процесу

// успадковуються новим процесом

CREATE\_NEW\_CONSOLE, // нова консоль

nullptr, // використовуємо середовище оточення процесу-предка

nullptr, // поточний диск і каталог, як і в процесі-предка

&si, // вигляд головного вікна - за замовчуванням

&pi // тут будуть дескриптори і ідентифікатори

// нового процесу і його первинного потоку

)

)

{

printf("Create process failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// закриваємо дескриптори нового процесу

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

// закриваємо непотрібний дескриптор каналу

CloseHandle(hInheritWritePipe);

// читаємо з анонімного каналу

for (int i = 0; i < 10; i++)

{

int nData;

DWORD dwBytesRead;

if (

!ReadFile(

hReadPipe,

&nData,

sizeof(nData),

&dwBytesRead,

nullptr

)

)

{

printf("Read from the pipe failed.\n");

printf("Press any key to finish\n");

getchar();

return GetLastError();

}

printf("The number %d is read from the pipe\n", nData);

}

// закриваємо дескриптор каналу

CloseHandle(hReadPipe);

printf("The process finished reading from the pipe\n");

printf("Press any key to exit \n");

getchar();

return 0;

}

У наведених лістингах (першому і другому), показується, як організувати двосторонній обмін даними по анонімному з'єднання між клієнтом і сервером. Для цього дескриптори читання і запису анонімного каналу використовуються як сервером, так і клієнтом цього анонімного каналу.

У наступних лістингах спочатку наведена програма процесу-клієнта анонімного каналу, а потім програма процесу-сервера анонімного каналу. Як і в лістингах попередніх, дескриптори анонімного каналу передаються через командний рядок.

### Процес-клієнт анонімного каналу

#include <windows.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

HANDLE hWritePipe, hReadPipe;

HANDLE hEnableRead; // для синхронізації обміну даними

wchar\_t lpszEnableRead[] = L"EnableRead";

// відкриваємо подія, що дозволяє читання

hEnableRead = OpenEvent(EVENT\_ALL\_ACCESS, FALSE, lpszEnableRead);

// перетворимо символьне уявлення дескрипторів в число

hWritePipe = (HANDLE)atoi(argv[1]);

hReadPipe = (HANDLE)atoi(argv[2]);

// чекаємо команди про початок запису в анонімний канал

printf("Press any key to start communication.\n");

getchar();

// пишемо в анонімний канал

for (int i = 0; i < 10; i++)

{

DWORD dwBytesWritten;

if (

!WriteFile(

hWritePipe,

&i,

sizeof(i),

&dwBytesWritten,

nullptr

)

)

{

printf("Write to file failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

printf("The number %d is written to the pipe.\n", i);

}

printf("The process finished writing to the pipe.\n");

// чекаємо дозволу на читання

WaitForSingleObject(hEnableRead, INFINITE);

// читаємо відповідь з анонімного каналу

for (int j = 0; j < 10; j++)

{

int nData;

DWORD dwBytesRead;

if (

!ReadFile(

hReadPipe,

&nData,

sizeof(nData),

&dwBytesRead,

nullptr

)

)

{

printf("Read from the pipe failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

printf("The number %d is read from the pipe.\n", nData);

}

printf("The process finished reading from the pipe.\n");

printf("Press any key to exit.\n");

getchar();

// закриваємо дескриптори каналу

CloseHandle(hWritePipe);

CloseHandle(hReadPipe);

CloseHandle(hEnableRead);

return 0;

}

В наступному лістингу наведена програма процесу-сервера анонімного каналу, який запускає клієнта і передає йому дескриптори анонімного каналу через командний рядок.

### Програма процес-сервера анонімного каналу

#include <windows.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

wchar\_t lpszComLine[500]; // для командного рядка

HANDLE hEnableRead; // для синхронізації обміну даними

wchar\_t IpszEnableRead[] = L"EnableRead";

STARTUPINFO si;

PROCESS\_INFORMATION pi;

HANDLE hWritePipe, hReadPipe;

SECURITY\_ATTRIBUTES sa;

// створюємо подію для синхронізації обміну даними

hEnableRead = CreateEvent(nullptr, FALSE, FALSE, IpszEnableRead);

// встановлює атрибути захисту каналу

sa.nLength = sizeof(SECURITY\_ATTRIBUTES);

sa.lpSecurityDescriptor = nullptr; // захист за замовчуванням

sa.bInheritHandle = TRUE; // дескриптори успадковані

// створюємо анонімний канал

if (

!CreatePipe(

&hReadPipe, // дескриптор для читання

&hWritePipe, // дескриптор для запису

&sa, // атрибути захисту за замовчуванням, дескриптори успадковані

0// розмір буфера за замовчуванням

)

)

{

printf("Create pipe failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// встановлюємо атрибути нового процесу

ZeroMemory(&si, sizeof(STARTUPINFO));

si.cb = sizeof(STARTUPINFO);

// формуємо командний рядок

wsprintfW(

lpszComLine,

L"дванадцятий додаток %d %d",

(int)hWritePipe,

(int)hReadPipe

);

// запускаємо новий консольний процес

if (

!CreateProcess(

nullptr, // ім'я процесу

lpszComLine, // командний рядок

nullptr, // атрибути захисту процесу за замовчуванням

nullptr, // атрибути захисту первинного потоку за замовчуванням

TRUE, // успадковані дескриптори поточного процесу

// успадковуються новим процесом

CREATE\_NEW\_CONSOLE, // нова консоль

nullptr, // використовуємо середовище оточення процесу-предка

nullptr, // поточний диск і каталог, як і в процесі-предка

&si, // вигляд головного вікна - за замовчуванням

&pi // тут будуть дескриптори і ідентифікатори

// нового процесу і його первинного потоку

)

)

{

printf("Create process failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// закриваємо дескриптори нового процесу

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

// читаємо з анонімного каналу

for (int i = 0; i < 10; i++)

{

int nData;

DWORD dwBytesRead;

if (

!ReadFile(

hReadPipe,

&nData,

sizeof(nData),

&dwBytesRead,

nullptr

)

)

{

printf("Read from the pipe failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

printf("The number %d is read from the pipe.\n", nData);

}

printf("The process finished reading from the pipe.\n");

// даємо сигнал на дозвіл читання клієнтом

SetEvent(hEnableRead);

// пишемо відповідь на анонімний канал

for (int j = 10; j < 20; j++)

{

DWORD dwBytesWritten;

if (

!WriteFile(

hWritePipe,

&j,

sizeof(j),

&dwBytesWritten,

nullptr

)

)

{

printf("Write to file failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

printf("The number %d is written to the pipe.\n", j);

}

// закриваємо дескриптори каналу

CloseHandle(hReadPipe);

CloseHandle(hWritePipe);

CloseHandle(hEnableRead);

printf("The process finished writing to the pipe.\n");

printf("Press any key to exit.\n");

getchar();

return 0;

}

Відзначимо що в цьому лістингу наступний момент: для організації двостороннього обміну даними по анонімному каналу сервер та клієнти каналу повинні синхронізувати доступ до цього каналу. Тобто для організації передачі даних необхідно розробити протокол передачі даних або використовувати об'єкти синхронізації, які виключають одночасний неконтрольований доступ паралельних потоків до анонімного каналу. У наведеному лістингу подія *hEnableRead* сигналізує клієнту про те, що сервер закінчив читання даних і тепер дані з каналу може читати клієнт. При відсутності такої синхронізації можливо одночасне читання даних сервером і клієнтом, так як вони працюють паралельно, і це викличе неправильну роботу програми і її зависання.

## Перенаправлення стандартного вводу-виводу

Анонімні канали часто використовуються для перенаправлення стандартного вводу-виводу. Щоб детальніше розібратися з цим питанням, спочатку стисло розглянемо стандартні засоби введення-виведення, що використовуються у мові програмування C++. Компілятор мови програмування C++ фірми Microsoft містить стандартну бібліотеку, яка підтримує три варіанти функцій стандартного вводу-виводу. Ці функції описуються в заголовних файлах *stdio.h*, *iostream.h* і *conio.h*.

Функції, які описані у файлі stdio.h, забезпечують введення-виведення в наступні стандартні файли:

* stdin — стандартний файл вводу;
* stdout — стандартний файл виводу;
* stderr — файл виводу повідомлень про помилки.

Ці функції складають стандартну бібліотеку введення-виведення мови програмування С.

Функції і оператори, які описані в заголовку *iostream.h*, забезпечують введення-виведення в стандартні потоки. Ці функції складають стандартну бібліотеку введення-виведення в мові програмування С++. При створенні консольного процесу або при розподілі консолі за допомогою функції *AllocConsole* стандартні файли, стандартні потоки вводу-виводу зв'язуються з дескрипторами, які задані в полях *hStdInput*, *hStdOutput* і *hStdError* структури типу *STARTUPINFO*. Тому якщо в ці поля будуть записані відповідні дескриптори анонімного каналу, то для передачі даних по анонімному каналу можна використовувати функції стандартного вводу-виводу. Така процедура називається перенаправленням стандартного вводу-виводу. Функції вводу-виводу з заголовкового файлу *conio.h* відрізняються від функцій введення-виведення з заголовкового файлу *stdio.h* стандартної бібліотеки мови програмування С тим, що вони завжди працюють з консоллю. Тому ці функції можна використовувати для вводу-виводу на консоль навіть у разі перенаправлення стандартного вводу-виводу.

У наступних лістингах наведено програми, в яких стандартний ввід-висновок перенаправляється в анонімний канал, а для обміну даними по анонімному каналу використовуються перевантажені оператори введення-виведення мови програмування С++. Приклад включає програми наступних процесів: два процесу-клієнта, які обмінюються даними по анонімному каналу, і процес-сервер, який створює клієнтів та передає їм дескриптори анонімного каналу через поля структури STARTUPINFO. В лістингах також наведені програми процесів-клієнтів.

### Процес-клієнт, який записує дані в анонімний канал

#include <windows.h>

#include <conio.h>

#include <iostream>

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

// події для синхронізації обміну даними

HANDLE hReadFloat, hReadText;

wchar\_t lpszReadFloat[] = L"ReadFloat";

wchar\_t lpszReadText[] = L"ReadText";

// відкриваємо події

hReadFloat = CreateEvent(NULL, FALSE, FALSE, lpszReadFloat);

hReadText = CreateEvent(NULL, FALSE, FALSE, lpszReadText);

// чекаємо команди про початок запису в анонімний канал

\_cputs("Press any key to start communication.\n");

\_getch();

// пишемо цілі числа в анонімний канал

for (int i = 0; i < 5; ++i)

{

Sleep(500);

cout << i << endl;

}

// чекаємо дозвіл на читання плаваючих чисел з каналу

WaitForSingleObject(hReadFloat, INFINITE);

// читаємо плаваючі числа з анонімного каналу

for (int j = 0; j < 5; ++j)

{

float nData;

cin >> nData;

\_cprintf("The number %2.1f is read from the pipe.\n", nData);

}

// відзначаємо, що можна читати текст з анонімного каналу

SetEvent(hReadText);

// тепер передаємо текст

cout << "This is a demo sentence." << endl;

// відзначаємо кінець передачі

cout << '\0' << endl;

\_cputs("The process finished transmission of data.\n");

\_cputs("Press any key to exit.\n");

\_getch();

CloseHandle(hReadFloat);

CloseHandle(hReadText);

return 0;

}

### Процес-клієнт, читання даних з анонімного каналу

#include <windows.h>

#include <conio.h>

#include <iostream>

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

// події для синхронізації обміну даними

HANDLE hReadFloat, hReadText;

wchar\_t IpszReadFloat[] = L"ReadFloat";

wchar\_t IpszReadText[] = L"ReadText";

// відкриваємо події

hReadFloat = CreateEvent(NULL, FALSE, FALSE, IpszReadFloat);

hReadText = CreateEvent(NULL, FALSE, FALSE, IpszReadText);

// читаємо цілі числа з анонімного каналу

for (int i = 0; i < 5; ++i)

{

int nData;

cin >> nData;

\_cprintf("The number %d is read from the pipe\n", nData);

}

// дозволяємо читати плаваючі числа з анонімного каналу

SetEvent(hReadFloat);

// пишемо плаваючі числа анонімний канал

for (int j = 0; j < 5; ++j)

{

Sleep(500);

cout << (j \* 0.1) << endl;

}

// чекаємо дозволу на читання тексту

WaitForSingleObject(hReadText, INFINITE);

\_cputs("The process read the text: ");

// тепер читаємо текст

char lpszlnput[80];

do

{

Sleep(500);

cin >> lpszlnput;

\_cputs(lpszlnput);

\_cputs(" ");

} while (\*lpszlnput != '\0');

\_cputs("\nThe process finished transmission of data.\n");

\_cputs("Press any key to exit.\n");

\_getch();

CloseHandle(hReadFloat);

CloseHandle(hReadText);

return 0;

}

### Процес-сервер, анонімного каналу

#include <windows.h>

#include <cstdio>

int main()

{

wchar\_t lpszComLine1[] = L"чотирнадцятий додаток.exe"; // имя первого клиента

wchar\_t lpszComLine2[] = L"п’ятнадцятий додаток.exe"; // имя второго клиента

STARTUPINFO si;

PROCESS\_INFORMATION pi;

HANDLE hWritePipe, hReadPipe;

SECURITY\_ATTRIBUTES sa;

// встановлює атрибути захисту каналу

sa.nLength = sizeof(SECURITY\_ATTRIBUTES);

sa.lpSecurityDescriptor = nullptr; // захист за замовчуванням

sa.bInheritHandle = TRUE; // дескриптори успадковані

// створюємо анонімний канал

if (

!CreatePipe(

&hReadPipe, // дескриптор для читання

&hWritePipe, // дескриптор для запису

&sa, // атрибути захисту за замовчуванням, дескриптори успадковані

0// розмір буфера за замовчуванням

)

)

{

printf("Create pipe failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// встановлюємо атрибути нового процесу

ZeroMemory(&si, sizeof(STARTUPINFO));

si.cb = sizeof(STARTUPINFO);

// використовувати стандартні дескриптори

si.dwFlags = STARTF\_USESTDHANDLES;

// встановлюємо стандартні дескриптори

si.hStdInput = hReadPipe;

si.hStdOutput = hWritePipe;

si.hStdError = hWritePipe;

// запускаємо першого клієнта

if (

!CreateProcess(

nullptr, // ім'я процесу

lpszComLine1, // командний рядок

nullptr, // атрибути захисту процесу за замовчуванням

nullptr, // атрибути захисту первинного потоку за замовчуванням

TRUE, // успадковані дескриптори поточного процесу

// успадковуються новим процесом

CREATE\_NEW\_CONSOLE, // створюємо нову консоль

nullptr, // використовуємо середовище оточення процесу-предка

nullptr, // поточний диск і каталог, як і в процесі-предка

&si, // вигляд головного вікна - за замовчуванням

&pi // тут будуть дескриптори і ідентифікатори

// нового процесу і його первинного потоку

)

)

{

printf("Create process failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// закриваємо дескриптори першого клієнта

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

// запускаємо другого клієнта

if (

!CreateProcess(

nullptr, // ім'я процесу

lpszComLine2, // командний рядок

nullptr, // атрибути захисту процесу за замовчуванням

nullptr, // атрибути захисту первинного потоку за замовчуванням

TRUE, // успадковані дескриптори поточного процесу

// успадковуються новим процесом

CREATE\_NEW\_CONSOLE, // створюємо нову консоль

nullptr,//використовуємо середовище оточення процесу - предка

nullptr,// поточний диск і каталог, як і в процесі-предка

&si,// вигляд головного вікна - за замовчуванням

&pi// тут будуть дескриптори і ідентифікатори

// нового процесу і його первинного потоку

)

)

{

printf("Create process failed.\n");

printf("Press any key to finish.\n");

getchar();

return GetLastError();

}

// закриваємо дескриптори другого клієнта

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

// закриваємо дескриптори каналу

CloseHandle(hReadPipe);

CloseHandle(hWritePipe);

printf("The clients are created.\n");

printf("Press any key to exit.\n");

getchar();

return 0;

}

# Робота з іменованими каналами в Windows

Робота з іменованими каналами, також як і робота з анонімними каналами, вимагає спільного використання цілого ряду функцій. Тому спочатку будуть дані характеристики іменованих каналів та розглянуто всі функції, які призначені для передачі даних по іменованим каналах. Потім будуть наведені кілька прикладів, які проілюструють використання цих функцій. Після цього будуть розглянуті службові функції для роботи з іменованими каналами.

## Іменовані канали

Іменованим каналом називається об'єкт ядра операційної системи, які забезпечує передачу даних між процесами, які виконуються на комп'ютерах в локальній мережі. Процес, який створює іменований канал, називається сервером іменованого каналу. Процеси, які зв'язуються з іменованим каналом, називаються клієнтами іменованого каналу. Перерахуємо характеристики іменованих каналів:

* можуть бути як напівдуплексні, так і дуплексні;
* передача даних може здійснюватися як потоком, так і повідомленнями;
* обмін даними може бути як синхронним, так і асинхронним;
* можливість моделювання будь-якої топології зв'язків.

Тепер наведемо порядок роботи з іменованими каналами, який і буде використовуватися в подальшому:

* створення іменованого каналу сервером:
* з'єднання сервера з примірником іменованого каналу;
* з'єднання клієнта з примірником іменованого каналу;
* обмін даними по іменованого каналу;
* від'єднання від сервера примірника іменованого каналу;
* закриття іменованого каналу клієнтом і сервером.

Докладно ці пункти будуть розглянуті в наступних розділах.

## Створення іменованих каналів

Іменовані канали створюються процесом-сервером за допомогою функції *CreateNamedPipe*, яка має наступний прототип:

HANDLE WINAPI CreateNamedPipe(

\_In\_     LPCTSTR               lpName, // ім'я каналу

\_In\_     DWORD                 dwOpenMode, // атрибути каналу

\_In\_     DWORD                 dwPipeMode, // режим передачі даних

\_In\_     DWORD                 nMaxInstances, // максимальна кількість примірників

\_In\_     DWORD                 nOutBufferSize, // розмір вихідного буфера

\_In\_     DWORD                 nInBufferSize, // розмір вхідного буфера

\_In\_     DWORD                 nDefaultTimeOut, // час очікування зв'язку з клієнтом

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes // атрибути безпеки

);

При вдалому завершенні функція *CreateNamedPipe* повертає дескриптор іменованого каналу, а в разі невдачі — одне з двох значень:

* *INVALID\_HANDLE\_VALUE* — невдале завершення;
* *ERROR\_INVALID\_PARAMETER* — значення параметра *nMaxInstances* більше, ніж величина *PIPE\_UNLIMITED\_INSTANCES*.

Опишемо параметри цієї функції.

Параметр *lpName* вказує на рядок, яка повинна мати вигляд:

* \\.\pipe\pipe\_name

Тут позначає локальну машину, так як новий іменований канал завжди створюється на локальній машині, слово *pipe* — фіксовано, a *pipe\_name* позначає ім'я каналу, яке задається користувачем і нечутливий до регістру.

Параметр *dwOpenMode* задає прапори, які визначають напрямок передачі даних, буферизацію, синхронізацію обміну даними та права доступу до іменованого каналу. Для визначення напрямку передачі даних використовуються прапори:

* *PIPE\_ACCESS\_DUPLEX* — читання і запис в канал;
* *PIPE\_ACCESS\_INBOUND* — клієнт пише, а сервер читає дані;
* *PIPE\_ACCESS\_OUTBOUND* — сервер пише, а клієнт читає дані.

Крім того, кожен з цих прапорів дозволяє використовувати дескриптор каналу для синхронізації.

Прапор, який визначає напрямок передачі даних по іменованого каналу, повинен збігатися для всіх екземплярів одного і того ж іменованого каналу. Для визначення способу буферизації і синхронізації використовуються прапори:

* *FILE\_FLAG\_WRITE\_THROUGH* — забороняє буферизацію при передачі даних по мережі;
* *FILE\_FLAG\_OVERLAPPED* — дозволяє асинхронну передачу даних по каналу. Ці прапори можуть бути різними для кожного екземпляра одного і того ж іменованого каналу.

У цьому ж пункті можуть бути задані інші режими доступу до іменованого каналу. розглянуті вище прапори *PIPE\_ACCESS\_DUPLEX*, *PIPE\_ACCESS\_INBOUND* і *PIPE\_ACCESS\_OUTBOUND* визначають специфічні права доступу до іменованого каналу, які включають наступні родові права доступу:

* *РІРЕ\_ТУРЕ\_ВУТЕ* — запис даних потоком;
* *РIРЕ\_ТYPE\_MESSAGE* — запис даних повідомленнями;
* *PIPE\_READMODE\_BYTE* — читання даних потоком;
* *PIPE\_READMODE\_MESSAGE* — читання даних повідомленнями.

За замовчуванням дані по іменованого каналу передаються потоком.

Прапори, що визначають спосіб читання і запису даних в іменований канал, повинні бути однаковими для всіх екземплярів одного і того ж каналу. Для визначення синхронізації доступу до іменованого каналу використовуються прапори:

* *PIPE\_WAIT* — синхронний зв'язок з каналом та обмін даними по каналу;
* *PIPE\_NOWAIT* — асинхронний зв'язок з каналом та обмін даними по каналу.

Ці прапори можуть бути різними для кожного екземпляра іменованого каналу.

Параметр *nMaxInstances* визначає максимальне число примірників іменованого каналу, яке може перебувати в межах від 1 до *PIPE\_UNLIMITED\_INSTANCES*. Кожен примірник каналу призначений для обміну даними по каналу між сервером і окремим клієнтом.

Параметри *nOutBufferSize* і *nInBufferSize* визначають відповідно розміри вихідного і вхідного буферів для обміну даними по іменованого каналу. Однак ці значення розглядаються операційними системами Windows тільки як побажання користувача, а сам вибір розмірів буферів залишається за операційною системою.

Параметр *nDefauitTimeOut* встановлює час очікування клієнтом зв'язку з сервером. Цей час використовується при виклику клієнтом функції *WaitNamedPipe*, в якій параметр *nTimeOut* миє значення *NMPWAIT\_USE\_ DEFAULT\_WAIT*.

Для зв'язку сервера з декількома клієнтами по одному іменованого каналу сервер повинен створити кілька примірників цього каналу. Кожен примірник іменованого каналу створюється викликом функції *СreateNamedPipe*, яка повертає дескриптор примірника іменованого каналу. Зазначимо, що в цьому випадку потік, який створює екземпляр іменованого каналу, повинен мати право доступу *FILE\_CREATE\_PIPE\_INSTANCE* до іменованого каналу. Цим правом за замовчуванням має власник іменованого каналу, тобто той процес, який створив цей іменований канал.

## З'єднання сервера з клієнтом

Після того як сервер створив іменований канал, він повинен дочекатися з'єднання клієнта з цим каналом. Для цього сервер викликає функцію *ConnectNamedPipe*, яка має наступний прототип;

BOOL WINAPI ConnectNamedPipe(

\_In\_        HANDLE       hNamedPipe, // дескриптор каналу

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped // асинхронний зв'язок

);

У разі успішного завершення ця функція повертає ненульове значення, а у випадку невдачі — значення *false*. Сервер використовує цю функцію для зв'язку з клієнтом по кожному вільному примірнику іменованого каналу.

Після закінчення обміну даними з клієнтом сервер може викликати функцію DisconnectNamedPipe, яка має наступний прототип:

BOOL WINAPI DisconnectNamedPipe(

\_In\_ HANDLE hNamedPipe // дескриптор каналу

);

І повертає ненульове значення — в разі успіху — або значення *false* — у випадку невдачі. Ця функція розриває зв'язок сервера з клієнтом. Після цього клієнт не може обмінюватися даними з сервером за даним іменованого каналу і тому будь-яка операція доступу до іменованого каналу з боку клієнта викличе помилку. Після розриву зв'язку з одним клієнтом сервер знову може викликати функцію *ConnectNamedPipe*, щоб встановити зв'язок з цього ж іменованого каналу з іншим клієнтом.

## З'єднання клієнтів з іменованим каналом

Перш ніж з'єднуватися з іменованим каналом, клієнт повинен визначити, чи доступний який-небудь примірник цього каналу для з'єднання. З цією метою клієнт повинен викликати функцію *WaitNamedPipe*, яка має наступний прототип:

BOOL WINAPI WaitNamedPipe(

\_In\_ LPCTSTR lpNamedPipeName, // вказівник на ім'я

\_In\_ DWORD   nTimeOut // інтервал очікування

);

У разі успішного завершення ця функція повертає ненульове значення, а у випадку невдачі — *false*. Опишемо параметри цієї функції.

Параметр *lpNamedPipeName* вказує на рядок, яка повинна мати вигляд:

\\server\_name\pipe\pipe\_name

Тут *server\_name* позначає ім'я комп'ютера, на якому виконується сервер іменованого каналу, слово *pipe* фіксовано, a *pipe\_name* задає ім'я іменованого каналу.

Параметр *nTimeOut* задає часовий інтервал, протягом якого клієнт чекає зв'язок з сервером. Цей часовий інтервал визначається в мілісекундах або може дорівнювати одному з наступних значень:

* *NMPWAIT\_USE\_DEFAULT\_WAIT* — інтервал часу очікування визначається значенням параметра *nDefaultTimeOut*, який задається в функції *CreateNamedPipe*;
* *NMPWAIT\_WAIT\_FOREVER* — нескінченне час очікування зв'язку з іменованими каналом.

Зробимо два важливих зауваження щодо роботи функції *WaitNamedPipe*.

Зауваження

Якщо не існує примірників іменованого каналу з ім'ям *lpNamedPipe*, то функція *WaitNamedPipe* негайно завершується невдачею незалежно від часу очікування, заданого параметром *nTimeOut*.

Зауваження

Якщо клієнт з'єднується з каналом до виклику сервером функції *ConnectNamedPipe*, то функція *WaitNamedPipe* повертає значення *false* і *GetLastError* функція поверне код *ERROR\_PIPE\_CONNECTED*. тому функцію *WaitNamedPipe* потрібно викликати тільки після з'єднання сервера з каналом допомогою функції *ConnectNamedPipe*.

Після того як виявлено вільний примірник каналу, щоб встановити зв'язок з цим каналом клієнт повинен викликати функцію *CreateFile*, яка має наступний прототип:

HANDLE WINAPI CreateFile(

\_In\_     LPCTSTR               lpFileName, // вказівник на ім'я каналу

\_In\_     DWORD                 dwDesiredAccess, // читання або запис в канал

\_In\_     DWORD                 dwShareMode, // режим спільного використання

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // атрибути безпеки

\_In\_     DWORD                 dwCreationDisposition, // прапор відкриття каналу

\_In\_     DWORD                 dwFlagsAndAttributes, // прапори та атрибути

\_In\_opt\_ HANDLE                hTemplateFile // додаткові атрибути

);

У разі успішного завершення ця функція повертає дескриптор іменованого каналу, а в разі невдачі — значення *INVALID\_HANDLE\_VALUE*.

Якщо функція використовується для відкриття іменованого каналу, то її параметри можуть приймати наступні значення.

Параметр *lpFileName* повинен вказувати на ім'я каналу, яке повинно бути задано у тому ж форматі, що і у функції *WaitNamedPipe*. Відзначимо, що якщо клієнт працює на тій же машині, що і сервер, і використовує для відкриття іменованого каналу у функції *CreateFile* ім'я каналу

* \\ .\pipe\pipe\_name

файлова система іменованих каналів (*Named Pipe File System*, *NPFS*) відкриває цей іменований канал у режимі передачі даних потоком.

Щоб відкрити іменований канал у режимі передачі даних повідомленнями потрібно задавати ім'я каналу у вигляді:

* \\server\_name\pipe\pipe\_name

Параметр *dwDesiredAccess* може приймати одне з наступних значень:

* 0 — дозволяє отримати атрибути каналу;
* *GENERIC\_READ* — дозволяє читання з каналу;
* *GENERIC\_WRITE* — дозволяє запис в канал.

Слід зазначити, що функція *CreateFile* завершується невдачею, якщо доступ до іменованого каналу, визначений цими значеннями, не відповідає значенням параметра *dwOpenMode* у функції *CreateNamedPipe*. Крім того в цьому параметрі програміст може визначити стандартні права доступу до іменованого каналу.

Параметр *dwShareMode* визначає режим спільного використання іменованого каналу і може приймати значення 0, яке забороняє спільне використання іменованого каналу або будь-яку комбінацію з наступних значень:

* *FILE\_SHARE\_READ* — дозволяє спільне читання з каналу;
* *FILE\_SHARE\_WRITE* — дозволяє спільну запис в канал.

Параметр *lpSecurityAttributes* задає атрибути безпеки іменованого каналу. Поки значення цього параметра будемо встановлювати в *NULL*.

Для іменованого каналу параметр *dwCreationDisposition* має дорівнювати значенню *OPEN\_EXISTING*, так як клієнт завжди відкриває існуючий іменований канал.

Для іменованого каналу параметр *dwFlagsAndAttributes* можна задати рівним 0, що визначає прапори та атрибути за замовчуванням.

Значення параметра *hTemplateFile* задається рівним *NULL*.

Зробимо наступні зауваження щодо роботи з функцією *CreateFile* у разі, коли вона використовується для відкриття доступу до іменованого каналу.

Зауваження

Незважаючи на те, що функція *WaitNamedPipe* може успішно завершитися, наступний виклик функції *CreateFile* може завершитися невдачею з наступних причин: між викликами цих функцій сервер закрив канал або між викликами функцій інший клієнт зв'язався з примірником цього каналу. Для запобігання останньої ситуації сервер повинен створювати новий примірник іменованого каналу після кожного успішного завершення функції *ConnectNamedPipe* або створити відразу кілька примірників іменованого каналу.

Зауваження

Якщо заздалегідь відомо, що сервер викликав функцію *ConnectNamedPipe*, то функція *CreateFile* може викликатися без попереднього виклику функції *WaitNamedPipe*.

Відзначимо також один момент, який стосується зв'язку сервера з клієнтом іменованого каналу. Може виникнути така ситуація, що сервер викликав функцію *ConnectNamedPipe*, а клієнта, який хоче зв'язатися з іменованим каналом, не існує. У цьому випадку серверний додаток буде заблоковано. Щоб мати можливість опрацювати таку ситуацію, функцію *ConnectNamedPipe* слід викликати в окремому потоці серверного додатка. Тоді для розблокування серверного додатка можна викликати функцію для зв'язку клієнта з іменованим каналом з іншого потоку цього додатка.

## Обмін даними по іменованого каналу

Як і у випадку з анонімним каналом, для обміну даними по іменованого каналу використовуються функції *ReadFile* і *WriteFile*, але з однією відмінністю, яка полягає в наступному. Так як у випадку іменованого каналу дозволений асинхронний обмін даними, то у функції *ReadFile* і *WriteFile* може використовуватися параметр *lpOverlapped* при тій умові, що у виклику функції *CreateNamedPipe* в параметрі *dwOpenMode* був встановлений прапор *FILE\_FLAG\_OVERLAPPED*. Максимально в іменований канал може бути записано до 65 535 байт однією операцією *WriteFile*. Крім того, для асинхронного обміну даними по іменованого каналу можуть використовуватися також функції *ReadFileEx* і *WriteFileEx*.

Після завершення обміну даними по іменованого каналу потоки повинні закрити дескриптори примірників іменованого каналу, використовуючи функцію *CloseHandle*.

Тепер перейдемо до прикладів програм, в яких показано, як обмінюватися даними по іменованого каналу. Спочатку розглянемо приклад, в якому процес-сервер створює іменований канал, а потім чекає, поки клієнт не з'єднається з іменованим каналом. Після цього сервер читає з іменованого каналу десять чисел і виводить їх на консоль. Спочатку наведемо програму процесу-сервера іменованого каналу.

### Процес-сервер іменованого каналу

#include <windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hNamedPipe;

// створюємо іменований канал для читання

hNamedPipe = CreateNamedPipeW(

L"\\\\.\\pipe\\demo\_pipe", // ім'я каналу

PIPE\_ACCESS\_INBOUND, // читаємо з каналу

PIPE\_TYPE\_MESSAGE | PIPE\_WAIT, // синхронна передача повідомлень

1, // максимальна кількість примірників каналу

0, // розмір вихідного буфера за замовчуванням

0, // розмір вхідного буфера за замовчуванням

INFINITE, // клієнт чекає зв'язок нескінченно довго

nullptr // захист за замовчуванням

);

// перевіряємо на успішне створення

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Create named pipe failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

// чекаємо, поки клієнт зв'яжеться з каналом

cout << "The server is waiting for connection with a client." << endl;

if (

!ConnectNamedPipe(

hNamedPipe, // дескриптор каналу

nullptr // зв'язок синхронна

)

)

{

cerr << "The connection failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// читаємо дані з каналу

for (int i = 0; i < 10; i++)

{

int nData;

DWORD dwBytesRead;

if (

!ReadFile(

hNamedPipe, // дескриптор каналу

&nData, // адреса буфера для введення даних

sizeof(nData), // кількість читаних байтів

&dwBytesRead, // число прочитаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "Read file failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо прочитані дані на консоль

cout << "The number " << nData << " was read by the server" << endl;

}

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "The data are read by the server. " << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

Тепер, в лістингу наведемо програму процесу-клієнта іменованого каналу, який спочатку зв'язується з іменованим каналом, а потім записує в нього десять чисел.

### Процес клієнт, іменованого каналу

#include <windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hNamedPipe;

wchar\_t pipeName[] = L"\\\\.\\pipe\\demo\_pipe";

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_WRITE, // записуємо в канал

FILE\_SHARE\_READ, // дозволяємо одночасне читання з каналу

nullptr, // захист за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

0, // атрибути за замовчуванням

nullptr // додаткових атрибутів немає

);

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to exit.";

return 0;

}

// пишемо в іменований канал

for (int i = 0; i < 10; i++)

{

DWORD dwBytesWritten;

if (

!WriteFile(

hNamedPipe, // дескриптор каналу

&i, // дані

sizeof(i), // розмір даних

&dwBytesWritten, // кількість записаних байтів

nullptr // синхронний запис

)

)

{

// помилка запису

cerr << "Writing to the named pipe failed: " << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводим число в консоль

cout << "The number " << i << " is written to the named pipe." << endl;

Sleep(1000);

}

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "The data are written by the client." << endl

<< "Press any key to exit.";

cin.get();

return 0;

}

У лістингу розглянемо програму процесу-сервера іменованого каналу, який спочатку створює іменований канал, потім чекає підключення до нього клієнта. Після цього сервер приймає від клієнта одне повідомлення, виводить це повідомлення на консоль і посилає клієнту відповідне повідомлення.

### Сервер іменованого каналу, який відповідатиме на повідомлення клієнта

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hNamedPipe;

DWORD dwBytesRead; // для кількості прочитаних байтів

DWORD dwBytesWrite; // для кількості записаних байтів

char pchMessage[80]; // для повідомлення

int nMessageLength; // довжина повідомлення

// створюємо іменований канал для читання і запису

hNamedPipe = CreateNamedPipeW(

L"\\\\.\\pipe\\demo\_pipe", // ім'я каналу

PIPE\_ACCESS\_DUPLEX, // читаємо з каналу і пишемо в канал

PIPE\_TYPE\_MESSAGE | PIPE\_WAIT, // синхронна передача повідомлень

1, // максимальна кількість примірників каналу

0, // розмір вихідного буфера за замовчуванням

0, // розмір вхідного буфера за замовчуванням

INFINITE, // клієнт чекає зв'язок нескінченно довго

nullptr // безпека за замовчуванням

);

// перевіряємо на успішне створення

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Create named pipe failed." << endl

<< "The last error code: " << GetLastError() << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

// чекаємо, поки клієнт зв'яжеться з каналом

cout << "The server is waiting for connection with a client." << endl;

if (

!ConnectNamedPipe(

hNamedPipe, // дескриптор каналу

nullptr // зв'язок синхронний

)

)

{

cerr << "Connect named pipe failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// читаємо повідомлення від клієнта

if (

!ReadFile(

hNamedPipe, // дескриптор каналу

pchMessage, // адреса буфера для введення даних

sizeof(pchMessage), // кількість читаних байтів

&dwBytesRead, // кількість прочитаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "Data reading from the named pipe failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо отриманий від клієнта повідомлення на консоль

cout << "The server received the message from a client: "

<< endl << '\t' << pchMessage << endl;

// вводимо рядок

cout << "Input a string: ";

cin.getline(pchMessage, 80);

// визначаємо довжину рядка

nMessageLength = strlen(pchMessage) + 1;

// відповідаємо клієнту

if (

!WriteFile(

hNamedPipe, // дескриптор каналу

pchMessage, // адреса буфера для виведення даних

nMessageLength, // кількість записуваних байт

&dwBytesWrite, // кількість записаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "Write file failed." << endl

<< "The last error code: " << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо надіслане клієнту повідомлення на консоль

cout << "The server sent the message to a client: "

<< endl << '\t' << pchMessage << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

cin.get();

return 0;

}

У цій програмі звернемо увагу на наступний момент: якщо клієнт і сервер працюють на різних комп'ютерах в локальній мережі, то вхід, як на комп'ютер сервера, так і на комп'ютер клієнта, повинен здійснюватися з однаковими іменами (логінами) і паролями. Так як за замовчуванням атрибути безпеки іменованого каналу встановлюються таким чином, що він належить тільки користувачеві, що створив цей іменований канал. У наступному прикладі процесу-сервера іменованого каналу атрибути безпеки будуть встановлені таким чином, щоб вони дозволяли доступ до іменованого каналу будь-якому користувачеві. Тепер у лістингу наведемо програму процесу-клієнта іменованого каналу, який спочатку вводить з консолі ім'я комп'ютера в локальній мережі, на якому запущений сервер іменованого каналу. Потім зв'язується з цим іменованим каналом. Після цього клієнт передає серверу повідомлення і отримує від нього відповідне повідомлення і виводить на консоль.

### Клієнт іменованого каналу, який відсилає повідомлення серверу

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

char machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwBytesWritten; // кількість записаних байтів

DWORD dwBytesRead; // для кількості прочитаних байтів

char pchMessage[80]; // для повідомлення

int nMessageLength; // довжина повідомлення

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter a name of the server machine:";

cin >> machineName;

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr); // додаткових атрибутів немає

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

// вводимо рядок

cin.get();

cout << "Input a string:";

cin.getline(pchMessage, 80);

// визначаємо довжину рядка

nMessageLength = strlen(pchMessage) + 1;

// пишемо в іменований канал

if (

!WriteFile(

hNamedPipe, // дескриптор каналу

pchMessage, // дані

nMessageLength, // розмір даних

&dwBytesWritten, // кількість записаних байтів

nullptr // синхронний запис

)

)

{

// помилка запису

cerr << "Write file failed:" << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо надіслане повідомлення на консоль

cout << "The client sent the message to a server: "

<< endl << '\t' << pchMessage << endl;

// читаємо з іменованого каналу

if (

!ReadFile(

hNamedPipe, // дескриптор каналу

pchMessage, // дані

sizeof(pchMessage), // розмір даних

&dwBytesRead, // кількість записаних байтів

nullptr // синхронне читання

)

)

{

// помилка читання

cerr << "Read file failed:. " << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// вьгоодим отримане повідомлення на консоль

cout << "The client received the message from a server:" << endl

<< '\t' << pchMessage << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

cin.get();

return 0;

}

Тепер наведемо програму процесу-сервера іменованого каналу, який створює іменований канал, доступний всім користувачам. У цьому випадку зв'язатися з іменованим каналом можна будь-якому користувачеві.

### Сервер, який створює загальнодоступний іменований канал

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hNamedPipe;

SECURITY\_ATTRIBUTES sa; // атрибути безпеки

SECURITY\_DESCRIPTOR sd; // дескриптор безпеки

DWORD dwBytesRead; // для кількості прочитаних байтів

DWORD dwBytesWrite; // кількість записаних байтів

char pchMessage[80]; // для повідомлення

int nMessageLength; // довжина повідомлення

// ініціалізація атрибутів безпеки

sa.nLength = sizeof(sa);

sa.bInheritHandle = FALSE; // дескриптор каналу ненаследуемый

// ініціалізуємо дескриптор безпеки

InitializeSecurityDescriptor(&sd, SECURITY\_DESCRIPTOR\_REVISION);

// дозволяємо доступ до іменованого каналу усім користувачам

SetSecurityDescriptorDacl(&sd, TRUE, nullptr, FALSE);

sa.lpSecurityDescriptor = &sd;

// створюємо іменований канал для читання і запису

hNamedPipe = CreateNamedPipeW(

L"\\\\.\\pipe\\demo\_pipe", // ім'я каналу

PIPE\_ACCESS\_DUPLEX, // читаємо з каналу і пишемо в канал

PIPE\_TYPE\_MESSAGE | PIPE\_WAIT, // синхронна передача повідомлень

1, // максимальна кількість примірників каналу

0, // розмір вихідного буфера за замовчуванням

0, // розмір вхідного буфера за замовчуванням

INFINITE, // клієнт чекає зв'язок нескінченно довго

&sa // доступ для всіх користувачів

);

// перевіряємо на успішне створення

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Create named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

// чекаємо, поки клієнт зв'яжеться з каналом

cout << "The server is waiting for connection with a client." << endl;

if (

!ConnectNamedPipe(

hNamedPipe, // дескриптор каналу

nullptr // зв'язок синхронна

)

)

{

cerr << "Connect named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// читаємо повідомлення від клієнта

if (

!ReadFile(

hNamedPipe, // дескриптор каналу

pchMessage, // адреса буфера для введення даних

sizeof(pchMessage), // кількість читаних байтів

&dwBytesRead, // кількість прочитаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "reading Data from the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо отриманий від клієнта повідомлення на консоль

cout << "The server received the message from a client: "

<< endl << '\t' << pchMessage << endl;

// вводимо рядок

cout << "Input a string: ";

cin.getline(pchMessage, 80);

// визначаємо довжину рядка

nMessageLength = strlen(pchMessage) + 1;

// відповідаємо клієнту

if (

!WriteFile(

hNamedPipe, // дескриптор каналу

pchMessage, // адреса буфера для виведення даних

nMessageLength, // кількість записуваних байт

&dwBytesWrite, // кількість записаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "Write file failed." << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо надіслане клієнту повідомлення на консоль

cout << "The server-sent the message to a client: "

<< endl << '\t' << pchMessage << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

cin.get();

return 0;

}

На завершення цього розділу відзначимо, що для створення і зв'язку з іменованими каналами користувач повинен мати відповідні повноваження.

## Копіювання даних з іменованого каналу

Для копіювання даних з іменованого каналу використовується функція *PeekNamedPipe*, яка копіює дані в буфер, не видаляючи їх з каналу. Ця функція має наступний прототип:

BOOL WINAPI PeekNamedPipe(

\_In\_      HANDLE  hNamedPipe, // дескриптор каналу

\_Out\_opt\_ LPVOID  lpBuffer, // буфер даних

\_In\_      DWORD   nBufferSize, // розмір буфера даних

\_Out\_opt\_ LPDWORD lpBytesRead, // кількість прочитаних байтів

\_Out\_opt\_ LPDWORD lpTotalBytesAvail, // кількість доступних байтів

\_Out\_opt\_ LPDWORD lpBytesLeftThisMessage // кількість непрочитаних байтів

);

У разі успішного завершення ця функція повертає ненульове значення, а у випадку невдачі — *false*. Слід зазначити, що якщо даних в каналі немає, то функція негайно повертає управління і встановлює їх в нуль значення змінних, на які вказують параметри *lpBytesRead*, *lpTotalBytesAvail* і *lpBytesLeftThisMessage*. Параметри функції *PeekNamedPipe* мають наступне призначення.

У параметрі *hNamedPipe* повинен бути встановлений дескриптор іменованого каналу. Причому канал повинен бути відкритий в режимі читання.

Параметр *lpBuffer* повинен вказувати на область пам'яті, в яку функція читає дані з іменованого каналу. Якщо цього параметра встановлено значення *NULL* то дані з іменованого каналу читатися не будуть.

У параметрі *nBufferSize* повинен бути встановлений розмір області пам'яті, на яку вказує параметр *lpBuffer*.

Параметр *lpBytesRead* повинен вказувати на змінну типу *DWORD*, в яку функція записує кількість прочитаних байтів. Якщо дані з іменованого каналу не читаються, то значення цього параметра може бути встановлено в *NULL*.

Параметр *lpTotalBytesAvail* повинен вказувати на змінну типу *DWORD*, в яку функція записує кількість доступних для читання байтів, які перебувають у іменованому каналі. Якщо дані з іменованого каналу не читаються, то значення цього параметра може бути встановлено в *NULL*.

Параметр *lpBytesLeftThisMessage* повинен вказувати на змінну, в яку функція поміщає кількість непрочитаних байтів з повідомлення. Якщо дані з іменованого каналу не читаються, то значення цього параметра може бути встановлено в *NULL*.

Відзначимо, що якщо дані передаються потоком, а не повідомленнями, функція *PeekNamedPipe* читає рівно стільки байтів даних, яка довжина буфера даних. Якщо ж дані передаються повідомленнями, функція читає повністю повідомлення, яке входить в буфер даних. В іншому випадку читається частину повідомлення, а кількість непрочитаних байтів повертається через змінну, на яку вказує параметр *lpBytesLeftThisMessage*.

У лістингу наведена програма процесу-клієнта іменованого каналу, який використовує функцію *PeekNamedPipe* для копіювання даних з іменованого каналу.

### Зразок копіювання повідомлення клієнтом іменованого каналу

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cin;

using std::cout;

using std::endl;

int main(int argc, char \*argv[])

{

char machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwBytesRead; // для кількості прочитаних байтів

DWORD dwTotalBytesAvail; // кількість байтів в каналі

DWORD dwBytesLeftThisMessage; // кількість непрочитаних байтів

char pchMessage[80]; // для повідомлення

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter a name of the server machine: ";

cin >> machineName;

cin.get();

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr// додаткових атрибутів немає

);

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

cin.get();

return 0;

}

// чекаємо команду на копіювання повідомлення з каналу

cout << "Press any key to peek a message." << endl;

cin.get();

// копіюємо інформацію з іменованого каналу

if (

!PeekNamedPipe(

hNamedPipe, // дескриптор каналу

pchMessage, // дані

sizeof(pchMessage), // розмір даних

&dwBytesRead, // кількість записаних байтів

&dwTotalBytesAvail, // кількість байтів в каналі

&dwBytesLeftThisMessage // кількість непрочитаних байтів

)

)

{

// помилка читання повідомлення

cerr << "Peek named pipe failed:" << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо отримане повідомлення на консоль

if (dwTotalBytesAvail)

cout << "The peeked message: "

<< endl << '\t' << pchMessage << endl;

else

cout << "There is no mesage." << endl;

// тепер читаємо повідомлення з іменованого каналу

if (

!ReadFile(

hNamedPipe, // дескриптор каналу

pchMessage, // дані

sizeof(pchMessage), // розмір даних

&dwBytesRead, // кількість записаних байтів

nullptr// синхронне читання

)

)

{

// помилка читання

cerr << "Read file failed:" << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

cin.get();

return 0;

}

// виводимо отримане повідомлення на консоль

cout << "The client received the message from a server: "

<< endl << '\t' << pchMessage << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit." << endl;

cin.get();

return 0;

}

## Передача транзакцій по іменованого каналу

Для обміну повідомленнями по мережі може також використовуватися функція *TransactNamedPipe*, яка об'єднує операції запису і читання в одну операцію, яка називається трансакцією. Зазначимо, що функція *TransactNamedPipe* може використовуватися тільки в тому випадку, якщо сервер при створенні іменованого каналу встановив прапор *PIPE\_TYPE\_MESSAGE*. Функція *TransactNamedPipe* має наступний прототип

BOOL WINAPI TransactNamedPipe(

\_In\_        HANDLE       hNamedPipe, // дескриптор іменованого каналу

\_In\_        LPVOID       lpInBuffer, // буфер для запису в канал

\_In\_        DWORD        nInBufferSize, // довжина буфера для запису

\_Out\_       LPVOID       lpOutBuffer, // буфер для читання з каналу

\_In\_        DWORD        nOutBufferSize, // довжина буфера для читання

\_Out\_       LPDWORD      lpBytesRead, // кількість прочитаних байтів

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped // асинхронний доступ до каналу

);

У разі успішного завершення функція *TransactNamedPipe* повертає ненульове значення, а у випадку невдачі — *false*. Параметри функції мають наступне призначення.

У параметрі *hNamedPipe* повинен бути встановлений дескриптор іменованого каналу.

Параметр *lpInBuffer* повинен вказувати на буфер, з якого записуються дані в іменований канал.

Параметр *dwInBufferSize* повинен містити довжину переданого повідомлення в байтах.

Параметр *lpOutBuffer* повинен вказувати на буфер, в який читаються дані з іменованого каналу.

Параметр *dwOutBufferSize* повинен містити довжину буфера, який читається повідомлення.

Параметр *lpBytesRead* повинен вказувати на змінну типу *DWORD*, в яку функція помістить кількість прочитаних байтів. Якщо здійснюється асинхронний доступ до іменованого каналу, то значення цього параметра можна встановити в *NULL*.

Параметр *lpOveriapped* використовується в тому випадку, якщо здійснюється асинхронний доступ до іменованого каналу. В цьому випадку параметр повинен вказувати на структуру типу *OVERLAPPED*. Поки будемо встановлювати цей параметр *NULL*, що задає синхронну передачу даних.

У лістингу наведена програма процесу-клієнта іменованого каналу, яка передає транзакцію, використовуючи функцію *TransactNamedPipe*. Відмітим, що якщо клієнт і сервер іменованого каналу працюють на одному комп'ютері, то для зв'язку клієнта з іменованим каналом потрібно вводити повне ім'я комп'ютера. Якщо ж замість імені комп'ютера буде введена точка, то іменований канал відкриється в режимі передачі даних потоком, а не повідомленнями. Це, в свою чергу, викличе помилку при передачі даних.

### Приклад передачи транзакції по іменованому каналу

#include <windows.h>

#include <iostream>

#include <string>

using std::cout;

using std::wcin;

using std::cerr;

using std::endl;

int main()

{

wchar\_t machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwBytesRead; // для кількості прочитаних байтів

wchar\_t pchlnBuffer[80]; // для запису повідомлення

char pchOutBuffer[80]; // для читання повідомлення

int nMessageLength; // довжина повідомлення

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter а паші of the server machine: ";

wcin >> machineName;

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr// додаткових атрибутів немає

);

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// вводимо рядок

wcin.get();

cout << "Input a string: ";

wcin.getline(pchlnBuffer, 80);

// визначаємо довжину рядка

nMessageLength = wcslen(pchlnBuffer) + 1;

// пишемо і читаємо з іменованого каналу однією транзакцією

if (

!TransactNamedPipe(

hNamedPipe, // дескриптор каналу

&pchlnBuffer, // адреса вхідного буфера каналу

nMessageLength, // довжина вхідного повідомлення

&pchOutBuffer, // адреса вихідного буфера каналу

sizeof(pchOutBuffer), // довжина вихідного буфера каналу

&dwBytesRead, // кількість прочитаних байтів

nullptr// передача транзакції синхронна

)

)

{

// помилка транзакції

cerr << "Transact named pipe failed:" << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hNamedPipe);

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// виводимо надіслане повідомлення на консоль

cout << "The message sent: "

<< endl << '\t' << pchlnBuffer << endl;

// виводимо отримане повідомлення на консоль

cout << "The message received: "

<< endl << '\t' << pchOutBuffer << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

Для передачі єдиною транзакції по іменованого каналу використовується функція *CallNamedPipe*, яка працює наступним чином. Спочатку вона зв'язується з іменованим каналом, використовуючи його ім'я. При цьому зауважимо, що іменований канал повинен бути відкритий в режимі передачі даних повідомленнями. Потім функція передає за іменованого каналу єдине повідомлення і отримує повідомлення у відповідь, а після цього розриває зв'язок з іменованим каналом. Функція *CallNamedPipe* має наступний прототип:

BOOL WINAPI CallNamedPipe(

\_In\_  LPCTSTR lpNamedPipeName, // ім'я іменованого каналу

\_In\_  LPVOID  lpInBuffer, // буфер для записи данных в канал

\_In\_  DWORD   nInBufferSize, // розмір буфера для запису даних

\_Out\_ LPVOID  lpOutBuffer, // буфер для читання даних з каналу

\_In\_  DWORD   nOutBufferSize, // розмір буфера для читання даних

\_Out\_ LPDWORD lpBytesRead, // кількість прочитаних байтів

\_In\_  DWORD   nTimeOut // інтервал очікування

);

У разі успішного завершення функція *CallNamedPipe* повертає ненульове значення, а у випадку невдачі — *false*. Опишемо параметри цієї функції.

Параметр *lpNamedPipeName* повинен вказувати на рядок, що містить ім'я іменованого каналу.

Параметр *lpInBuffer* повинен вказувати на буфер, з якого записуються дані в іменований канал.

Параметр *nInBufferSize* повинен містити довжину переданого повідомлення в байтах.

Параметр *lpOutBuffer* повинен вказувати на буфер, з який читаються дані іменованого каналу.

Параметр *nOutBufferSize* повинен містити довжину буфера, який читається повідомлення.

Параметр *lpBytesRead* повинен вказувати на змінну типу *DWORD*, в яку функція помістить кількість прочитаних байтів.

У параметрі *nTimeOut* повинен бути встановлений інтервал очікування в мілісекундах, протягом якого функція чекає зв'язку з іменованим каналом.

Крім того, можна встановити наступні значення цього параметра:

* *NMPWAIT\_NOWAIT* — якщо немає вільного примірника іменованого каналу, то функція негайно повертає управління;
* *NMPWAIT\_WAIT\_FOREVER* — функція чекає нескінченно довго зв'язку з примірником іменованого каналу;
* *NMPWAIT\_USE\_DEFAULT\_WAIT* — інтервал очікування визначається значенням, заданим при створенні іменованого каналу функцією CreateNamedPipe

У лістингу наведена програма процесу-клієнта іменованого каналу, який передає транзакцію, використовуючи функцію CallNamedPipe.

### Приклад передачи транзакцій по іменованому каналу

#include <windows.h>

#include <iostream>

#include <string>

using std::wcin;

using std::cout;

using std::cerr;

using std::endl;

int main(int argc, char \*argv[])

{

wchar\_t machineName[80];

wchar\_t pipeName[80];

DWORD dwBytesRead; // для кількості прочитаних байтів

wchar\_t pchInBuffer[80]; // для запису повідомлення

wchar\_t pchOutBuffer[80]; // для читання повідомлення

int nMessageLength; // довжина повідомлення

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter a name of the server machine: ";

wcin >> machineName;

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// вводимо рядок

wcin.get();

cout << "Input a string: ";

wcin.getline(pchInBuffer, 80);

// визначаємо довжину рядка

nMessageLength = wcslen(pchInBuffer) + 1;

// зв'язуємося, пишемо і читаємо з іменованого каналу однією транзакцією

if (

!CallNamedPipe(

pipeName, // ім'я іменованого каналу

&pchInBuffer, // адреса вхідного буфера каналу

nMessageLength, // довжина вхідного повідомлення

&pchOutBuffer, // адреса вихідного буфера каналу

sizeof(pchOutBuffer), // довжина вихідного буфера каналу

&dwBytesRead, // кількість прочитаних байтів

NMPWAIT\_WAIT\_FOREVER// чекаємо нескінченно довго

)

)

{

// помилка транзакції

cerr << "Call named pipe failed:" << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// виводимо надіслане повідомлення на консоль

cout << "The message sent: "

<< endl << '\t' << pchInBuffer << endl;

// виводимо отримане повідомлення на консоль

cout << "The message received: "

<< endl << '\t' << pchOutBuffer << endl;

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

## Визначення та зміна стану іменованого каналу

Для отримання інформації про стан іменованого каналу використовується функція *GetNamedPipeHandleState*, яка має наступний прототип:

BOOL WINAPI GetNamedPipeHandleState(

\_In\_      HANDLE  hNamedPipe, // дескриптор іменованого каналу

\_Out\_opt\_ LPDWORD lpState, // стан каналу

\_Out\_opt\_ LPDWORD lpCurInstances, // кількість екземплярів каналу

\_Out\_opt\_ LPDWORD lpMaxCollectionCount, // максимальна кількість байтів

\_Out\_opt\_ LPDWORD lpCollectDataTimeout, // інтервал очікування

\_Out\_opt\_ LPTSTR  lpUserName, // ім'я клієнта іменованого каналу

\_In\_      DWORD   nMaxUserNameSize // довжина буфера для імені клієнта

);

У разі успішного завершення функція *GetNamedPipeHandleState* повертає ненульове значення, а у випадку невдачі — *false*. Параметри функції мають наступне призначення.

У параметрі *hNamedPipe* повинен бути встановлений дескриптор іменованого каналу. Причому канал повинен бути відкритий в режимі читання.

Параметр *lpState* повинен вказувати на змінну типу *DWORD*, в яку функція записує будь-яку комбінацію з наступних значень:

* *PIPE\_NOWAIT* — канал не блокований;
* *PIPE\_READMODE\_MESSAGE* — канал открит в режимі передачі даних повідомленнями.

Якщо значення *PIPE\_NOWAIT* не встановлено, то канал блокований. Якщо не встановлено значення *PIPE\_READMODE\_MESSAGE*, то канал відкритий в режимі передачі даних потоком. Якщо визначати стан іменованого каналу не потрібно, то в параметрі *lpState* може бути встановлено значення *NULL*.

Параметр *lpCurInstances* повинен вказувати на змінну типу *DWORD*, в яку функція записує кількість створених примірників іменованого каналу. Якщо ця інформація не потрібна, то значення цього параметра може бути встановлено в *NULL*.

Параметр *lpMaxCollectionCount* повинен вказувати на змінну типу *DWORD*, в яку функція запише максимальна кількість байтів, які клієнт іменованого каналу повинен записати в цей канал, перш ніж дані будуть передані до сервера каналу. Цей параметр повинен бути встановлений у *NULL*, якщо функція викликається сервером іменованого каналу або якщо клієнт і сервер працюють на одному комп'ютері і для зв'язку з сервером клієнт використовує символ '.' замість імені сервера. Якщо інформація про максимальну кількість байтів не потрібна, то в цьому параметрі може бути встановлено значення *NULL*.

Параметр *lpCollectDataTimeout* повинен вказувати на змінну типу *DWORD*, в яку функція помістить кількість мілісекунд, які можуть пройти, перш ніж дані можуть бути передані по мережі. Як і в попередньому випадку, цей параметр повинен бути встановлений у *NULL*, в тому випадку, якщо функція викликається сервером іменованого каналу або якщо клієнт і сервер працюють на одному комп'ютері, і для зв'язку з сервером клієнт використовує символ замість імені сервера. Якщо інформація про інтервал затримки перед передачею даних не потрібна, то в цьому параметрі може бути встановлено значення *NULL*.

Параметр *lpUserName* повинен вказувати на символьний масив, в який функція помістить рядок з ім'ям власника іменованого каналу. Ім'я власника можна отримати тільки в тому випадку, якщо доступ до каналу відкритий всім користувачам. Якщо інформація про ім'я власника каналу не потрібна, то цей параметр може бути встановлений в *NULL*.

Параметр *nMaxUserNameSize* повинен містити розмір області пам'яті, на яку вказує параметр *lpUserName*.

У лістингу наведена програма процесу-клієнта іменованого каналу, яка отримує інформацію про стан іменованого каналу за допомогою виклику функції *GetNamedPipeHandleState*.

### Приклад визначення стану іменованого каналу

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

wchar\_t machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwState; // стан каналу

DWORD dwCurInstances; // кількість екземплярів каналу

DWORD dwMaxCollectionCount; // розмір буфера клієнта каналу

DWORD dwCollectDataTimeout; // затримка перед передачею даних

TCHAR chUserName[255]; // ім'я власника іменованого каналу

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter a name of the server machine: ";

wcin >> machineName;

wcin.get();

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

//канал

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr// додаткових атрибутів немає

);

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// визначаємо стан каналу

if (

!GetNamedPipeHandleState(

hNamedPipe, // дескриптор іменованого каналу

&dwState, // стан іменованого каналу

&dwCurInstances, // кількість екземплярів каналу

&dwMaxCollectionCount, // розмір буфера клієнта каналу

&dwCollectDataTimeout, // макс, затримка перед передачею даних

chUserName, // ім'я користувача каналу

255// максимальна довжина імені

)

)

{

cerr << "Get named pipe handle failed state." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// виводимо стан каналу на консоль

cout << "State: ";

switch (dwState)

{

case (PIPE\_NOWAIT):

cout << "PIPE\_NOWAIT" << endl; break;

case (PIPE\_READMODE\_MESSAGE):

cout << "PIPE\_READMODE\_MESSAGE" << endl; break;

case (PIPE\_NOWAIT | PIPE\_READMODE\_MESSAGE):

cout << "PIPE\_NOWAIT and PIPE\_READMODE\_MESSAGE" << endl; break;

default:

cout << "Unknown state." << endl; break;

}

cout << "Current instances:" << dwCurInstances << endl

<< "Max collection count : "<< dwMaxCollectionCount << endl

<< "Data Collection timeout:" << dwCollectDataTimeout << endl

<< "User name:" << chUserName << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

Змінити деякі характеристики іменованого каналу можна за допомогою функції SetNamedPipeHandleState, яка має наступний прототип:

BOOL WINAPI SetNamedPipeHandleState(

\_In\_     HANDLE  hNamedPipe, // дескриптор іменованого каналу

\_In\_opt\_ LPDWORD lpMode, // новий режим передачі даних

\_In\_opt\_ LPDWORD lpMaxCollectionCount, // максимальна кількість байтів

\_In\_opt\_ LPDWORD lpCollectDataTimeout // інтервал очікування

);

У разі успішного завершення функція *SetNamedPipeHandleState* повертає ненульове значення, а у випадку невдачі — *false*. Параметри функції мають наступне призначення.

У параметрі *hNamedPipe* повинен бути встановлений дескриптор іменованого каналу. Причому канал повинен бути відкритий в режимі запису.

Параметр *lpMode* повинен вказувати на змінну типу *DWORD*, яка містить нові режими роботи іменованого каналу. Можливо змінити наступні режими роботи іменованого каналу: режим передачі даних і режим очікування при виконанні запису або зчитування даних в іменований канал, а також очікування сервером з'єднання клієнта з іменованим каналом. Режим передачі даних може приймати наступні значення:

* *PIPE\_READMODE\_BYTE* — передача даних потоком;
* *PIPE\_READMODE\_MESSAGE* — передача даних повідомленнями.

Режим очікування може приймати наступні значення:

* *PIPE\_WAIT* — блокування додатки до завершення виконання функцій *ConnectNamedPipe*, *WriteFile* і *ReadFile*;
* *PIPE\_NOWAIT* — виконання функцій *ConnectNamedPipe*, *WriteFile* і *ReadFile* не блокує роботу додатка.

Зазначимо, що ці режими очікування не впливають на роботу асинхронних операцій доступу до іменованого каналу, а призначені тільки для роботи менеджера локальної мережі. У параметрі *lpMode* може бути встановлена будь-яка комбінація ознак режимів передачі даних і очікування або тільки один з цих прапорів. Якщо режими роботи іменованого каналу не змінюються, то параметр *lpMode* повинен містити значення *NULL*.

Параметр *lpMaxCollectionCount* повинен вказувати на змінну типу *DWORD*, яка містить максимальну кількість байтів, які клієнт іменованого каналу може записати в цей канал, перш ніж ці дані будуть передані до сервера каналу. Однак якщо клієнт відкрив іменований канал у режимі *FILE\_FLAG\_WRITE\_THROUGH*, параметр *lpMaxCollectionCount* ігнорується. Цей параметр повинен бути встановлений у *NULL*, якщо функція не змінює розмір максимальної кількості записуваних перед передачею байтів.

Параметр *lpCollectDataTimeout* повинен вказувати на змінну типу *DWORD*, яка містить нову кількість мілісекунд, які можуть пройти, перш ніж дані бути передані по мережі. Як і в попередньому випадку, якщо клієнт відкрив іменований канал у режимі *FILE\_FLAG\_WRITE\_THROUGH*, то параметр *lpCollectDataTimeout* ігнорується. Цей параметр повинен бути встановлений у *NULL*, якщо функція не змінює інтервал очікування до передачі даних по іменованого каналу.

У лістингу наведена програма процесу-клієнта іменованого каналу, яка змінює стан іменованого каналу за допомогою виклику функції *SetNamedPipeHandleState*.

### Приклад зміни стану іменованого каналу

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::endl;

using std::cout;

using std::wcin;

int main(int argc, char \*argv[])

{

wchar\_t machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwState; // стан каналу

DWORD dwCurInstances; // кількість екземплярів каналу

DWORD dwMaxCollectionCount; // розмір буфера клієнта каналу

DWORD dwCollectDataTimeout; // тимчасова затримка перед передачею даних

TCHAR chUserName[255]; // ім'я власника іменованого каналу

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter а паші of the server machine: ";

wcin >> machineName;

wcin.get();

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr// додаткових атрибутів немає

);

// перевіряємо зв'язок з каналом

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// визначаємо стан каналу

if (

!GetNamedPipeHandleState(

hNamedPipe, // дескриптор іменованого каналу

&dwState, // стан іменованого каналу

&dwCurInstances, // кількість екземплярів каналу

&dwMaxCollectionCount, // розмір буфера клієнта каналу

&dwCollectDataTimeout, // макс, затримка перед передачею даних

chUserName, // ім'я користувача каналу

255// максимальна довжина імені

)

)

{

cerr << "Get named pipe handle failed state." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// виводимо інтервал очікування на консоль

cout << "data Collection timeout:" << dwCollectDataTimeout << endl;

// змінюємо стан іменованого каналу

dwCollectDataTimeout = 100;

if (

!SetNamedPipeHandleState(

hNamedPipe, // дескриптор іменованого каналу

nullptr, // режим передачі даних не змінюємо

nullptr, // розмір буфера не змінюємо

&dwCollectDataTimeout// макс, затримка дорівнює 100 мілісекунд

)

)

{

cerr << "Set named pipe handle failed state." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// визначаємо стан каналу

GetNamedPipeHandleState(

hNamedPipe, // дескриптор іменованого каналу

&dwState, // стан іменованого каналу

&dwCurInstances, // кількість екземплярів каналу

&dwMaxCollectionCount, // розмір буфера клієнта каналу

&dwCollectDataTimeout, // макс, затримка перед передачею даних

chUserName, // ім'я користувача каналу

255// максимальна довжина імені

);

// виводимо інтервал очікування на консоль

cout << "data Collection timeout;" << dwCollectDataTimeout << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

## Отримання інформації про іменованому каналі

Для отримання інформації про атрибути іменованого каналу, які не можуть бути змінені, використовується функція *GetNamedPipeInfo*. Ця функція має наступний прототип:

BOOL WINAPI GetNamedPipeInfo(

\_In\_      HANDLE  hNamedPipe, // дескриптор іменованого каналу

\_Out\_opt\_ LPDWORD lpFlags, // тип каналу

\_Out\_opt\_ LPDWORD lpOutBufferSize, // розмір вихідного буфера

\_Out\_opt\_ LPDWORD lpInBufferSize, // розмір вхідного буфера

\_Out\_opt\_ LPDWORD lpMaxInstances // макс, кількість примірників каналу

);

У разі успішного завершення функція *GetNamedPipeInfo* повертає ненульове значення, а у випадку невдачі — *false*. Опишемо параметри цієї функції.

У параметрі *hNamedPipe* повинен бути встановлений дескриптор іменованого каналу. Причому канал повинен бути відкритий в режимі читання.

Параметр *lpFlags* повинен вказувати на змінну типу *DWORD*, в якій встановлено тип іменованого каналу, про який треба одержати інформацію. Тип іменованого каналу повинен містити інформацію про спосіб передачі даних по каналу і вказувати, кому належить дескриптор: клієнта або сервера іменованого каналу. Для цього повинні використовуватися наступні константи:

* *PIPE\_CLIENT\_END* — дескриптор клієнта іменованого каналу;
* *PIPE\_SERVER\_ANDROID* — дескриптор сервера іменованого каналу;
* *PIPE\_TYPE\_BYTE* — передача даних потоком;
* *PIPE\_TYPE\_MESSAGE* — Передача доних повідомленнями.

Якщо ця інформація не встановлена, тобто параметр *lpFlags* має значення *NULL* функція розглядає той кінець іменованого каналу, який з'єднаний з клієнтом, по якому дані передаються потоком.

Параметр *lpOutBufferSize* повинен вказувати на змінну типу *DWORD*, в яку функція *GetNamedPipeInfo* помістить розмір вихідного буфера іменованого каналу. Якщо інформація про розмір вихідного буфера не потрібна, то значення цього параметра може бути встановлено в *NULL*.

Параметр *lpInBufferSize* повинен вказувати на змінну типу *DWORD*, в яку функція *GetNamedPipeInfo* помістить розмір вхідного буфера іменованого каналу. Якщо інформація про розмір вхідного буфера не потрібна, то значення цього параметра може бути встановлено в *NULL*.

Параметр *lpMaxInstances* повинен вказувати на змінну типу *DWORD*, в яку функція *GetNamedPipeInfo* помістить число, що позначає максимально допустима кількість примірників іменованого каналу. Якщо це число дорівнює значенню *PIPE\_UNLIMITED\_INSTANCES*, то кількість примірників іменованого каналу обмежена лише наявністю системних ресурсів. Якщо інформація про максимальній кількості примірників іменованого каналу не потрібна, то значення цього параметра може бути встановлено в *NULL*.

У лістингу наведена програма процесу-клієнта іменованого каналу, яка отримує інформацію про незмінних атрибутів іменованого каналу за допомогою виклику функції *GetNamedPipeInfo*.

### Приклад отримання інформації по іменований канал

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

wchar\_t machineName[80];

wchar\_t pipeName[80];

HANDLE hNamedPipe;

DWORD dwFlags = PIPE\_CLIENT\_END | PIPE\_TYPE\_MESSAGE; // клієнт каналу і передача даних повідомленнями

DWORD dwOutBufferSize; // стан каналу

DWORD dwInBufferSize; // кількість екземплярів каналу

DWORD dwMaxInstances; // розмір буфера клієнта каналу

// вводимо ім'я машини в мережі, на якій працює сервер

cout << "Enter а паші of the server machine: ";

wcin >> machineName;

wcin.get();

// підставляємо ім'я машини в ім'я каналу

wsprintfW(pipeName, L"\\\\%s\\pipe\\demo\_pipe", machineName);

// зв'язуємося з іменованим каналом

hNamedPipe = CreateFile(

pipeName, // ім'я каналу

GENERIC\_READ | GENERIC\_WRITE, // читаємо і записуємо в канал

FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, // дозволяємо читання і запис

nullptr, // безпека за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

FILE\_ATTRIBUTE\_NORMAL, // атрибути за замовчуванням

nullptr // додаткових атрибутів немає

);

if (hNamedPipe == INVALID\_HANDLE\_VALUE)

{

cerr << "Connection with the named pipe failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// отримуємо інформацію про канал

if (

!GetNamedPipeInfo(

hNamedPipe, // дескриптор іменованого каналу

&dwFlags, // тип каналу

&dwOutBufferSize, // розмір вихідного буфера

&dwInBufferSize, // розмір вхідного буфера

&dwMaxInstances// максимальна кількість примірників каналу

)

)

{

cerr << "Get named pipe info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to exit.";

wcin.get();

return 0;

}

// виводимо інформацію на консоль

cout << "Out buffer size:" << dwOutBufferSize << endl

<< "In buffer size:" << dwInBufferSize << endl

<< "Maxinstances : "<< dwMaxInstances << endl;

// закриваємо дескриптор каналу

CloseHandle(hNamedPipe);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

# Робота з поштовими скриньками в Windows

## Концепція поштових скриньок

Поштовою скринькою називається об'єкт ядра операційної системи, який забезпечує передачу повідомлень від процесів-клієнтів до процесів-серверів, тих, що виконуються на комп'ютерах в межах локальної мережі. Процес, який створює поштову скриньку, називається сервером поштової скриньки. Процеси, які зв'язуються з іменованою поштовою скринькою, називаються клієнтами поштової скриньки. Перерахуємо характеристики поштових скриньок:

* мають ім'я, яке використовується клієнтами для зв'язку з поштовими скриньками;
* напрямок передачі даних від клієнта до сервера;
* передача даних здійснюється повідомленнями;
* обмін даними може бути як синхронним, так і асинхронним.

Хоча передача даних здійснюється тільки від клієнта до сервера, один поштову скриньку може мати декілька серверів. Це відбувається в тому випадку, якщо кілька серверів створюють поштові скриньки з однаковими іменами. Тоді всі повідомлення, які надсилає клієнт в таку поштову скриньку, будуть отримувати всі сервери поштової скриньки. Однак це виконується тільки за умови, що довжина повідомлення менше 425 байт, так як в цьому випадку повідомлення передаються дейтаграммами. Таким чином, можна сказати, що поштові скриньки забезпечують односпрямований зв'язок «багато-до-багатьох». При цьому доставка повідомлення від клієнта до сервера поштової скриньки не підтверджується системою. Зауважимо також, що операційні системи сімейства *Windows NT* не підтримують передачу повідомлень довжиною 425 і 426 байт.

Трохи торкнемося питання передачі повідомлень за допомогою поштових скриньок. Якщо довжина повідомлення менше ніж 425 байт, то таке повідомлення передається як дейтаграма. Дейтаграма являє собою невеликий пакет з передаваних по мережі повідомленням, що містить також інформацію про відправника та одержувача повідомлення. Дейтаграма розсилається всім серверам даної поштової скриньки. Так як розмір пакета невеликий, то дейтаграми розсилаються швидко, але немає гарантії доставки повідомлення, так як в дейтаграмме не зберігається інформація, підтримуюча контроль доставки. Якщо ж довжина повідомлення більше 426 байт, то такі повідомлення можуть передаватися тільки від одного клієнта до одного сервера, використовуючи при цьому *SMB* (*Server Message Block*) протокол передачі даних по мережі. При цьому зазначимо, що довжина повідомлення, що передається в поштову скриньку, не може перевищувати 64 Кбайт. Крім того, при використанні протоколу *SMB* повідомлення передаються лише від одного клієнта до одного сервера поштової скриньки.

Поштові скриньки можна розглядати як псевдофайли, розташовані в оперативній пам'яті комп'ютера. Тому для доступу до поштових скриньок використовуються ті ж функції, що і для доступу до звичайних файлів.

Тепер наведемо порядок роботи з поштовими скриньками, який і буде використовуватися в подальшому.

1. Створення поштової скриньки на сервері.

2. З'єднання клієнта з поштовою скринькою.

3. Обмін даними через поштову скриньку.

4. Закриття поштової скриньки клієнтом і сервером.

## Створення поштових скриньок

Створюються поштові скриньки процесом-сервером за допомогою функції CreateMailslot, яка має наступний прототип:

HANDLE WINAPI CreateMailslot(

\_In\_     LPCTSTR               lpName, // ім'я поштової скриньки

\_In\_     DWORD                 nMaxMessageSize, // максимальна довжина повідомлення

\_In\_     DWORD                 lReadTimeout, // інтервал очікування

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes // атрибути безпеки

);

У разі успішного завершення ця функція повертає дескриптор поштової скриньки, а в разі невдачі — значення *INVALID\_HANDLE\_VALUE*. Опишемо параметри функції.

Параметр *lpName* вказує на рядок, яка повинна мати вигляд:

* \\.\mailslot\[path]name

Тут символ '.' означає локальну машину, так як новий поштовий ящик завжди створюється на локальній машині, слово *mailslot* — фіксовано, a *[path]name* визначає ім'я поштової скриньки, яке задається користувачем, це поле нечутливий до верхнього і нижнього регістрів.

Параметр *nMaxMessageSize* задає максимальну довжину повідомлення в байтах, яке може бути записано в поштову скриньку.

Параметр *lReadTimeout* задає в мілісекундах часовий інтервал, протягом якого функція *ReadFile* чекає надходження повідомлення в почтовий ящик. Якщо цього параметра встановлено значення 0, то в разі відсутності в поштовій скриньці повідомлення функція негайно повертає управління. Для завдання нескінченного часу очікування в цьому параметрі потрібно встановити значення *MAILSLOT\_WAIT\_FOREVER*.

Зауважимо, що кілька процесів можуть створити поштові скриньки з одним і тим же ім'ям. У цьому випадку повідомлення, надіслане клієнтом, можуть доставлятися не тільки в поштову скриньку одного процесу, а також у поштові скриньки всіх таких процесів за умови, що вони працюють на комп'ютерах в межах одного домену. Режим доставки повідомлень залежить від режиму відкриття поштової скриньки клієнтом.

## З'єднання клієнтів з поштовою скринькою

Як вже говорилося, для встановлення зв'язку з поштовою скринькою клієнт використовує функцію CreateFile, яка має наступний прототип:

HANDLE WINAPI CreateFile(

\_In\_     LPCTSTR               lpFileName, // вказівник на ім'я каналу

\_In\_     DWORD                 dwDesiredAccess, // читання або запис в канал

\_In\_     DWORD                 dwShareMode, // режим спільного використання

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // атрибути безпеки

\_In\_     DWORD                 dwCreationDisposition, // прапор відкриття каналу

\_In\_     DWORD                 dwFlagsAndAttributes, // прапори та атрибути

\_In\_opt\_ HANDLE                hTemplateFile // додаткові атрибути

);

У разі успішного завершення ця функція повертає дескриптор поштової скриньки, а в разі невдачі — значення *INVALID\_HANDLE\_VALUE*.

Параметри функції *CreateFile* можуть приймати наступні значення (якщо ця функція використовується для відкриття поштової скриньки).

Параметр *lpFileName* повинен вказувати на ім'я поштової скриньки, яке може бути задано в одному з наступних форматів:

* поштова скринька на локальному комп'ютері:

\\.\mailslot\ім'я\_поштової\_скриньки

* поштова скринька на комп'ютері з вказаним ім'ям:

\\ім'я\_компьтера\mailslot\ім'я\_поштової\_скриньки

* поштова скринька в домені із зазначеним ім'ям:

\\ім'я\_домену\mailslot\ім'я\_поштової\_скриньки

* поштова скринька в первинному домені системи:

\\\*\mailslot\ім'я\_поштової\_скриньки

У першому випадку повідомлення надходять тільки в поштові ящики з заданим ім'ям, які розташовані на локальній машині. У другому випадку повідомлення надходять у поштові скриньки з заданим ім'ям, розташовані на комп'ютері з вказаним ім'ям. У третьому випадку повідомлення надходять у поштові скриньки з заданим ім'ям, які створені всередині домену із зазначеним ім'ям. У четвертому випадку повідомлення надходять у поштові скриньки з заданим ім'ям, які створені усередині первинного домену системи.

Параметр *dwDesiredAccess* повинен мати значення *GENERIC\_READ*, яке дозволяє запис в поштову скриньку.

Параметр *dwShareMode* визначає режим спільного використання поштової скриньки і може приймати будь-яку комбінацію з наступних значень:

* *FILE\_SHARE\_READ* — дозволяє спільне читання з поштової скриньки;
* *FILE\_SHARE\_WRITE* — дозволяє спільну запис в поштову скриньку.

Параметр *lpSecurityAttributes* задає атрибути безпеки поштової скриньки. Поки цей параметр будемо встановлювати в *NULL*.

Для поштової скриньки параметр *dwCreationDisposition* має дорівнювати значенню *OPEN\_EXISTING*, так як клієнт завжди відкриває існуючий поштову скриньку.

Для поштової скриньки параметр *dwFlagsAndAttributes* можна задати рівним 0, що визначає прапори та атрибути за замовчуванням, або встановити в цьому параметрі значення *FILE\_ATTRIBUTE\_NORMAL*.

Параметр *hTemplateFile* при роботі з поштовими скриньками не використовується, тому в ньому встановлюється значення *NULL*.

## Обмін даними через поштову скриньку

Для обміну даними через поштову скриньку використовуються звичайні функції доступу до файлу *WriteFile* і *ReadFile*. Процес-клієнт записує дані в поштову скриньку за допомогою функцій *WriteFile*, а процес-сервер читає дані з поштової скриньки, використовуючи функції *ReadFile*. В цьому розділі наведемо лише програми, які обмінюються даними через поштову скриньку. Звичайно, процес-сервер також може записувати повідомлення в поштову скриньку, але це має сенс лише в тому випадку, якщо в домені розташовано декілька поштових скриньок з однаковими іменами, які були створені різними процесами. Тоді однакові повідомлення отримують всі поштові скриньки з однаковими іменами.

У лістингу наведемо програму процесу-сервера, який створює поштову скриньку. У цій програмі поштова скринька створюється на локальній машині.

### Процес-сервер поштової скриньки

#include <windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hMailslot; // дескриптор поштової скриньки

// створюємо поштовий ящик

hMailslot = CreateMailslotW(

L"\\\\.\\mailslot\\demo\_mailslot", // ім'я поштової скриньки

0, // довжина довільна повідомлення

MAILSLOT\_WAIT\_FOREVER, // чекаємо повідомлення довільно довго

NULL // безпека за замовчуванням

);

// перевіряємо на успішне створення

if (hMailslot == INVALID\_HANDLE\_VALUE)

{

cerr << "Create mailslot failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

cout << "The mailslot is created." << endl;

cout << "The mailslot is waiting a message." << endl;

// читаємо одне ціле число з поштової скриньки

int nData;

DWORD dwBytesRead;

if (

!ReadFile(

hMailslot, // дескриптор поштової скриньки

&nData, // адреса буфера для введення даних

sizeof(nData), // кількість читаних байтів

&dwBytesRead, // кількість прочитаних байтів

(LPOVERLAPPED)NULL // синхронна передача даних

)

)

{

cerr << "Read file failed." << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hMailslot);

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

// виводимо число на консоль

cout << "The number" << nData << "was read by the server" << endl;

// закрьшаем дескриптор поштової скриньки

CloseHandle(hMailslot);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

Тепер, в лістингу наведемо програму процесу-клієнта поштової скриньки. У цій програмі передбачається, що процес-клієнт запускається на тій же локальній машині, що і процес-сервер. Інакше в ім'я поштової скриньки потрібно вказати не крапку, а ім'я комп'ютера, на якому створено поштову скриньку, або ім'я домену, в якому створені поштові скриньки з однаковими іменами.

### Процес-клієнт поштової скриньки

#include <Windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hMailslot;

wchar\_t mailslotName[] = L"\\\\.\\mailslot\\demo\_mailslot";

// зв'язуємося з поштовою скринькою

hMailslot = CreateFile(

mailslotName, // ім'я поштової скриньки

GENERIC\_WRITE, // записьюаем в ящик

FILE\_SHARE\_READ, // дозволяємо одночасне читання з ящика

nullptr, // захист за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

0, // атрибути за замовчуванням

nullptr // додаткових атрибутів немає

);

// перевіряємо зв'язок з поштовою скринькою

if (hMailslot == INVALID\_HANDLE\_VALUE)

{

cerr << "Create file failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish the client.";

wcin.get();

return 0;

}

// вводимо ціле число

int n;

cout << "Input an integer: ";

wcin >> n;

// пишемо число в поштову скриньку

DWORD dwBytesWritten;

if (

!WriteFile(

hMailslot, // дескриптор поштової скриньки

&n, // дані

sizeof(n), // розмір даних

&dwBytesWritten, // кількість записаних байтів

nullptr // синхронний запис

)

)

{

// помилка запису

cerr << "Write file failed:" << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish the client.";

wcin.get();

CloseHandle(hMailslot);

return 0;

}

// закриваємо дескриптор каналу

CloseHandle(hMailslot);

// завершуємо процес

cout << "The number is written by the client." << endl

<< "Press any key to exit." << endl;

wcin.get();

return 0;

}

На завершення цього розділу відзначимо, що, оскільки доступ до поштових скриньок не відрізняється від доступу до файлів, для обміну даними через поштові скриньки можна також використовувати функції *WriteFileEx* і *ReadFileEx*.

## Отримання інформації про поштовій скриньці

Для отримання інформації про характеристики поштової скриньки використовується функція *GetMailslotInfo*, яка має наступний прототип:

BOOL WINAPI GetMailslotInfo(

\_In\_      HANDLE  hMailslot, // дескриптор поштової скриньки

\_Out\_opt\_ LPDWORD lpMaxMessageSize, // максимальна довжина повідомлення

\_Out\_opt\_ LPDWORD lpNextSize, // довжина наступного повідомлення

\_Out\_opt\_ LPDWORD lpMessageCount, // повідомлень

\_Out\_opt\_ LPDWORD lpReadTimeout // інтервал очікування повідомлення

);

У разі успішного завершення ця функція повертає ненульове значення, а у випадку невдачі — *NULL*. Параметри функції *GetMailslotInfo* мають наступне призначення.

У параметрі *hMailslot* повинен бути встановлений дескриптор поштової скриньки, який був повернутий функцією *CreateMailslot*.

Параметр *lpMaxMessageSize* повинен вказувати на змінну типу *DWORD*, в яку функція *GetMailslotInfo* помістить максимальну довжину повідомлення, яке може бути записано в поштову скриньку. Якщо це значення не потрібно, то цей параметр може бути встановлений в *NULL*.

Параметр *lpNextSize* повинен вказувати на змінну типу *DWORD*, в яку функція *GetMailslotInfo* помістить довжину наступного повідомлення в поштовій скриньці. Якщо в поштовій скриньці немає повідомлень, то цей параметр функція запише значення *MAILSLOT\_NO\_MESSAGE*. Якщо значення довжини останнього повідомлення не потрібно, то параметр *lpNextSize* може бути встановлений в *NULL*.

Параметр *lpMessageCount* повинен вказувати на змінну типу *DWORD*, в яку функція *GetMailslotInfo* помістить кількість повідомлень, що знаходяться в поштовій скриньці. Якщо це значення не потрібно, то цей параметр може бути встановлений в *NULL*.

Параметр *lpReadTimeout* повинен вказувати на змінну типу *DWORD*, в яку функція *GetMailslotInfo* помістить ціле число без знака, що позначає інтервал часу в мілісекундах. Якщо поштова скринька порожня, то протягом цього інтервалу функція *ReadFile* буде чекати, поки процес-клієнт не запише повідомлення в поштову скриньку. Якщо це значення не потрібно, то в цьому параметрі може бути встановлено значення *NULL*.

В лістингах наведено програми, в яких функція *GetMailslotInfo* використовується для отримання інформації про повідомлення, яке зберігаються в поштовій скриньці. Спочатку наведемо програму сервера поштової скриньки, який читає повідомлення по мірі їх надходження від клієнта.

### Процес-сервер поштової скриньки

#include <windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::cin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hMailslot; // дескриптор поштової скриньки

DWORD dwNextMessageSize; // розмір наступного повідомлення

DWORD dwMessageCount; // кількість повідомлень

// створюємо поштовий ящик

hMailslot = CreateMailslotW(

L"\\\\.\\mailslot\\demo\_mailslot", // ім'я поштової скриньки

0, // довжина довільна повідомлення

MAILSLOT\_WAIT\_FOREVER, // чекаємо повідомлення довільно довго

nullptr // безпека за замовчуванням

);

// перевіряємо на успішне створення

if (hMailslot == INVALID\_HANDLE\_VALUE)

{

cerr << "Create mailslot failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

cin.get();

return 0;

}

cout << "The mailslot is created." << endl;

// чекаємо повідомлень

cout << "Press any key to read messages." << endl;

cin.get();

// отримуємо інформацію про поштовій скриньці

if (

!GetMailslotInfo(

hMailslot, // дескриптор поштової скриньки

nullptr, // максимальний розмір повідомлення не потрібен

&dwNextMessageSize, // розмір наступного повідомлення

&dwMessageCount, // кількість повідомлень

nullptr// інтервал очікування не потрібен

)

)

{

cerr << "Get mailslot info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

cin.get();

return 0;

}

// читаємо повідомлення

while (dwMessageCount != 0)

{

DWORD dwBytesRead;

char\* pchMessage;

// захоплюємо пам'ять для повідомлень

pchMessage = (char\*) new char[dwNextMessageSize];

// читаємо одне повідомлення

if (

!ReadFile(

hMailslot, // дескриптор каналу

pchMessage, // адреса буфера для введення даних

dwNextMessageSize, // кількість читаних байтів

&dwBytesRead, // кількість прочитаних байтів

nullptr // синхронна передача даних

)

)

{

cerr << "Read file failed." << endl

<< "The last error code:" << GetLastError() << endl;

CloseHandle(hMailslot);

cout << "Press any key to finish server.";

cin.get();

return 0;

}

// виводимо повідомлення на консоль

cout << "The message <<" << pchMessage << ">> was read" << endl;

// отримуємо інформацію про наступному повідомленні

if (

!GetMailslotInfo(

hMailslot, // дескриптор поштової скриньки

nullptr, // максимальний розмір повідомлення не потрібен

&dwNextMessageSize, // розмір наступного повідомлення

&dwMessageCount, // кількість повідомлень

nullptr// інтервал очікування не потрібен

)

)

{

cerr << "Get mailslot info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

cin.get();

return 0;

}

// звільняємо пам'ять для повідомлень

delete[] pchMessage;

}

// закриваємо дескриптор поштової скриньки

CloseHandle(hMailslot);

// завершуємо процес

cout << "Press any key to exit.";

cin.get();

return 0;

}

Тепер, в лістингу наведемо програму клієнта поштової скриньки, яка надсилає повідомлення процесу-сервера. Зазначимо, що як процес-сервер, так і процес-клієнт знаходяться на локальному комп'ютері.

### Процес-клієнт поштової скриньки

#include <windows.h>

#include <iostream>

#include <string>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hMailslot;

wchar\_t mailslotName[] = L"\\\\.\\mailslot\\demo\_mailslot";

// зв'язуємося з поштовою скринькою

hMailslot = CreateFileW(

mailslotName, // ім'я поштової скриньки

GENERIC\_WRITE, // записуємо в ящик

FILE\_SHARE\_READ, // дозволяємо одночасне читання з ящика

nullptr, // захист за замовчуванням

OPEN\_EXISTING, // відкриваємо існуючий канал

0, // атрибути за замовчуванням

nullptr // додаткових атрибутів немає

);

// перевіряємо зв'язок з поштовою скринькою

if (hMailslot == INVALID\_HANDLE\_VALUE)

{

cerr << "Create file failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish the client.";

wcin.get();

return 0;

}

// вводимо кількість переданих повідомлень

int n;

cout << "Input a number of messages: ";

wcin >> n;

wcin.get();

// пишемо повідомлення в поштову скриньку

for (int i = 0; i < n; ++i)

{

DWORD dwBytesWritten;

wchar\_t pchMessage[256];

int nMessageSize;

cout << "Input message: ";

// читаємо повідомлення

wcin.getline(pchMessage, 256);

// определем довжину повідомлення

nMessageSize = wcslen(pchMessage) + 1;

// пишемо повідомлення

if (

!WriteFile(

hMailslot, // дескриптор поштової скриньки

pchMessage, // дані

nMessageSize, // розмір даних

&dwBytesWritten, // кількість записаних байтів

nullptr // синхронний запис

)

)

{

// помилка запису

cerr << "Write file failed:" << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish the client.";

wcin.get();

CloseHandle(hMailslot);

return 0;

}

}

// закриваємо дескриптор каналу

CloseHandle(hMailslot);

// завершуємо процес

cout << "The messages are written by the client." << endl

<< "Press any key to exit." << endl;

wcin.get();

return 0;

}

## Зміна часу очікування повідомлення

Для зміни часу очікування сервером повідомлення від клієнта використовується функція *SetMailslotInfo*, яка має наступний прототип:

BOOL WINAPI SetMailslotInfo(

\_In\_ HANDLE hMailslot, // дескриптор поштової скриньки

\_In\_ DWORD  lReadTimeout // интервал ожидания сообщения

);

У разі успішного завершення ця функція повертає ненульове значення, а у випадку невдачі — *NULL*.

У параметрі *hMailslot* повинен бути встановлений дескриптор поштової скриньки, який був отриманий викликом функції *CreateMailslot*.

Параметр *lReadTimeout* задає в мілісекундах новий часовий інтервал, протягом якого функція *ReadFile* чекає надходження повідомлення в поштову скриньку. Якщо в цьому параметрі встановлюється значення 0, то в разі відсутності в поштовій скриньці повідомлення функція негайно повертає управління. Для завдання нескінченного часу очікування в цьому параметрі потрібно встановити значення *MAILSLOT\_WAIT\_FOREVER*.

У лістингу наведена програма, в якій функція *SetMailslotInfo* використовується для установки нового часового інтервалу для очікування надходження повідомлення в поштову скриньку.

### Зміна часу очікування повідомлення

#include <windows.h>

#include <iostream>

using std::cerr;

using std::cout;

using std::wcin;

using std::endl;

int main(int argc, char \*argv[])

{

HANDLE hMailslot; // дескриптор поштової скриньки

DWORD dwReadTimeout; // інтервал для очікування повідомлення

// створюємо поштовий ящик

hMailslot = CreateMailslotW(

L"\\\\.\\mailslot\\demo\_mailslot", // ім'я поштової скриньки

0, // довжина довільна повідомлення

0, // інтервал очікування дорівнює нулю

nullptr // захист за замовчуванням

);

// перевіряємо на успішне створення

if (hMailslot == INVALID\_HANDLE\_VALUE)

{

cerr << "Create mailslot failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

cout << "The mailslot is created." << endl;

// отримуємо інформацію про поштовій скриньці

if (

!GetMailslotInfo(

hMailslot, // дескриптор поштової скриньки

nullptr, // максимальний розмір повідомлення не потрібен

nullptr, // розмір такого повідомлення не потрібен

nullptr, // кількість повідомлень не потрібно

&dwReadTimeout// інтервал очікування повідомлення

)

)

{

cerr << "Get mailslot info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

cout << "Read timeout:" << dwReadTimeout << endl;

if (

!SetMailslotInfo(

hMailslot, // дескриптор поштової скриньки

3000// змінюємо інтервал очікування

)

)

{

cerr << "Set mailslot info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

// отримуємо інформацію про поштовій скриньці

if (

!GetMailslotInfo(

hMailslot, // дескриптор поштової скриньки

nullptr, // максимальний розмір повідомлення не потрібен

nullptr, // розмір такого повідомлення не потрібен

nullptr, // кількість повідомлень не потрібно

&dwReadTimeout// інтервал очікування повідомлення

)

)

{

cerr << "Get mailslot info failed." << endl

<< "The last error code:" << GetLastError() << endl;

cout << "Press any key to finish server.";

wcin.get();

return 0;

}

cout << "Read timeout:" << dwReadTimeout << endl;

// закриваємо дескриптор поштової скриньки

CloseHandle(hMailslot);

// завершуємо процес

cout << "Press any key to exit.";

wcin.get();

return 0;

}

# Висновки

Отож, є такі механізми, для обміну даними між процесами:

Механізми обміну повідомленнями є способом координації в конкурентних, паралельних, та об'єктно-орієнтованих системах, та організації взаємодії між процесами. Координація робиться шляхом відсилання повідомлень отримувачу. Повідомлення можуть мати форму викликів функцій, сигналів, та пакетів даних. До найвідоміших моделей обчислень, основаних на обміні повідомленнями належать модель Актора та числення процесів. Мікроядерні операційні системи координують роботу ядра та програм обміном повідомлень.

Механізми синхронізації – приведення двох або декількох процесів або потоків (нитей) до такого їхнього протікання, коли певні стадії різних процесів здійснюються в певному порядку, або одночасно, для уникнення конкуренції потоків або взаємного блокування. Загальна ідея полягає в тому, що в певних точках процесам необхідно разом домовитися про певний порядок дій зі спільними ресурсами.

Механізми розподілу пам'яті – існує набір способів управління пам'яттю пам'яттю комп'ютера, що здійснюється менеджером пам'яті. Головною задачею управління пам'яттю є надання можливості для динамічного виділення ділянок пам'яті для програм на їх прохання, та її звільнення для повторного використання, коли вона вже не потрібна. Це є важливим для будь-якого складної комп’ютерної системи, де може більше одного активного процесу працюючих одночасно.

Механізми віддалених викликів (RPC) – протокол, що дозволяє програмі, запущеній на одному комп'ютері бути викликаною на іншому комп'ютері без написання безпосередньо коду для цієї операції.

Для того щоб вибрати оптимальний механізм потрібно оцінити продуктивность різних механізмів обміну даними між паралельними процесами, використавши наступні параметри:

* пропускна здатність – кількість повідомлень в одиницю часу, яке ядро ОС або процес здатний обробити;
* затримки (час між відправленням повідомлення одним потоком і його отриманням іншим потоком).

# Перелік використаних джерел

* К. Р. ФІНОГЕНОВ «WIN32 основи програмування» друге видання, виправлене і доповнене МОСКВА ДІАЛОГ-МИФИ 2006
* Олександр Побегайло «системне програмування у WINDOWS» Санкт-Петербург «ПХВ-Петербург» 2006
* https://msdn.microsoft.com – Мережа Розробників Майкрософт