МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра прикладної математики

ФАЙЛОВА АБСТРАКЦІЯ В ОС WINDOWS

методичні рекомендації до виконання лабораторної роботи №6

із дисципліни:

«Операційні системи»

Київ — 2015

Зміст

ВСТУП

У традиційному розумінні, *файл* — це іменована область даних у зовнішній пам’яті. В операційній системі (ОС) Windows та у Windows API, під файлом розуміють як традиційні файли, так і комунікаційні ресурси, дискові пристрої, консолі, іменовані конвеєри, поштові ящики тощо. Різні типи файлів подібні один на одний, але відрізняються в окремих властивостях та областях застосування.

Windows API надає програмісту функції, які дозволяють аплікаціям отримувати доступ до файлів (у широкому сенсі слова) незалежно від файлової системи та пристрою зберігання файлів.

У даній роботі розглядатимемо засоби роботи з двома видами файлової абстракції — іменованими конвеєрами та поштовими ящиками — в ОС Windows версії 2000 та вище.

1 ПОСТАНОВКА ЗАДАЧІ

У даній лабораторній роботі потрібно ознайомитися з концепцією файлової абстракції в ОС Windows, навчитися створювати аплікації, які для обміну даними використовують засоби Windows API для роботи з іменованими конвеєрами або поштовими ящиками.

У рамках виконання лабораторної роботи потрібно:

а) ознайомитися з теоретичними відомостями, викладеними в розділі 2;

б) написати за допомогою Windows API будь-якою мовою програмування дві аплікації:

1) сервер поштового ящика або сервер іменованого конвеєра (на вибір);

2) клієнт поштового ящика або клієнт іменованого конвеєра, відповідно;

сервер та клієнт повинні використовувати тільки засоби Windows API і не повинні використовувати інші можливості систем програмування;

в) відлагодити програму в ОС Windows версії 2000 та вище;

г) підготувати звіт із лабораторної роботи відповідно до вимог розділу 3.

Кожен студент повинен написати сервер та клієнт вибраної файлової абстракції для виконання деякої конкретної задачі. Сервери та клієнти файлової абстракції різних студентів не можуть мати однакові функціональні можливості.

2 ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1 Загальна інформація про файлову абстракцію в ОС Windows

Концепцію *файлової абстракції* вперше було запропоновано в ОС Unix, у якій доступ до об'єктів різної природи (файлів на диску, дискових пристроїв, сокетів і навіть ділянок оперативної пам'яті) здійснюють за допомогою уніфікованих функцій. Ураховуючи практичну зручність цієї концепції, ідею файлової абстракції також імплементовано в інших ОС, зокрема, в ОС Windows.

Файлова абстракція у Windows включає в себе такі типи об'єктів:

- традиційні файли;
- файлові потоки;
- каталоги;
- фізичні та логічні диски;
- консолі;
- пристрої та драйвери пристроїв;
- комунікаційні ресурси (фізичні чи логічні пристрої для забезпечення двонаправленого асинхронного потоку даних, такі як порти, факсові машини та модеми);
- конвеєри;
- поштові ящики.

Варто зазначити, що в результаті багаторічного розвитку Unix і Windows та багаторазових спроб досягнути уніфікації різних файлових абстракцій вдалося досягти тільки часткового успіху. Оскільки процедура підготовки файлових об'єктів різної природи специфічна для кожного

класу об’єктів, повністю уніфікувати її не вдалося. Разом із тим, для виконання низки ключових операцій (зокрема, створення об’єктів, передачі даних тощо) Windows API пропонує уніфіковані засоби, які розглядатимемо в даній роботі на прикладі іменованих конвеєрів та поштових ящиків.

Спочатку розглянемо індивідуальні особливості використання іменованих конвєерів та поштових ящиків, а потім розглянемо уніфіковані функції Windows API, які можна використовувати для роботи з обома файловими абстракціями.

2.2 Іменовані конвеєри в ОС Windows

2.2.1 Загальні відомості

Конвеєр (pipe) — це ділянка спільної пам’яті, яку процеси використовують для спілкування між собою. Процес, який створює конвеєр, називають *сервером конвеєра* (pipe server). Процес, який під’єднують до конвеєра, називають *клієнтом конвеєра* (pipe client).

Англомовний термін *pipe* («труба» в дослівному перекладі) вказує на те, що конвеєр можна використовувати як інформаційний трубопровід. Якщо конвеєр односторонній, то один процес може записати дані в конвеєр («трубу») з одного кінця, а деякий інший процес — зчитати їх з іншого кінця. Якщо ж конвеєр двосторонній (*дуплексний*), то процеси можуть як записувати, так і зчитувати дані з обох кінців конвеєра.

В ОС Windows є два типи конвеєрів:

- *анонімні конвеєри* (anonymous pipes) — неіменовані односторонні конвеєри, які в основному використовують для передачі даних між батьківським та дочірнім процесами. Анонімні конвеєри завжди локальні, і

їх не можна використовувати для зв'язку мережею;

- *іменовані конвеєри* (named pipes) — односторонні чи двосторонні конвеєри для спілкування між сервером конвеєра та одним чи декількома його клієнтами. Будь-які процеси можуть отримувати доступ до іменованих конвеєрів (з урахуванням обмежень безпеки), що робить ці конвеєри зручним способом спілкування між спорідненими або розрізненими процесами (у тому числі мережею).

Застосування анонімних конвеєрів потребує менших зусиль з боку ОС, але іменовані конвеєри мають ширше коло застосування. Оскільки Windows API надає засоби роботи з файловими абстракціями тільки для іменованих конвеєрів, розгляд анонімних конвеєрів виходить за межі даних методичних рекомендацій.

Усі екземпляри (instances) іменованого конвеєра мають однакове ім'я, але кожний екземпляр має свій власний буфер і дескриптор та надає окремий «трубопровід» для спілкування сервера й клієнта. Наявність декількох екземплярів одного іменованого конвеєра дозволяє одночасно використовувати конвеєр декільком клієнтам.

Кожний процес може виступати як у ролі сервера, так і в ролі клієнта, що дозволяє реалізувати рівноправне спілкування між процесами.

У найпростішому випадку, життєвий цикл конвеєра виглядає наступним чином:

- сервер створює екземпляр конвеєра (див. розділ 2.2.2);
- клієнт під'єднується до цього екземпляра (див. розділ 2.4.1);
- сервер установлює з'єднання з клієнтом (див. розділ 2.2.5);
- сервер спілкується з клієнтом (див. розділи 2.4.2–2.4.3);
- сервер від'єднується від клієнта (див. розділ 2.2.6);
- сервер закриває дескриптор створеного раніше екземпляра конве-

єра (див. розділ 2.2.6).

2.2.2 Створення екземпляра конвеєра на сервері

Сервер конвеєра може створювати один чи більше екземплярів іменованого конвеєра за допомогою функції `CreateNamedPipe`. Формат цієї функції наступний:

```
HANDLE WINAPI CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Розгляньмо кожен із аргументів цієї функції детальніше:

- `lpName` — рядок, що визначає унікальне *ім'я* конвеєра, яке відрізняє його від інших іменованих конвеєрів в ОС. Рядок повинен мати наступний формат:

`\\.\pipe\`*pipename*,

де *pipename* — безпосередньо, ім'я конвеєра. Ім'я конвеєра може включати будь-які символи, окрім зворотного слеша, і повинно містити не більше

за 256 символів. Імена конвеєрів не чутливі до регістра. Клієнти конвеєра використовують це ім'я для під'єднання до нього за допомогою функції `CreateFile` (див. розділ 2.4.1);

- `dwOpenMode` — *режим відкриття конвеєра* (pipe open mode). Ми детальніше розглянемо різні режими відкриття конвеєра в розділі 2.2.3.

- `dwPipeMode` — *режим конвеєра* (pipe mode). Ми детальніше розглянемо різні режими відкриття конвеєра в розділі 2.2.4.

- `nMaxInstances` — максимальна кількість екземплярів, яку можна створити для цього конвеєра. Під час створення першого екземпляра це значення задають уперше. Під час створення усіх наступних екземплярів це значення не може змінитися. Значення `nMaxInstances` повинно лежати в проміжку від 1 до `PIPE_UNLIMITED_INSTANCES` (255). У разі використання значення `PIPE_UNLIMITED_INSTANCES` кількість екземплярів обмежено ресурсами ОС. Якщо значення `nMaxInstances` перевищить `PIPE_UNLIMITED_INSTANCES`, то функція поверне `INVALID_HANDLE_VALUE`;

- `nOutBufferSize` — кількість байтів, зарезервована під вихідний буфер;

- `nInBufferSize` — кількість байтів, зарезервована під ухідний буфер;

- `nDefaultTimeOut` — кількість часу, у мілісекундах, протягом якого функція `WaitNamedPipe` (див. розділ 2.2.5) ждатиме відповіді від конвеєра. Якщо `nDefaultTimeOut` установити в 0, буде використано значення в 50 мс;

- `lpSecurityAttributes` — указівник на структуру `SECURITY_ATTRIBUTES`, яка визначає безпековий дескриптор для новостворюваного конвеєра та визначає, чи можуть дочірні процеси наслідувати де-

скриптор конвеєра. Детальний розгляд цього аргумента виходить за рамки даних методичних рекомендацій. У даній роботі достатньо встановлювати цей аргумент у `NULL`.

У випадку успішного завершення, функція повертає дескриптор створеного екземпляра іменованого конвеєра (на стороні сервера). У випадку помилки функція повертає `INVALID_HANDLE_VALUE`. Деталізацію помилки можна отримати за допомогою функції `GetLastError`. Ця функція не має аргументів і повертає код останньої помилки, що мала місце в системі.

2.2.3 Режими відкриття конвеєра

Як під час створення екземпляра іменованого конвеєра на сервері, так і під час під'єднання до конвеєра з боку клієнта (див. розділ 2.4.1) можна задавати режим відкриття конвеєра, який, серед іншого, визначає:

- режим доступу (access mode);
- асинхронний режим (overlap mode);
- режим наскрізного запису (write-through mode).

Усі екземпляри одного й того ж іменованого конвеєра повинні мати однаковий *режим доступу*. Задавання режиму доступу до конвеєра має ефект, аналогічний задаванню права доступу читання/запису, пов'язаного з дескриптором даного конвеєра, із боку клієнта (див. розділ 2.4.1). Таблиця 2.1 наводить перелік можливих режимів доступу, еквівалентні їм права доступу читання/запису та короткий коментар.

Таблиця 2.1 – Режими доступу до іменованого конвеєра

Режим доступу	Право доступу	Коментар
`PIPE_ACCESS_INBOUND`	`GENERIC_READ`	Сервер може тільки читати з конвеєра, клієнт може тільки писати в конвеєр
`PIPE_ACCESS_OUTBOUND`	`GENERIC_WRITE`	Сервер може тільки писати в конвеєр, клієнт може тільки чисати з конвеєра
`PIPE_ACCESS_DUPLEX`	`GENERIC_READ | GENERIC_WRITE`	І сервер, і клієнт можуть і писати в конвеєр, і читати з нього

Коли клієнтів під'єднують до екземпляра конвеєра за допомогою функції `CreateFile` (див. розділ 2.4.1), вони повинні задавати права доступу читання/запису, еквівалентні праву доступу до екземпляра конвеєра, заданому на момент його створення.

В *асинхронному режимі* роботи конвеєра функції, що виконують довготривалі операції читання, записування чи з'єднання, повертають керування миттєво. Це дозволяє потоку, що виконує відповідну тривалу операцію, не бути заблокованим. Для того, щоб задати асинхронний режим конвеєра, потрібно використати прапорець `FILE_FLAG_OVERLAPPED` в аргументі `dwOpenMode` функції `CreateNamedPipe` (див. розділ 2.2.2). У даній роботі достатньо реалізувати тільки синхронний режим роботи конвеєра.

Режим наскрізного запису впливає на операції запису в конвеєри байтового типу (див. розділ 2.2.4) під час зв'язку клієнта та сервера, розташованих на різних комп'ютерах. У цьому режимі, функції запису в конвеєр не повертають керування, поки дані не буде передано мережею в буфер конвеєра машини-приймача (фактично, синхронний режим запису). Якщо режим наскрізного запису не встановлено, то функція запису поверне керування після запису даних у вихідний буфер конвеєра-передавача, тобто до моменту їх надсилання мережею.

Для того, щоб задати режим наскрізного запису для конвеєра, потрібно використати прапорець `FILE_FLAG_WRITE_THROUGH` в аргументі `dwOpenMode` функції `CreateNamedPipe` (див. розділ 2.2.2). Режим наскрізного запису для дескриптора екземпляра конвеєра неможливо змінити після його створення. Режим наскрізного запису може бути різним для сервера та для клієнта одного й того самого екземпляра конвеєра.

2.2.4 Режими конвеєра

Як під час створення екземпляра іменованого конвеєра на сервері, так і під час під'єднання до конвеєра з боку клієнта (див. розділ 2.4.1) можна задавати режим конвеєра, який, серед іншого, визначає:

- режим типу конвеєра (type mode);
- режим читання (read mode);
- режим очікування (wait mode).

Режим типу конвеєра визначає, яким чином дані записують у конвеєр:

- *режим байтового типу*. У цьому режимі дані записують у кон-

веєр як потік сирих байтів, і ОС не розрізняє між байтами, записаними під час виконання різних операцій записування. Щоб задати цей режим, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_TYPE_BYTE` або використати значення за замовчуванням;

- *режим повідомленнєвого типу*. У цьому режимі ОС розглядає байти, записані під час кожної операції записування, як окреме повідомлення. ОС завжди виконує операції записування в конвеєри повідомленнєвого типу в режимі наскрізного запису. Щоб задати режим повідомленнєвого типу, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_TYPE_MESSAGE`.

Усі екземпляри одного й того ж іменованого конвеєра повинні мати однаковий режим типу.

Режим читання конвеєра визначає, яким чином дані зчитують із конвеєра:

- *режим байтового читання*. У цьому режимі дані зчитують як потік сирих байтів. Операція зчитування успішно завершується, коли всі наявні в конвеєрі байти зчитано (або коли зчитано зазначену в операції кількість байтів). Конвеєр байтового типу може працювати виключно в режимі байтового читання. Щоб задати цей режим читання, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_READMODE_BYTE` або використати значення за замовчуванням;

- *режим повідомленнєвого читання*. У цьому режимі дані зчитують як потік повідомлень. Операція зчитування успішно завершується тільки тоді, коли все повідомлення зчитано. Якщо зазначена в операції зчитування кількість байтів менша за розмір чергового повідомлен-

ня, функція зчитує вказане число байтів із повідомлення та повертає 0 (див. розділ 2.4.2). Залишок повідомлення можна зчитати під час іншої операції зчитування. Конвеєр повідомленнєвого типу може працювати як у режимі байтового читання, так і в режимі повідомленнєвого читання, при цьому режим читання може бути неоднаковим для клієнта та сервера. Щоб задати цей режим читання, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_READMODE_MESSAGE`.

Режим очікування конвеєра визначає, яким чином функції читання, записування та з'єднання опрацьовують довготривалі операції в окремих випадках (див. розділи 2.2.5, 2.4.2, 2.4.3):

- *режим блокування очікування*. У цьому режимі відповідні функції не повертають керування, поки відповідну операцію не буде завершено. Щоб задати цей режим очікування, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_WAIT` або використати значення за замовчуванням;

- *режим неблокування очікування*. У цьому режимі відповідні функції повертають керування миттєво, навіть якщо відповідну операцію не завершено. Щоб задати цей режим очікування, потрібно в якості аргумента `dwPipeMode` функції `CreateNamedPipe` (див. розділ 2.2.2) задати значення `PIPE_NOWAIT`.

Клієнти та сервери конвеєра можуть змінювати режим відповідного екземпляра конвеєра під час його роботи за допомогою функції `SetNamedPipeHandleState`. Формат цієї функції наступний:

```
BOOL WINAPI SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,
```

```
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout
);
```

Аргумент `hNamedPipe` цієї функції визначає дескриптор екземпляра конвеєра, режими якого потрібно змінити, аргумент `lpMode` — новий режим конвеєра, який може складатися з комбінації прапораців режиму читання та режиму очікування, описаних вище, аргумент `lpMaxCollectionCount` — максимальна кількість байтів, які потрібно зібрати на машині клієнта перед їх надсиланням серверу (повинна дорівнювати `NULL`, якщо `hNamedPipe` відповідає дескриптору екземпляра на сервері чи якщо сервер і клієнт знаходяться на одній машині), аргумент `lpCollectDataTimeout` — кількість часу, у мілісекундах, який може сплинути до момент повного надсилання даних мережею (повинна дорівнювати `NULL`, якщо `hNamedPipe` відповідає дескриптору екземпляра на сервері чи якщо сервер і клієнт знаходяться на одній машині).

У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

2.2.5 З'єднання клієнта конвеєра з сервером

Після успішного створення екземпляра іменованого конвєера на сервері клієнт може під'єднатися до конвеєра за допомогою функції `CreateFile` (див. розділ 2.4.1). Якщо в ОС є вільний екземпляр відповідного іменованого конвеєра, ця функція поверне клієнту дескриптор відповідного екземпляра. У протилежному випадку клієнт може використати фун-

кцію WaitNamedPipe, щоб діждатися звільнення екземпляра потрібного конвеєра. Формат цієї функції наступний:

```
BOOL WINAPI WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
    DWORD nTimeOut
);
```

Аргумент lpNamedPipeName визначає рядок, формат якого аналогічний формату аргумента lpName функції CreateNamedPipe (див. розділ 2.2.2), аргумент nTimeOut — кількість часу, у мілісекундах, протягом якого функція ждатиме відповіді від конвеєра. Якщо протягом цього часу жодний екземпляр конвеєра не буде звільнено, функція поверне 0. У протилежному випадку функція поверне дескриптор вільного екземпляра конвеєра.

Якщо в якості аргумента nTimeOut використати значення NMPWAIT_USE_DEFAULT_WAIT, то функція ждатиме відповіді від конвеєра стільки мілісекунд, скільки вказано в аргументі nDefaultTimeOut функції CreateNamedPipe.

Сервер конвеєра може визначити, що клієнт під'єднався до екземпляра конвеєра, за допомогою функції ConnectNamedPipe. Формат цієї функції наступний:

```
BOOL WINAPI ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

Аргумент hNamedPipe функції визначає дескриптор екземпляра іменованого конвеєра, аргумент lpOverlapped — указівник на стру-

ктуру OVERLAPPED, який не може бути NULL, якщо конвеєр створено з прапорцем FILE_FLAG_OVERLAPPED (див. розділ 2.2.3). У даній роботі достатньо реалізувати конвеєр, який працює тільки в синхронному режимі, тому розгляд цієї структури виходить за рамки даних методичних рекомендацій. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

Якщо клієнт під'єднується до конвеєра до того, як сервер викличе функцію ConnectNamedPipe, ця функція поверне 0, а виклик функції GetLastError поверне ERROR_PIPE_CONNECTED. Це може статися в тому випадку, коли клієнт під'єднується після створення екземпляра конвеєра за допомогою функції CreateNamedPipe, але до виклику функції ConnectNamedPipe. Незважаючи не те, що функція ConnectNamedPipe повертає 0, у цьому випадку вважається, що зв'язок між сервером і клієнтом встановлено успішно.

Якщо екземпляр конвеєра створено в режимі блокування очікування (див. розділ 2.2.4), то функція ConnectNamedPipe не поверне керування, поки до екземпляра не буде під'єднано деякий клієнт.

Після встановлення з'єднання між сервером та клієнтом у вказаний вище спосіб вони можуть обмінюватися даними за допомогою функції Windows API для роботи з файловими абстракціями ReadFile та WriteFile, які буде розглянуто в розділах 2.4.2–2.4.3.

2.2.6 Завершення роботи з екземпляром конвеєра

Після завершення роботи з екземпляром конвеєра, сервер повинен викликати функцію очищення буфера FlushFileBuffers, щоб пере-

конатися, що в конвеєрі не залишилося непрочитаних байтів або повідомлень. Формат цієї функції наступний:

```
BOOL WINAPI FlushFileBuffers(
    HANDLE hFile
);
```

Аргумент `hFile` цієї функції визначає дескриптор відкритого абстрактного файлу (у даному разі — екземпляра іменованого конвеєра). У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0. Варто зазначити, що функція не повертає керування, поки клієнт не прочитає всі дані з конвеєра.

Після успішного повернення з функції `FlushFileBuffers`, сервер повинен викликати функцію `DisconnectNamedPipe`, щоб розірвати зв'язок із клієнтом. Формат цієї функції наступний:

```
BOOL WINAPI DisconnectNamedPipe(
    HANDLE hNamedPipe
);
```

Аргумент `hNamedPipe` цієї функції визначає дескриптор екземпляра іменованого конвеєра, від якого потрібно від'єднати клієнта. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

У випадку успішного завершення функції `DisconnectNamedPipe`, дескриптор екземпляра конвеєра на клієнтській стороні стає невалідним, а всі недочитані дані стають утраченими.

Після від'єднання клієнта, сервер може закрити дескриптор екземпляра конвеєра або повторно викликати функцію `ConnectNamedPipe` (див. розділ 2.2.5) для з'єднання з іншим клієнтом.

Для закриття дескриптора сервер повинен викликати функцію `CloseHandle`. Формат цієї функції наступний:

```
BOOL WINAPI CloseHandle(
    HANDLE hObject
);
```

Аргумент `hObject` цієї функції визначає дескриптор, який потрібно закрити. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

2.2.7 Організація роботи сервера конвеєра з декількома клієнтами

До цього моменту ми розглядали загальні принципи роботи з конвеєром у випадку одного сервера та одного клієнта. Проте, на практиці значно частіше зустрічається ситуація, коли один сервер конвеєра повинен спілкуватися з декількома клієнтами.

Найпростіший варіант реалізації взаємодії одного сервера з декількома клієнтами — використовувати один екземпляр конвеєра, почергово під'єднуючи до нього той чи інший клієнт. Цілком очевидно, що такий підхід матиме низьку продуктивність. Ліпшим є підхід, за якого сервер конвеєра створює декілька екземплярів для того, щоб можна було одночасно спілкуватися з декількома клієнтами.

Існує дві базові стратегії обслуговування декількох екземплярів конвеєра:

- створювати для кожного нового екземпляра конвеєра окремий потік (фактично, використання конвеєра в синхронному режимі);

- використовувати конвеєр в асинхронному режимі, працюючи в одному потоці.

Аплікації, які використовують першу стратегію, простіше писати, але при цьому потрібно витрачати ресурси ОС для підтримки декількох потоків, що може бути витратно у випадку надто великої кількості клієнтів. Під час використання єдиного потоку простіше координувати операції спілкування з різними клієнтами та забезпечувати захист спільних ресурсів. Проте, такий підхід вимагає окремих зусиль для координації асинхронних операцій. У даній роботі достатньо реалізувати тільки іменований конвеєр, який працює в синхронному режимі.

2.3 Поштові ящики

2.3.1 Загальні відомості

Поштовий ящик (mailslot) — механізм односторонньої міжпроцесової взаємодії, за якого аплікація — власник ящика може отримувати повідомлення від інших аплікацій. Як правило, цей механізм використовують для спілкування процесів мережею (але необов'язково).

У той час як іменовані конвеєри (див. розділ 2.2) є зручним способом спілкування двох процесів, поштові ящики дозволяють одному процесу передавати повідомлення декільком процесам одночасно. При цьому, повідомлення з використаннями поштових ящиків передають за допомогою *датаграм* (datagrams) — малих пакетів даних. На відміну від надсилання даних конвеєром, під час надсилання датаграм неможливо отримати підтвердження про успішне надходження датаграми адресату. У зв'язку з цим, використання іменованих конвеєрів можна порівняти з використан-

ням телефону (спілкуються тільки двоє, але завжди можна дізнатися, чи дійшло потрібне повідомлення до адресата), а використання поштових ящиків — із використанням телебачення (декілька адресатів можуть дивитися телевізор одночасно, але якщо деякому глядачеві сигнал не доходить, то адресант про це дізнатися не має можливості).

Поштовий ящик — це файлова абстракція, яка постійно знаходиться в оперативній пам'яті. Дані в поштовому ящику можуть мати довільний формат, але між різними комп'ютерами за один раз можна переслати до 424 байтів. Для пересилання більшої кількості байтів потрібно використовувати інші механізми, наприклад, іменовані конвеєри.

На відміну від традиційних файлів, поштові ящики тимчасові: коли всі дескриптори ящика закрито, ОС видаляє його та дані, які він містить.

Процес, який створює поштовий ящик та володіє ним, називають *сервером поштового ящика* (mailslot server). Після створення ящика сервер отримує його дескриптор, який може використовувати для читання повідомлень із ящика. Процес не може створити віддалений поштовий ящик: усі ящики локальні відносно процесу, який їх створює.

Процес, який надсилає дані в поштовий ящик, називають *клієнтом поштового ящика* (mailslot client). Будь-який процес, який знає назву поштового ящика, може надсилати в нього дані. Нові повідомлення дописують у поштовий ящик після вже наявних у ньому. Сервер поштового ящика може одночасно виступати в ролі клієнта цього ж ящика.

Декілька процесів у рамках одного домена (але на різних комп'ютерах) можуть одночасно створити поштові ящики з однаковим іменем. У цьому випадку, повідомлення, надіслане в ящик із відповідним іменем, отримають усі процеси, які створили ящики з аналогічним іменем. Оскільки сервер поштового ящика може також записувати в нього дані, та-

кий підхід дозволяє організувати ефективну комунікацію між декількома процесами в рамках одного домена.

У найпростішому випадку, життєвий цикл поштового ящика виглядає наступним чином:

- сервер створює поштовий ящик (див. розділ 2.3.2);
- клієнт отримує дексриптор ящика для надсилання в нього повідомлення (див. розділ 2.4.1);
- клієнт надсилає в ящик повідомлення (див. розділ 2.4.3);
- сервер зчитує з ящика надіслане повідомлення (див. розділ 2.4.2);
- клієнт закриває дескриптор поштового ящика (див. розділ 2.3.4);
- сервер закриває дескриптор поштового ящика (див. розділ 2.3.4).

2.3.2 Створення поштового ящика на сервері

Сервер конвеєра може створити поштовий ящик за допомогою функції `CreateMailslot`. Формат цієї функції наступний:

```
HANDLE WINAPI CreateMailslot(
    LPCTSTR lpName,
    DWORD nMaxMessageSize,
    DWORD lReadTimeout,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Розгляньмо кожен із аргументів цієї функції детальніше:

- `lpName` — рядок, що визначає унікальне *ім'я* поштового ящика. Рядок повинен мати наступний формат:

`\\.\mailslot\`*[path]name*,

де *name* — безпосередньо, ім'я поштового ящика. Повне ім'я поштового ящика може включати декілька псевдокаталогів, розділених зворотними слешами. Приклади валідних імен поштових ящиків:

`\\.\mailslot\example`

`\\.\mailslot\abc\def\ghi`;

- `nMaxMessageSize` — максимальний розмір, у байтах, повідомлення, яке можна записати в поштовий ящик. Якщо в якості аргумента задати 0, повідомлення зможуть мати довільний розмір;

- `lReadTimeout` — кількість часу, у мілісекундах, протягом якого функція зчитування з поштового ящика (див. розділ 2.4.2) ждатиме, поки повідомлення буде записано в ящик. Якщо в якості аргумента задати 0, функція зчитування поверне керування миттєво. Якщо в якості аргумента задати значення `MAILSLOT_WAIT_FOREVER`, функція зчитування ждатиме появи в ящику повідомлення довільну кількість часу;

- `lpSecurityAttributes` — указівник на структуру `SECURITY_ATTRIBUTES`, яка визначає безпековий дескриптор для новостворюваного поштового ящика та визначає, чи можуть дочірні процесу наслідувати дескриптор поштового ящика. У даній роботі достатньо встановлювати цей аргумент у `NULL`.

У випадку успішного завершення, функція повертає дескриптор створеного поштового ящика. У випадку помилки функція повертає `INVALID_HANDLE_VALUE`. Деталізацію помилки можна отримати за допомогою функції `GetLastError`.

2.3.3 Визначення властивостей поштового ящика після його створення

Після створення поштового ящика сервер може отримувати за допомогою функції `GetMailslotInfo` інформацію про властивості поштового ящика, потрібні для організації зчитування повідомлень із нього, зокрема, кількість повідомлень у ящику та розмір чергового повідомлення. Формат цієї функції наступний:

```
BOOL WINAPI GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpMessageCount,
    LPDWORD lpReadTimeout
);
```

Розгляньмо кожен із аргументів цієї функції детальніше:

- `hMailslot` — дескриптор поштового ящика, отриманий за допомогою функції `CreateMailslot` (див. розділ 2.3.2);

- `lpMaxMessageSize` — указівник на змінну, у яку буде записано максимальний розмір повідомлення, у байтах, яке можна записати в поштовий ящик. Це значення може перевищувати значення аргумента `nMaxMessageSize` функції `CreateMailslot` (див. розділ 2.3.2);

- `lpNextSize` — указівник на змінну, у яку буде записано розмір, у байтах, наступного повідомлення в ящику або значення `MAILSLOT_NO_MESSAGE`, якщо ящик порожній;

- `lpMessageCount` — указівник на змінну, у яку буде записано кількість повідомлень у ящику на момент виклику цієї функції;

- `lpReadTimeout` — указівник на змінну, у яку буде записано кількість часу, у мілісекундах, протягом якого функція зчитування з поштового ящика (див. розділ 2.4.2) ждатиме, поки повідомлення буде записано в ящик.

У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

2.3.4 Закриття дескриптора поштового ящика

Поштовий ящик припиняє своє існування у випадку настання однієї з двох подій:

- сервер закриває останній відкритий дескриптор поштового ящика за допомогою функції `CloseHandle` (див. розділ 2.2.6);

- процес, який володіє останнім відкритим дескриптором поштового ящика, припиняє своє виконання.

2.4 Засоби Windows API для роботи з файловими абстракціями

Windows API надає програмісту низку функцій для уніфікованої роботи з різними файловими абстракціями, зокрема, іменованими конвеєрами та поштовими ящиками, розглянутими в попередніх розділах. У даному розділі розглянемо такі функції Windows API:

- функцію для створення файлової абстракції `CreateFile`;
- функцію для читання з файлової абстракції `ReadFile`;
- функцію для писання у файлову абстракцію `WriteFile`.

Наприкінці розділу ми розглянемо приклади реалізації сервера та клієнта іменованого конвеєра та поштового ящика з використанням зазначених функції Windows API.

2.4.1 Використання функції `CreateFile`

Використання функції Windows API `CreateFile` дозволяє створювати або відкривати файлову абстракцію. Спочатку розглянемо загальні характеристики цієї функції, після чого проаналізуємо особливості її застосування до іменованих конвеєрів та поштових ящиків.

Функція має наступний формат:

```
HANDLE WINAPI CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

Розгляньмо аргументи цієї функції детальніше:

- `lpFileName` — ім’я файлової абстракції, яку потрібно створити

або відкрити;

- dwDesiredAccess — режим доступу до файлової абстракції. Найчастіше використовують такі значення: GENERIC_READ — створення (відкриття) файлової абстракції для читання, GENERIC_WRITE — для записування, GENERIC_READ | GENERIC_WRITE — для читання та записування одночасно;

- dwShareMode — режим спільного використання файлової абстракції. Розгляньмо можливі значення цього аргумента, потрібні для виконання даної роботи. Якщо в якості цього атрибута задати 0, то файлову абстракцію неможливо буде відкрити повторно, не закривши попередньо її дескриптор (корисно для іменованих конвеєрів). Якщо в якості цього атрибута задати FILE_SHARE_READ, то цю файлову абстракцію зможуть відкривати інші аплікації тільки для читання (корисно для поштових ящиків);

- lpSecurityAttributes — указівник на структуру SECURITY_ATTRIBUTES, яка визначає безпековий дескриптор для файлової абстракції та визначає, чи можуть дочірні процеси наслідувати дескриптор файлової абстракції. У даній роботі достатньо встановлювати цей аргумент у NULL;

- dwCreationDisposition — визначає дію, яку потрібно вчинити у випадку, якщо файлова абстракція з іменем lpFileName уже існує. У даній роботі в силу особливостей використання цієї функції (див. нижче) достатньо використовувати тільки значення OPEN_EXISTING, тобто функція відкриватиме тільки вже існуючу файлову абстракцію (якщо її не існує, то функція завершиться з помилкою);

- dwFlagsAndAttributes — атрибути файлової абстракції (див. нижче). У даній роботі достатньо використовувати значення 0 — набір

атрибутів за замовчуванням;

- `hTemplateFile` — валідний дескриптор файлової абстракції, яку буде використано як шаблон під час створення нової абстракції. У даній роботі в силу особливостей використання цієї функції (див. нижче) достатньо використовувати значення `NULL`.

У випадку успішного завершення, функція повертає дескриптор створеної або відкритої файлової абстракції. У випадку помилки функція повертає `INVALID_HANDLE_VALUE`.

Розгляньмо особливості застосування функції `CreateFile` до іменованих конвеєрів. Функцію `CreateFile` може викликати тільки клієнт конвеєра для під'єднання до існуючого екземпляра. У випадку наявності вільних екземплярів функція поверне дескриптор на будь-який із них. У випадку, коли немає жодного вільного екземпляра, функція поверне `INVALID_HANDLE_VALUE`, а виклик функції `GetLastError` поверне значення `ERROR_PIPE_BUSY`.

Повторно під'єднатися до екземпляра конвеєра не можна доти, доки його дескриптор не буде закрито за допомогою функції `CloseHandle` (див. розділ 2.2.6).

Під час використання функції `CreateFile` значення аргумента `dwDesiredAccess` повинно бути сумісне з режимом доступу до конвеєра (див. розділ 2.2.3).

Варто зазначити, що дескриптори, отримані в результаті застосування функції `CreateFile`, за замовчуванням мають режим байтового читання. Щоб змінити цей режим на режим повідомленнєвого читання, клієнт повинен викликати функцію `SetNamedPipeHandleState` (див. розділ 2.2.4).

Розгляньмо особливості застосування функції `CreateFile` до по-

штових ящиків. Функцію `CreateFile` може викликати тільки клієнт поштового ящика для отримання дескриптора існуючого ящика. Для відкриття поштового ящика на локальному комп'ютері, клієнт може використати в якості аргумента `lpFileName` рядок, повністю аналогічний тому, який було використано для створення ящика на сервері (див. розділ 2.3.2). У рамках виконання даної роботи достатньо реалізувати сервер та клієнт поштового ящика на одному комп'ютері.

Якщо клієнт здійснює спробу отримати дескриптор неіснуючого поштового ящика, функція `CreateFile` поверне `INVALID_HANDLE_VALUE`, а виклик функції `GetLastError` поверне значення `ERROR_FILE_NOT_FOUND`.

2.4.2 Використання функції `ReadFile`

Використання функції Windows API `ReadFile` дозволяє зчитувати дані з файлової абстракції. Спочатку розглянемо загальні характеристики цієї функції, після чого проаналізуємо особливості її застосування до іменованих конвеєрів та поштових ящиків.

Функція має наступний формат:

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

```
);
```

Розгляньмо аргументи цієї функції детальніше:

- `hFile` — дескриптор файлової абстракції;

- `lpBuffer` — указівник на буфер, у який буде записано прочитані дані. Буфер повинен залишатися валідним протягом виконання операції читання;

- `nNumberOfBytesToRead` — максимальне число байтів, яке потрібно прочитати;

- `lpNumberOfBytesRead` — указівник на змінну, у яку буде записано реально прочитану кількість байтів (для синхронного режиму читання);

- `lpOverlapped` — указівник на структуру `OVERLAPPED` (для асинхронного режиму читання) або `NULL` (для синхронного режиму читання).

У випадку успішного завершення, функція повертає ненульове значення. У випалку помилки функція повертає 0.

Розгляньмо особливості застосування функції `ReadFile` до іменованих конвеєрів. Якщо конвеєр працює в режимі блокування очікування (див. розділ 2.2.4), і на момент виклику функції `ReadFile` порожній, то функція не поверне керування, поки в конвеєрі не з'являться дані для прочитання. У режимі неблокування очікування функція в цьому ж випадку повертає значення 0 миттєво.

У випадку спроби викликати функцію `ReadFile` для прочитання даних з екземпляра конвеєра, дескриптор якого було закрито зі сторони запису, функція поверне 0, а виклик функції `GetLastError` поверне значення `ERROR_BROKEN_PIPE`. Це значення можна використовувати для виявлення ситуації, коли інша сторона конвеєра від'єдналася від нього.

Якщо конвеєр працює в режимі повідомленнєвого читання, і наступне повідомлення в конвеєрі перевищує за розміром значення параметра `nNumberOfBytesToRead`, то функція `ReadFile` зчитає тільки частину повідомлення і поверне 0. Решту повідомлення можна дочитати під час наступного виклику функції `ReadFile`.

Якщо функція `ReadFile` повертає ненульове значення, але `lpNumberOfBytesRead` вказує на число 0, то це означає, що під час виклику попереднього виклику функції `WriteFile` (див. розділ 2.4.3) її аргумент `nNumberOfBytesToWrite` було встановлено в 0.

Розгляньмо особливості застосування функції `ReadFile` до поштових ящиків. Незважаючи на те, що дескриптор поштового ящика, отриманий за допомогою функції `CreateFile` (див. розділ 2.4.1), за замовчуванням відповідає асинхронному режиму роботи поштового ящика, його можна також використовувати і в синхронному режимі. Для цього під час викликання функції `ReadFile` потрібно в якості аргумента `lpOverlapped` задавати `NULL`. У цьому випадку функція `ReadFile` не поверне керування, поки не буде отримано нове повідомлення для прочитання.

2.4.3 Використання функції `WriteFile`

Використання функції Windows API `WriteFile` дозволяє записувати дані у файлову абстракцію. Спочатку розглянемо загальні характеристики цієї функції, після чого проаналізуємо особливості її застосування до іменованих конвеєрів та поштових ящиків.

Функція має наступний формат:

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

Розгляньмо аргументи цієї функції детальніше:

- `hFile` — дескриптор файлової абстракції;

- `lpBuffer` — указівник на буфер, дані з якого буде записано у файлову абстракцію. Буфер повинен залишатися валідним протягом виконання операція читання;

- `nNumberOfBytesToWrite` — число байтів, яке потрібно записати;

- `lpNumberOfBytesWritten` — указівник на змінну, у яку буде записано реально записану кількість байтів (для синхронного режиму читання);

- `lpOverlapped` — указівник на структуру `OVERLAPPED` (для асинхронного режиму читання) або `NULL` (для синхронного режиму читання).

У випадку успішного завершення, функція повертає ненульове значення. У випалку помилки функція повертає 0.

Розгляньмо особливості застосування функції `WriteFile` до іменованих конвеєрів. Якщо конвеєр працює в режимі блокування очікування (див. розділ 2.2.4), і на момент виклику функції `WriteFile` у його буфері недостатньо місця, то функція не поверне керування, поки з конвеєра не буде зчитано достатню кількість даних, щоб вивільнити буфер. У режимі

неблокування очікування функція в цьому ж випадку повертає значення 0 миттєво і додатково:

- для конвеєрів повідомленнєвого типу — не записує в конвеєр жодного байта;

- для конвеєрів байтового типу — записує стільки байтів, скільки може вмістити буфер (при цьому значення `*lpNumberOfBytesWritten` буде менше за `nNumberOfBytesToWrite`).

2.4.4 Приклад реалізації іменованого конвеєра

Приклад 2.1.

Розгляньмо вихідний текст консольної програми Windows API на мові програмування C++, що реалізує сервер іменованого конвеєра.

У даній програмі організовано нескінченний цикл, у рамках якого сервер створює новий екземпляр іменованого конвеєра за допомогою функції `CreateNamedPipe` (див. розділ 2.2.2) та жде під'єднання нового клієнта. Із під'єднанням кожного нового клієнта сервер створює за допомогою функції `CreateThread` окремий потік для спілкування з ним. У рамках потоку сервер у циклі виконує такі дії:

- читає за допомогою функції `ReadFile` (див. розділ 2.4.2) повідомлення від клієнта;

- виводить його в консоль;

- надсилає клієнту за допомогою функції `WriteFile` (див. розділ 2.4.3) стандартну відповідь.

Виконання цього циклу повторюватиметься доти, доки клієнт не від'єднається від сервера, після чого потік припиняє своє виконання, очи-

щуючи за допомогою функції `FlushFileBuffers` буфер конвеєра, від'єднуючись за допомогою функції `DisconnectNamedPipe` від клієнта та закриваючи за допомогою функції `CloseHandle` дескриптор екземпляра конвеєра (див. розділ 2.2.6).

Результат роботи сервера під час спілкування з клієнтом із Прикладу 2.2 представлено на рисунку 2.1.

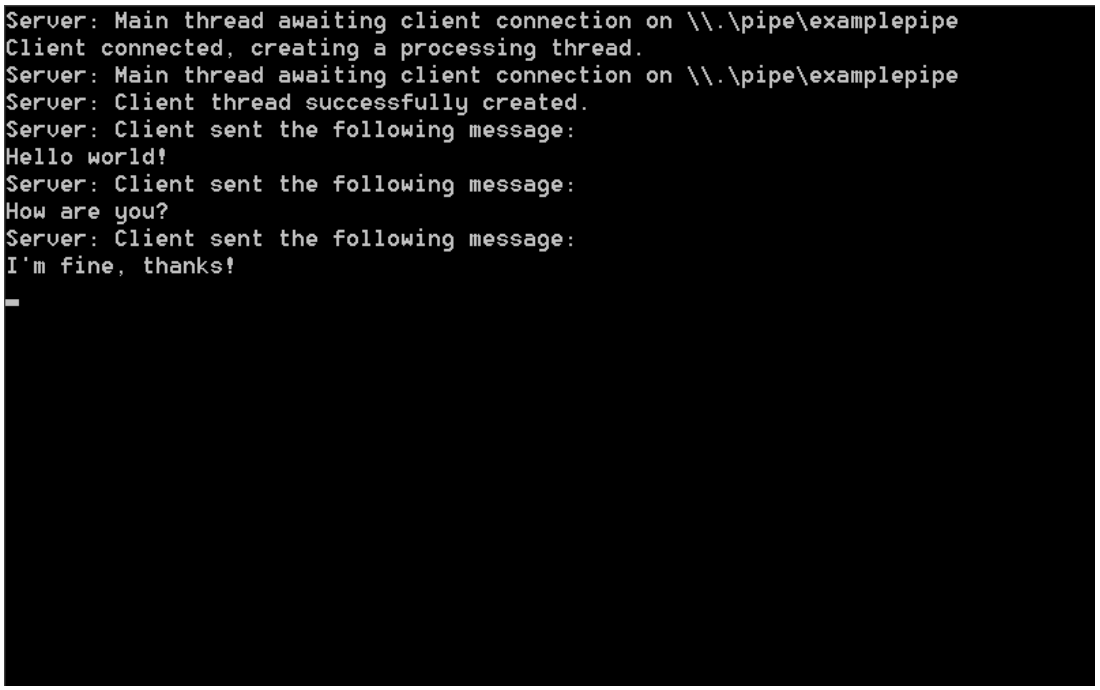

Рисунок 2.1 – Результат роботи сервера конвеєра з Прикладу 2.1

```
#include "stdafx.h"

#include <windows.h>
#include <strsafe.h>
#include <iostream>

using namespace std;

#define BUFSIZE 512

//ім'я конвеєра
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\examplepipe");

DWORD WINAPI InstanceThread(LPVOID);
VOID GetAnswerToRequest(LPTSTR, LPTSTR, LPDWORD);
```

```
int _tmain(int argc, TCHAR *argv[])
{
    BOOL fConnected = FALSE;
    DWORD  dwThreadId = 0;
    HANDLE hPipe = INVALID_HANDLE_VALUE;
    HANDLE hThread = NULL;

    for(;;)
    {
        cout << "Server: Main thread awaiting client connection on ";
        wcout << lpszPipename << endl;

        //створюємо екземпляр конвеєра
        hPipe = CreateNamedPipe(
            lpszPipename,             //ім'я конвеєра
            PIPE_ACCESS_DUPLEX,       //права доступу читання та запису
            PIPE_TYPE_MESSAGE |       //конвеєр повідомленнєвого типу
            PIPE_READMODE_MESSAGE |   //режим повідомленнєвого читання
            PIPE_WAIT,                //режим блокування очікування
            PIPE_UNLIMITED_INSTANCES, //максимальне число екземплярів необмежено
            BUFSIZE,                  //розмір вихідного буфера
            BUFSIZE,                  //розмір ухідного буфера
            0,                        //очікування з боку клієнта
            NULL);                    //атрибути безпеки за замовчуванням

        if(hPipe == INVALID_HANDLE_VALUE)
        {
            //якщо сталася помилка
            cout << "CreateNamedPipe failed, error code is = "
                << GetLastError() << "." << endl;

            return -1;
        }

        //ждемо, поки під'єднається клієнт
        fConnected = ConnectNamedPipe(hPipe, NULL);

        if(fConnected == 0)
        {
            //потрібно явно перевірити ситуацію, коли
            //ConnectNamedPipe повертає 0, проте
            //GetLastError повертає ERROR_PIPE_CONNECTED
            if(GetLastError() == ERROR_PIPE_CONNECTED)
            {
                fConnected = TRUE;
            }
        }

        if(fConnected)
        {
            //якщо клієнт успішно під'єднався
            cout << "Client connected, creating a processing thread." << endl;

            //створюємо потік для опрацювання цього клієнта
            hThread = CreateThread(
                NULL,           //атрибути безпеки непотрібні
                0,              //розмір стеку за замовчуванням
```

```
                InstanceThread, //указівник на локальну функцію
                (LPVOID) hPipe, //параметр цієї функції
                0,              //потік починає виконання відразу після створення
                NULL);          //нам непотрібний ідентифікатор потоку

            if(hThread == NULL)
            {
                //якщо сталася помилка
                cout << "CreateThread failed, error code is = "
                    << GetLastError() << "." << endl;

                return -1;
            }
            else{
                //закриваємо дескриптор потоку
                CloseHandle(hThread);
            }
        }
        else
        {
            //якщо сталася помилка
            //закриваємо дескриптор екземпляра конвеєра
            CloseHandle(hPipe);
        }
    }

    return 0;
}

//функція --- початкова адреса потоку
DWORD WINAPI InstanceThread(LPVOID lpvParam)
{
    //отримуємо дескриптор купи
    HANDLE hHeap = GetProcessHeap();

    //виділяємо пам'ять для повідомлень
    TCHAR* pchRequest = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE*sizeof(TCHAR));
    TCHAR* pchReply = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE * sizeof(TCHAR));

    DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;
    BOOL fSuccess = FALSE;
    HANDLE hPipe = NULL;

    //стандартна відповідь від сервера
    LPCTSTR lpDefaultReply = TEXT("Thanks for the message:\n");

    if(lpvParam == NULL)
    {
        //якщо потік було створено з невалідним дескриптором екземпляра конвеєра
        cout << "ERROR - Server:" << endl
            << "Unexpected NULL value for a pipe instance descriptor!" << endl
            << "Client thread exiting." << endl;

        //очищуємо пам'ять
        if(pchReply != NULL)
        {
            HeapFree(hHeap, 0, pchReply);
```

```
    }

    if(pchRequest != NULL)
    {
        HeapFree(hHeap, 0, pchRequest);
    }

    return 1;
}

if(pchRequest == NULL)
{
    //якщо сталася помилка
    cout << "ERROR - Server:" << endl
        << "Couldn't allocate heap memory for the client message!" << endl
        << "Client thread exiting." << endl;

    //закриваємо дескриптор екземпляра конвеєра
    CloseHandle(hPipe);

    //очищуємо пам'ять
    if(pchReply != NULL)
    {
        HeapFree(hHeap, 0, pchReply);
    }

    return 1;
}

if(pchReply == NULL)
{
    //якщо сталася помилка
    cout << "ERROR - Server:" << endl
        << "Couldn't allocate heap memory for the server reply!" << endl
        << "Client thread exiting." << endl;

    //закриваємо дескриптор екземпляра конвеєра
    CloseHandle(hPipe);

    //очищуємо пам'ять
    if(pchRequest != NULL)
    {
        HeapFree(hHeap, 0, pchRequest);
    }

    return 1;
}

cout << "Server: Client thread successfully created." << endl;

hPipe = (HANDLE) lpvParam;

while(1)
{
    //у циклі, поки клієнт не від'єднається

    //зчитуємо повідомлення від клієнта з конвеєра
```

```
fSuccess = ReadFile(
    hPipe,                  //дескриптор екземпляра конвеєра
    pchRequest,             //буфер, куди буде зчитано дані
    BUFSIZE * sizeof(TCHAR), //розмір буфера
    &cbBytesRead,           //кількість прочитаних байтів
    NULL);                  //синхронний режим

if(!fSuccess || cbBytesRead == 0)
{
    //якщо сталася помилка
    if(GetLastError() == ERROR_BROKEN_PIPE)
    {
        //якщо клієнт від'єднався
        cout << "Server: Client disconnected." << endl;
    }
    else
    {
        //в іншому випадку
        cout << "Server: Failed reading from the client end, error code is "
            << GetLastError() << "." << endl;
    }

    break;
}

cout << "Server: Client sent the following message:" << endl;
wcout << pchRequest << endl;

StringCchCopy(pchReply, BUFSIZE + lstrlen(lpDefaultReply) + 1, lpDefaultReply);

//перевіряємо, чи не перевищує повідомлення BUFSIZE
if(FAILED(StringCchCat(pchReply, BUFSIZE + lstrlen(lpDefaultReply) + 1,
    pchRequest)))
{
    //якщо перевищує
    cbReplyBytes = 0;
    pchReply[0] = 0;

    cout << "ERROR - Server: Reply message exceeds buffer size." << endl;
}
else{
    cbReplyBytes = (lstrlen(pchReply) + 1) * sizeof(TCHAR);
}

//записуємо відповідь у конвеєр
fSuccess = WriteFile(
    hPipe,        //дескриптор екземпляра конвеєра
    pchReply,     //буфер із даними
    cbReplyBytes, //кількість байтів для записування
    &cbWritten,   //кількість реально записаних байтів
    NULL);        //синхронний режим

if(!fSuccess || cbReplyBytes != cbWritten)
{
    //якщо сталася помилка
    cout << "ERROR - Server: Failed writing to the client, error code is "
        << GetLastError() << "." << endl;
```

```
            break;
        }
    }

    //очищуємо екземпляр конвеєра
    FlushFileBuffers(hPipe);

    //від'єднуємо клієнта від екземпляра конвеєра
    DisconnectNamedPipe(hPipe);

    //закриваємо дескриптор екземпляра конвеєра
    CloseHandle(hPipe);

    //очищуємо пам'ять
    HeapFree(hHeap, 0, pchRequest);
    HeapFree(hHeap, 0, pchReply);

    cout << "Server: Client thread exiting." << endl;

    return 0;
}
```

Приклад 2.2.

Розгляньмо вихідний текст консольної програми Windows API на мові програмування C++, що реалізує клієнт іменованого конвеєра.

Спочатку клієнт за допомогою функції `CreateFile` (див. розділ 2.4.1) під'єднується до екземпляра конвеєра, створеного в Прикладі 2.1. Після цього клієнт у циклі виконує такі дії:

- читає з консолі уведення від користувача;
- надсилає за допомогою функції `WriteFile` (див. розділ 2.4.3) серверу це уведення;
- читає за допомогою функції `ReadFile` (див. розділ 2.4.2) від сервера відповідь;
- виводить отриману відповідь у консоль.

Виконання цього циклу повторюватиметься доти, доки користувач не уведе рядок `"quit"`.

Варто зазначити, що в цій програмі не виконано перевірку на переповнення буфера під час уведення повідомлення з клавіатури. У роботах студентів відповідні перевірки повинно бути обов'язково передбачено.

Результат роботи клієнта під час спілкування з сервером із Прикладу 2.1 представлено на рисунку 2.2.

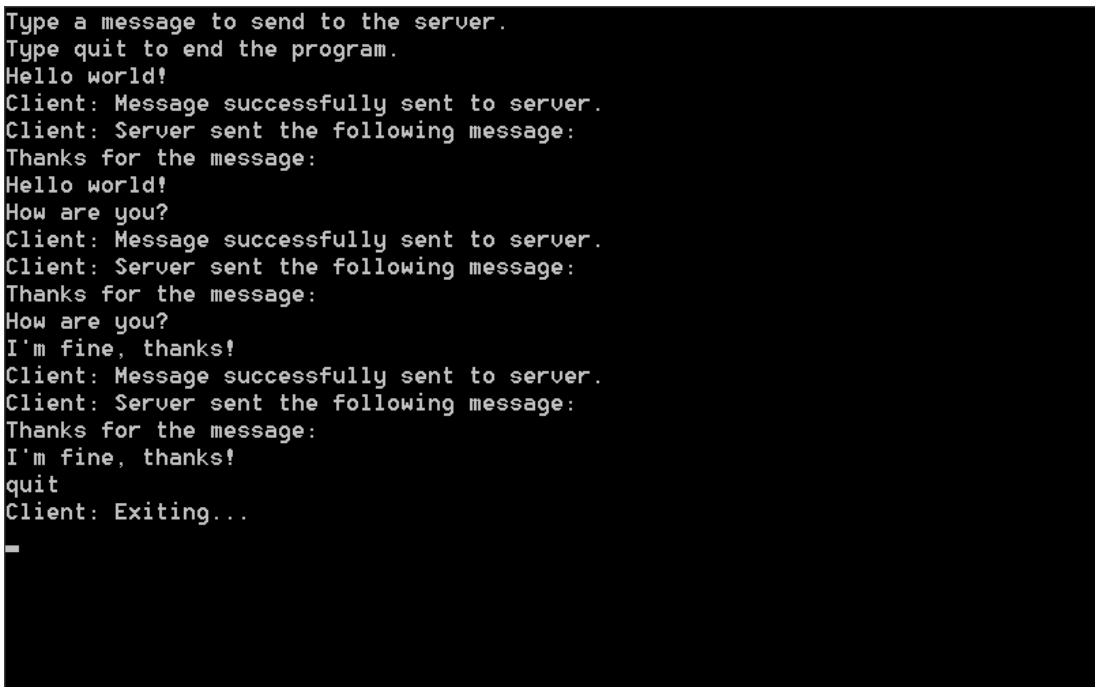

Рисунок 2.2 – Результат роботи клієнта конвеєра з Прикладу 2.2

```
#include "stdafx.h"

#include <windows.h>
#include <strsafe.h>
#include <iostream>

using namespace std;

#define BUFSIZE 512

//ім'я конвеєра
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\examplepipe");

int _tmain(int argc, TCHAR *argv[])
{
    HANDLE hPipe;
    TCHAR lpClientMessage[BUFSIZE];
    BOOL fSuccess = FALSE;
    DWORD cbReadBytes, cbToWriteBytes, cbWrittenBytes, dwMode;

    //отримуємо дескриптор купи
    HANDLE hHeap = GetProcessHeap();

    //виділяємо пам'ять для повідомлень
    TCHAR* pchMessage = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE * sizeof(TCHAR));
    TCHAR* pchReply = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE * sizeof(TCHAR));

    if(pchMessage == NULL)
    {
        //якщо сталася помилка
        cout << "ERROR - Client:" << endl
```

```
        << "Couldn't allocate heap memory for the client message!" << endl
        << "Client thread exiting.";

    //очищуємо пам'ять
    if(pchReply != NULL)
    {
        HeapFree(hHeap, 0, pchReply);
    }

    return 1;
}

if(pchReply == NULL)
{
    //якщо сталася помилка
    cout << "ERROR - Client:" << endl
        << "Couldn't allocate heap memory for the server reply!" << endl
        << "Client thread exiting.";

    //очищуємо пам'ять
    if(pchMessage != NULL)
    {
        HeapFree(hHeap, 0, pchMessage);
    }

    return 1;
}

while(1)
{
    //пробуємо відкрити конвеєр
    hPipe = CreateFile(
        lpszPipename,  //ім'я конвеєра
        GENERIC_READ |
        GENERIC_WRITE, //права доступу читання та запису
        0,             //без розділювання доступу
        NULL,          //атрибути безпеки за замовчуванням
        OPEN_EXISTING, //відкриваємо існуючі конвеєри
        0,             //атрибути за замовчуванням
        NULL);         //без шаблону

    if(hPipe != INVALID_HANDLE_VALUE)
    {
        //якщо успішно з'єднано, виходимо з циклу
        break;
    }

    if(GetLastError() == ERROR_PIPE_BUSY)
    {
        //якщо всі екземпляри зайнято, заждемо
        if(!WaitNamedPipe(lpszPipename, 10000))
        {
            cout << "Client: Could not open pipe in 20 seconds." << endl
                << "Quitting the client.";

            //закриваємо дескриптор екземпляра конвеєра
            CloseHandle(hPipe);
```

```
            //очищуємо пам'ять
            HeapFree(hHeap, 0, pchMessage);
            HeapFree(hHeap, 0, pchReply);

            return -1;
        }
    }
    else
    {
        //якщо сталася помилка
        cout << "Client: Could not open a pipe, error code is "
            << GetLastError() << "." << endl
            << "Quitting the client.";

        //закриваємо дескриптор екземпляра конвеєра
        CloseHandle(hPipe);

        //очищуємо пам'ять
        HeapFree(hHeap, 0, pchMessage);
        HeapFree(hHeap, 0, pchReply);

        return -1;
    }
}

//оскільки ми успішно під'єдналися до екземпляра конвеєра,
//потрібно змінити його режим читання,
//щоб режим відповідав режиму сервера
dwMode = PIPE_READMODE_MESSAGE;
fSuccess = SetNamedPipeHandleState(
    hPipe,   //дескриптор екземпляра конвеєра
    &dwMode, //новий режим конвеєра
    NULL,    //не задаємо максимальну кількість байтів
    NULL);   //не задаємо час

if(!fSuccess)
{
    //якщо сталася помилка
    cout << "Client: Could not change the pipe read mode, error code is "
        << GetLastError() << "." << endl
        << "Quitting the client.";

    //закриваємо дескриптор екземпляра конвеєра
    CloseHandle(hPipe);

    //очищуємо пам'ять
    HeapFree(hHeap, 0, pchMessage);
    HeapFree(hHeap, 0, pchReply);

    return -1;
}

cout << "Type a message to send to the server." << endl
    << "Type quit to end the program." << endl;

while(1)
```

```
{
    //зчитуємо з консолі повідомлення для відправлення на сервер
    _getws_s(lpClientMessage);

    if(!wcscmp(lpClientMessage, L"quit"))
    {
        //якщо це "quit"

        //виходимо з циклу
        break;
    }
    else{
        //копіюємо повідомлення в буфер для записування
        StringCchCopy(pchMessage, BUFSIZE, lpClientMessage);
        cbToWriteBytes = (lstrlen(lpClientMessage) + 1) * sizeof(TCHAR);

        //записуємо повідомлення в конвеєр
        fSuccess = WriteFile(
            hPipe,           //дескриптор екземпляра конвеєра
            pchMessage,      //буфер із даними
            cbToWriteBytes,  //кількість байтів для записування
            &cbWrittenBytes, //кількість реально записаних байтів
            NULL);           //синхронний режим

        if(!fSuccess)
        {
            //якщо сталася помилка
            cout << "Client: Could not write to the pipe, error code is "
                << GetLastError() << "." << endl
                << "Quitting the client.";

            //закриваємо дескриптор екземпляра конвеєра
            CloseHandle(hPipe);

            //очищуємо пам'ять
            HeapFree(hHeap, 0, pchMessage);
            HeapFree(hHeap, 0, pchReply);

            return -1;
        }

        cout << "Client: Message successfully sent to server." << endl;

        do
        {
            //зчитуємо повідомлення від сервера з конвеєра
            fSuccess = ReadFile(
                hPipe,                  //дескриптор екземпляра конвеєра
                pchReply,               //буфер, куди буде зчитано дані
                BUFSIZE * sizeof(TCHAR), //розмір буфера
                &cbReadBytes,           //кількість прочитаних байтів
                NULL);                  //синхронний режим

            if(!fSuccess && GetLastError() != ERROR_MORE_DATA)
            {
                //якщо більше немає даних
                break;
```

```
            }

            cout << "Client: Server sent the following message:" << endl;
            wcout << pchReply << endl;
        }
        while(!fSuccess); //повторюємо цикл, якщо ERROR_MORE_DATA

        if(!fSuccess)
        {
            //якщо сталася помилка
            cout << "Client: Could not read from the pipe, error code is "
                << GetLastError() << "." << endl
                << "Quitting the client.";

            //закриваємо дескриптор екземпляра конвеєра
            CloseHandle(hPipe);

            //очищуємо пам'ять
            HeapFree(hHeap, 0, pchMessage);
            HeapFree(hHeap, 0, pchReply);

            return -1;
        }
    }
}

//закриваємо дескриптор екземпляра конвеєра
CloseHandle(hPipe);

//очищуємо пам'ять
HeapFree(hHeap, 0, pchMessage);
HeapFree(hHeap, 0, pchReply);

cout << "Client: Exiting..." << endl;
getchar();

return 0;
}
```

2.4.5 Приклад реалізації поштового ящика

Приклад 2.3.

Розгляньмо вихідний текст консольної програми Windows API на мові програмування C++, що реалізує сервер поштового ящика.

У даній програмі сервер створює за допомогою функції `CreateMailslot` (див. розділ 2.3.2) поштовий ящик, після чого в нескінченному циклі

читає повідомлення, які йому надсилають клієнти.

Для того, щоб визначити, чи є в ящику повідомлення, сервер використовує функцію `GetMailslotInfo` (див. розділ 2.3.3). Якщо повідомлення в ящику наявні, сервер за допомогою функції `ReadFile` читає їх (див. розділ 2.4.2) та виводить на екран.

Результат роботи сервера під час спілкування з клієнтом із Прикладу 2.4 представлено на рисунку 2.3.

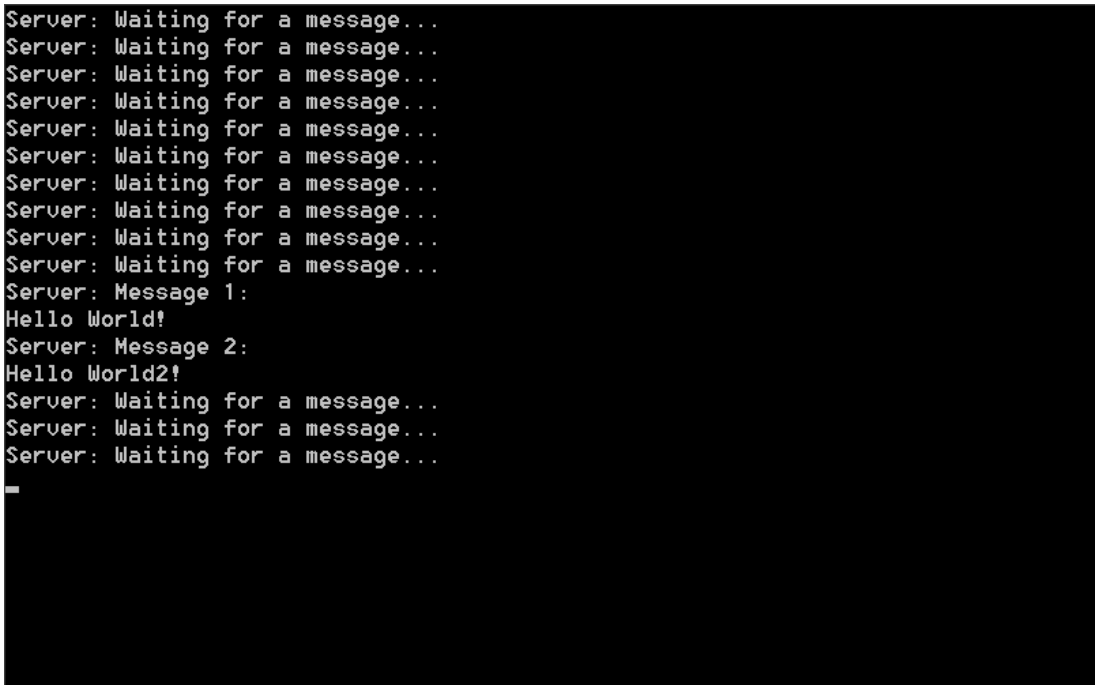

Рисунок 2.3 – Результат роботи сервера конвеєра з Прикладу 2.3

```
#include "stdafx.h"

#include <windows.h>
#include <strsafe.h>
#include <iostream>

using namespace std;

#define BUFSIZE 424

HANDLE hSlot;
TCHAR* pchMessage;

//ім'я поштового ящика
LPTSTR lpSlotName = TEXT("\\\\.\\mailslot\\exampleslot");
```

```
BOOL readSlot();
BOOL WINAPI makeSlot(LPTSTR);

int _tmain(int argc, TCHAR *argv[])
{
    //отримуємо дескриптор купи
    HANDLE hHeap = GetProcessHeap();

    //виділяємо пам'ять для повідомлень
    pchMessage = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE * sizeof(TCHAR));

    if(pchMessage == NULL)
    {
        //якщо сталася помилка
        cout << "ERROR - Server:" << endl
            << "Couldn't allocate heap memory for the message!" << endl
            << "Client thread exiting." << endl;

        return 1;
    }

    //створюємо поштовий ящик
    makeSlot(lpSlotName);

    while(TRUE)
    {
        if(!readSlot())
        {
            //якщо сталася помилка
            break;
        }
        else
        {
            //інакше, ждемо протягом 1 секунди
            Sleep(1000);
        }
    }

    //закриваємо дескриптор поштового ящика
    CloseHandle(hSlot);

    //очищуємо пам'ять
    HeapFree(hHeap, 0, pchMessage);
}

//функція для створення поштового ящика
BOOL WINAPI makeSlot(LPTSTR lpszSlotName)
{
    //create the mailslot
    hSlot = CreateMailslot(
        lpszSlotName,          //ім'я поштового ящика
        0,                     //без обмежень на розмір повідомлень
        MAILSLOT_WAIT_FOREVER, //не будемо ждати, поки завершаться операції
        NULL);                 //атрибути безпеки за замовчуванням

    if(hSlot == INVALID_HANDLE_VALUE)
    {
```

```
        //якщо сталася помилка
        cout << "ERROR - Server: couldn't create the mailslot, error code is "
            << GetLastError() << "." << endl;

        return FALSE;
    }

    return TRUE;
}

//функція для прочитання повідомлень із поштового ящика
BOOL readSlot()
{
    DWORD cbMessage, cMessage, cbRead;
    BOOL fResult;
    DWORD cAllMessages;

    cbMessage = cMessage = cbRead = 0;

    //читаємо інформацію про поштовий ящик
    fResult = GetMailslotInfo(hSlot, //дескриптор поштового ящика
        (LPDWORD) NULL,              //без обмежень на розмір повідомлень
        &cbMessage,                  //розмір наступного повідомлення
        &cMessage,                   //кількість повідомлень
        (LPDWORD) NULL);             //не будемо ждати, поки завершиться читання

    if(!fResult)
    {
        //якщо сталася помилка
        cout << "ERROR - Server: couldn't get the mailslot info, error code is "
            << GetLastError() << "." << endl;

        return FALSE;
    }

    if(cbMessage == MAILSLOT_NO_MESSAGE)
    {
        //якщо поштовий ящик порожній
        cout << "Server: Waiting for a message..." << endl;

        return TRUE;
    }

    cAllMessages = cMessage;

    while(cMessage != 0)
    {
        //у циклі по всіх повідомленнях

        //зчитуємо повідомлення з поштового ящика
        fResult = ReadFile(
            hSlot,      //дескриптор поштового ящика
            pchMessage, //буфер, куди буде зчитано дані
            cbMessage,  //розмір буфера
            &cbRead,    //кількість прочитаних байтів
            NULL);      //синхронний режим
```

```
        if(!fResult)
        {
            //якщо сталася помилка
            cout << "Server: Failed reading the message, error code is "
                << GetLastError() << "." << endl;

            return FALSE;
        }

        //виводимо повідомлення в консоль
        cout << "Server: Message " << cAllMessages - cMessage + 1 << ":" << endl;
        wcout << pchMessage << endl;

        //читаємо інформацію про поштовий ящик
        fResult = GetMailslotInfo(hSlot, //дескриптор поштового ящика
            (LPDWORD) NULL,              //без обмежень на розмір повідомлень
            &cbMessage,                  //розмір наступного повідомлення
            &cMessage,                   //кількість повідомлень
            (LPDWORD) NULL);             //не будемо ждати, поки завершиться читання



        if(!fResult)
        {
            //якщо сталася помилка
            cout << "ERROR - Server: couldn't get the mailslot info, error code is "
                << GetLastError() << "." << endl;

            return FALSE;
        }
    }

    return TRUE;
}
```

Приклад 2.4.

Розгляньмо вихідний текст консольної програми Windows API на мові програмування C++, що реалізує клієнт поштового ящика.

Спочатку клієнт за допомогою функції `CreateFile` (див. розділ 2.4.1) отримує дескриптор поштового ящика, створеного в Прикладі 2.3. Після цього клієнт виконує такі дії:

- читає з консолі уведення від користувача — кількість повідомлень, яку він хоче надіслати;
- читає з консолі повідомлення від користувача;
- надсилає за допомогою функції `WriteFile` (див. розділ 2.4.3) серверу це уведення;
- завершує свою роботу.

Варто зазначити, що в цій програмі не виконано перевірку на переповнення буфера під час уведення повідомлення з клавіатури. У роботах студентів відповідні перевірки повинно бути обов'язково передбачено.

Результат роботи клієнта під час спілкування з сервером із Прикладу 2.3 представлено на рисунку 2.4.

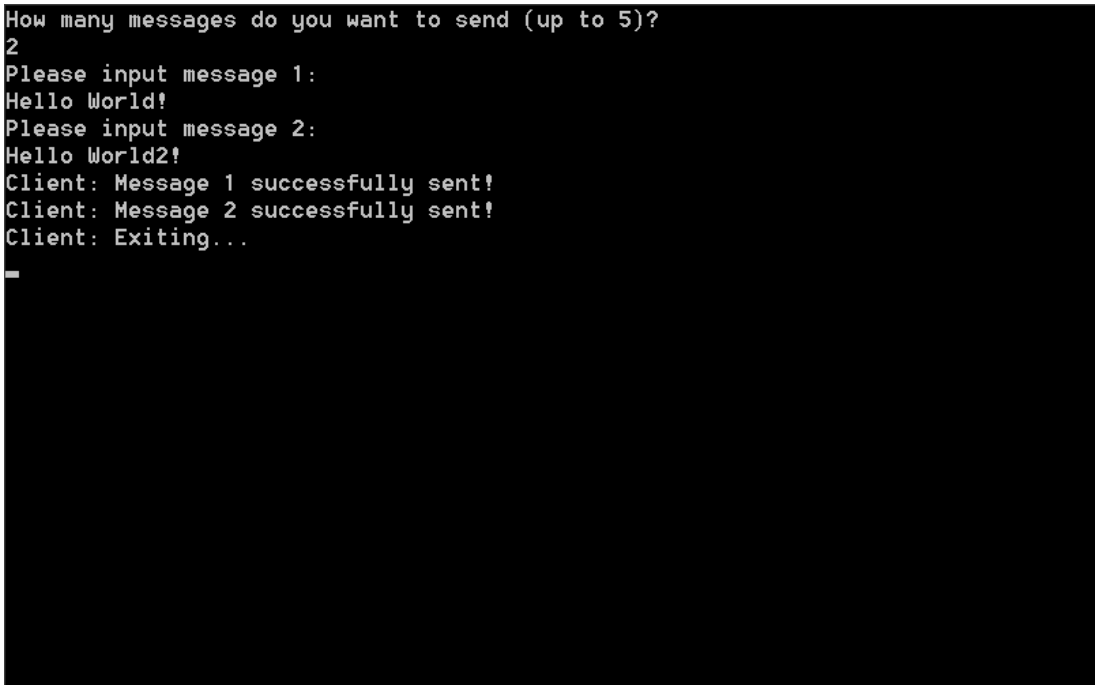

Рисунок 2.4 – Результат роботи клієнта поштового ящика з Прикладу 2.4

```
#include "stdafx.h"

#include <windows.h>
#include <strsafe.h>
#include <iostream>
#include <list>

#define BUFSIZE 424

using namespace std;

LPTSTR lpMessage;
list<LPTSTR> lpMessages;

//ім'я поштового ящика
LPTSTR lpSlotName = TEXT("\\\\.\\mailslot\\exampleslot");

BOOL writeSlot(HANDLE, LPTSTR, int);

int _tmain(int argc, TCHAR *argv[])
```

```
{
    HANDLE hFile;
    int messageCount;
    TCHAR messageCountStr[2];

    //отримуємо дескриптор купи
    HANDLE hHeap = GetProcessHeap();

    //виділяємо пам'ять для повідомлення
    TCHAR* pchMessage = (TCHAR*) HeapAlloc(hHeap, 0, BUFSIZE*sizeof(TCHAR));

    if(pchMessage == NULL)
    {
        //якщо сталася помилка
        cout << "ERROR - Server:" << endl
            << "Couldn't allocate heap memory for the message!" << endl;

        return 1;
    }

    //відкриваємо дескриптор поштового ящика
    hFile = CreateFile(lpSlotName, //ім'я поштового ящика
        GENERIC_WRITE,             //право доступу писання
        FILE_SHARE_READ,           //розділяємо доступ тільки для читання
        NULL,                      //атрибути безпеки за замовчуванням
        OPEN_EXISTING,             //відкриваємо існуючий поштовий ящик
        0,                         //атрибути за замовчуванням
        (HANDLE) NULL);            //без шаблону

    if(hFile == INVALID_HANDLE_VALUE)
    {
        //якщо сталася помилка
        cout << "ERROR - Client: couldn't get the handle to the mailslot." << endl
            << "Error code is " << GetLastError() << "." << endl;

        //очищуємо пам'ять
        HeapFree(hHeap, 0, pchMessage);

        return FALSE;
    }

    //запитуємо в користувача кількість повідомлень
    cout << "How many messages do you want to send (up to 5)?" << endl;
    _getws_s(messageCountStr);

    messageCount = _wtoi(messageCountStr);

    int i = 0;
    for(; i < messageCount; i++)
    {
        //у циклі запитуємо в користувача повідомлення
        cout << "Please input message " << (i + 1) << ":" << endl;

        lpMessage = new TCHAR[BUFSIZE];
        _getws_s(lpMessage);

        lpMessages.push_back(lpMessage);
```

```
    }

    list<TCHAR*>::iterator iter;
    i = 0;
    for(iter = lpMessages.begin(); iter != lpMessages.end(); iter++)
    {
        //копіюємо повідомлення в буфер для записування
        StringCchCopy(pchMessage, BUFSIZE, *iter);

        //записуємо дані з буфера в поштовий ящик
        writeSlot(hFile, pchMessage, ++i);
    }

    //очищуємо список
    lpMessages.clear();

    //закриваємо дескриптор поштового ящика
    CloseHandle(hFile);

    //очищуємо пам'ять
    HeapFree(hHeap, 0, pchMessage);

    cout << "Client: Exiting..." << endl;
    getchar();

    return TRUE;
}

//функція для записування в поштовий ящик
BOOL writeSlot(HANDLE hSlot, LPTSTR lpszMessage, int index)
{
    BOOL fResult;
    DWORD cbWritten;

    //записуємо повідомлення в поштовий ящик
    fResult = WriteFile(hSlot,             //дескриптор поштового ящика
        lpszMessage,                       //буфер із даними
        (DWORD) (lstrlen(lpszMessage) + 1)
            * sizeof(TCHAR),               //кількість байтів для записування
        &cbWritten,                        //кількість реально записаних байтів
        NULL);                             //синхронний режим

    if(!fResult)
    {
        //якщо сталася помилка
        cout << "ERROR - Client: couldn't write to the mailslot." << endl
            << "Error code is " << GetLastError() << "." << endl;

        return FALSE;
    }

    cout << "Client: Message " << index << " successfully sent!" << endl;

    return TRUE;
}
```

3 ПОРЯДОК ЗДАЧІ ЛАБОРАТОРНОЇ РОБОТИ ТА ВИМОГИ ДО ЗВІТУ

Здача лабораторної роботи передбачає:

- демонстрацію працездатності взаємодії розроблених сервера та клієнта іменованого конвеєра (або поштового ящика, на вибір студена) в ОС Windows версії, не нижчої за 2000;
- здачу викладачеві звіту з лабораторної роботи.

Під час здавання лабораторної роботи викладач має право ставити студентові питання за матеріалами розділу 2.

Під час демонстрації роботи програми викладач має право вимагати від студента внесення змін як у сервер, так і в клієнт.

Звіт про виконання лабораторної роботи повинен включати:

а) вступ;

б) постановку задачі;

в) теоретичні відомості;

г) опис розробленої програми (окремо — для сервера та клієнта файлової абстракції);

д) результати випробування розробленої програми;

е) висновки;

є) посилання на літературні джерела;

ж) додаток, у якому повинно бути розміщено лістинг розробленої програми.

Детальніші вимоги до звіту викладено в документі «Вимоги до оформлення звіту».

Текстову частину повинно бути оформлено відповідно до вимог ДСТУ 3008-95 «Документація. Звіти у сфері науки і техніки. Структура і правила оформлення» із застосуванням системи підготовки документів LaTeX.

ПЕРЕЛІК ПОСИЛАНЬ

1. Microsoft Library [Електронний ресурс]. — Режим доступу: https://msdn.microsoft.com/en-us/library/

2. Petzold C. Programming Windows / C. Petzold. — [5th ed.] — Microsoft Press, 1998. — 1100 p.

3. Simon R. Windows NT Win32 API SuperBible / R. Simon. — Waite Group Press, 1997. — 1510 p.