

# ЛАБОРАТОРНА РОБОТА №1

## ПРОЦЕСИ ТА ПОТОКИ В ОС WINDOWS

### 1.1 Вступ

Аплікації, написані для операційної системи (ОС) Microsoft Windows, складаються з одного або декількох процесів. Згідно з найпростішим визначенням, під *процесом* (process) розуміють виконувану програму. Кожний процес забезпечує аплікацію потрібними для виконання ресурсами.

Процес має віртуальний адресовий простір, виконуваний код, відкриті дескриптори системних об'єктів (open handles to system objects), безпековий контекст (security context), унікальний ідентифікатор, змінні середовища, клас пріоритету, мінімальний та максимальний розмір робочої множини (working set), а також щонайменше один потік.

*Потік* (thread) — це базова одиниця в рамках процесу, якій ОС виділяє процесорний час. Потік може виконувати будь-яку частину коду процесу, у тому числі частини коду, які в даний момент виконує інший потік. Усі потоки процесу мають спільний віртуальний адресовий простір та системні ресурси. Окрім того, кожний процес підтримує опрацьовувачі виключних ситуацій (exception handlers), планувальний пріоритет (scheduling priority), локальну пам'ять (local storage), унікальний ідентифікатор, а також низку структур, які ОС використовує для збереження контексту потоку під час його простоювання. *Контекст потоку* (thread context) включає значення регістрів процесора, системний стек (kernel stack), блок середовища потоку, а також користувацький стек (user stack) в адресовому просторі процесу, якому належить потік. Потоки також можуть мати власний безпековий контекст.

На момент створення кожний процес має щонайменше один потік (має назву *основного потоку* (primary thread)). У процесі своєї роботи процес може створювати додаткові потоки.

ОС Windows підтримує режим *витискуючої багатозадачності* (preemptive multitasking), яка створює ефект одночасного виконання багатьох потоків. На однопроцесорній машині це, звісно, не більше ніж ілюзія, яка створюється за рахунок швидкого перемикання між потоками. На мультипроцесорній машині система може справді одночасно виконувати стільки потоків, скільки в машині є процесорів.

У даній роботі розглядатимемо засоби роботи з багатопотоковими аплікаціями в ОС Windows версії XP та вище.

### 1.2 Постановка задачі

У даній лабораторній роботі потрібно ознайомитися з концепцією багатозадачності в ОС Windows, навчитися створювати за допомогою засобів Windows API багатопотокові аплікації та розбиратися з проблемами, що виникають під час спільного множинного доступу до ресурсів.

У рамках виконання лабораторної роботи потрібно:

- а) ознайомитися з теоретичними відомостями, викладеними в розділі 1.3;
  - б) написати будь-якою мовою програмування програму, яка повинна:
    - 1) за потреби створювати додаткові потоки, окрім головного;
    - 2) регулювати доступ потоків до деякого ресурсу (файлу, графічного об'єкта, статичної змінної і т.п.); зокрема, у програмі повинно бути передбачено можливість як синхронного, так і асинхронного доступу потоків до ресурсу (на вимогу користувача);
    - 3) регулювати пріоритети потоків на вимогу користувача;
    - 4) за потреби зупиняти та відновлювати потоки;
    - 5) використовувати тільки засоби Windows API і не повинна використовувати стандартні процедури MFC, спеціальні класи для реалізації багатопотокових аплікацій тощо;
  - в) відлагодити програму в ОС Windows версії XP та вище;
  - г) підготувати звіт із лабораторної роботи відповідно до вимог розділу 1.4.
- Кожен студент повинен написати програму з індивідуальними функціональними можливостями. Програми різних студентів не можуть мати однакові функціональні можливості.

Типова програма, яку можна написати в рамках даної лабораторної роботи, може передбачати наявність одного інтерфейсного потоку, який здійснює спілкування з користувачем та керування іншими потоками, а також декількох робочих потоків, які виконують фонові задачі (перерахунок даних, друк тексту, уведення/виведення, пошук файлів тощо).

### 1.3 Теоретичні відомості

### 1.3.1 Режим багатозадачності

Багатозадачеві операційні системи розділяють наявний процесорний час між процесами чи потоками, які його потребують. ОС Windows розроблено для витискуючої багатозадачності: вона виділяє *квант часу* (time slice) кожному потоку, який вона виконує. Щойно квант часу поточного потоку вичерпано, ОС зупиняє його виконання, дозволяючи таким чином виконуватися іншим потокам. Під час перемикання з одного потоку на інший, ОС зберігає контекст витисненого потоку та відновлює збережений раніше контекст наступного потоку в черзі.

Величина кванту часу залежить від ОС та від процесора. Проте, оскільки зазвичай квант часу доволі малий (приблизно 20 мс), створюється враження, що одночасно виконуються багато потоків. Проте, багатопотоковість варто використовувати обережно, оскільки за великої кількості потоків продуктивність ОС може знизитися.

### 1.3.2 Переваги багатозадачності

Використання багатозадачності має такі переваги:

- для користувача перевага полягає в можливості одночасно працювати з декількома аплікаціями. Наприклад, користувач може редагувати файли в одній аплікації у той час, поки музичний плеєр програв музичні файли;
- для розробника перевага полягає в можливості створення аплікацій, що використовують більше за один процес, а також створення процесів, що використовують більше за один потік. Наприклад, процес може мати потік графічного інтерфейсу користувача для опрацювання спілкування з користувачем, а також низку робочих потоків для виконання певних задач під час очікування введення з боку користувача. Якщо надати потоку інтерфейсу більший пріоритет, то аплікація буде швидше реагувати на запити користувача, у той час як робочі потоки ефективно використовуватимуть час процесора під час очікування введення з боку користувача.

Окрім цього, багатопотоковість можна використовувати для пришвидшення роботи аплікацій на мультипроцесорних системах. Фактично, у таких системах це — єдиний варіант розпаралелювання виконання задач.

Якщо аплікація має тільки один процес з одним потоком, то вона буде виконуватися однаково швидко як на однопроцесорній, так і на багатопроцесорній машині. Тому для ефективного використання можливостей мультипроцесорних систем доцільно володіти основами створення багатопотокових аплікацій.

### 1.3.3 Принципи застосування багатозадачності

Існує два основні підходи до реалізації багатозадачності:

- один процес із декількома потоками;
- декілька процесів, кожен із яких має один чи декілька потоків. У цьому разі аплікація може помістити потоки, які вимагають окремих адресових просторів та ресурсів, в окремі процеси (щоб захистити їх від впливу потоків з інших процесів).

У загальному випадку, використання багатопотокового процесу дозволяє виконувати такі задачі:

- керування введенням для декількох вікон;
- керування введенням із різних пристроїв;
- надавати різним задачам різний пріоритет, наприклад, високопріоритетний потік виконує задачі, чутливі до часу, а низькопріоритетний потік виконує всі інші задачі;
- підтримувати графічний інтерфейс користувача в активному стані під час виконання фонових задач;
- розпаралелювати опрацювання даних між декількома потоками на мультипроцесорних системах.

За допомогою потоків, багатопотоковий процес може як керувати взаємновиключними задачами (наприклад, забезпечувати взаємодію з користувачем та виконувати фонові обчислення), так і структурувати аплікацію, яка виконує декілька однотипних задач одночасно (наприклад, сервер іменованого конвеєра (named pipe server) може створити окремі, але однотипні потоки для кожного клієнта).

Як правило, підхід зі створенням одного багатопотокового процесу продуктивніший за підхід зі створенням декількох процесів із таких міркувань:

- ОС може швидше перемикати контекст між потоками радше ніж між процесами, тому що контекст процесу більший і складніший за контекст потоку;

- усі потоки в процесі розділяють спільний адресовий простір та можуть мати доступ до глобальних змінних процесу, що спрощує спілкування між потоками;
- усі потоки в процесі можуть розділяти відкриті дескриптори ресурсів, таких як файли або конвеєри.

#### 1.3.4 Застереження під час застосування багатозадачовості

Під час застосування багатозадачовості потрібно звертати увагу на низку факторів. Так, зокрема, багатозадачовість спряжена з використанням системних ресурсів:

- ОС використовує оперативну пам'ять для зберігання контекстів процесів та потоків, тому кількість процесів та потоків, які може бути створено, обмежено наявним обсягом пам'яті;
- відслідковування великої кількості потоків суттєво поглинає процесорний час. Якщо потоків стане занадто багато, більшість із них не зможуть ефективно виконуватися. Якщо більшість поточних потоків належать одному процесу, то потоки в інших процесах отримуватимуть доступ до процесора значно рідше.

Як правило, доцільно використовувати якомога менше потоків з метою мінімізації використання системних ресурсів, що в свою чергу збільшує продуктивність аплікації.

Окрім ресурсних застережень, надання декільком потокам спільного доступу до ресурсів може спричиняти *конфлікти*. Для їх упередження потрібно *синхронізувати* доступ до розділюваних ресурсів. Це стосується як системних ресурсів (наприклад, порти), так і ресурсів, розділюваних між декількома процесами (наприклад, дескриптори файлів), так і ресурсів одного процесу, розділюваних між декількома потоками (наприклад, глобальні змінні).

Якщо доступ до ресурсів не буде належним чином синхронізовано, можуть мати місце взаємоблокування (deadlocks) та гонки (race conditions). Windows API надає розробнику низку об'єктів синхронізації та функцій, які можна використовувати для координації спільного використання ресурсів з боку декількох потоків. Ми розглянемо питання синхронізації доступу потоків до спільних ресурсів детальніше в розділі 1.3.15.

#### 1.3.5 Планування потоків

Управління багатозадачовістю здійснює *планувальник* ОС (system scheduler). Його завдання полягає у визначенні потоку, який повинен отримати черговий квант процесорного часу.

При цьому планувальник ураховує *планувальні пріоритети* (scheduling priorities) потоків. Кожному потоку призначають планувальний пріоритет, який може набувати значень від 0 (найнижчий пріоритет) до 31 (найвищий пріоритет).

ОС розподіляє кванти часу згідно з *циклічним алгоритмом* (round robin) [1], застосовуючи його послідовно до кожної групи потоків з однаковим пріоритетом:

- спочатку ОС намагається виділити квант часу згідно з циклічним алгоритмом потокам із найвищим пріоритетом;
- якщо жоден із потоків із найвищим пріоритетом не готовий до виконання, ОС намагається виділити квант часу згідно з циклічним алгоритмом потокам із наступним за величиною пріоритетом, і так далі;
- якщо потік із вищим пріоритетом стає готовим до виконання у той час, як виконується потік із нижчим пріоритетом, ОС припиняє виконання останнього (надаючи йому завершити свій квант часу) та призначає повний квант часу потоку з вищим пріоритетом.

Пріоритет кожного потоку залежить від *класу пріоритету* (priority class) процесу, якому він належить, та *рівня пріоритету* (priority level) потоку в рамках відповідного класу пріоритету. Клас та рівень пріоритету об'єднують для визначення *базового пріоритету* (base priority) потоку.

#### 1.3.6 Клас пріоритету

Кожен процес належить до одного з класів пріоритетів (наведено в порядку зростання):

- `IDLE_PRIORITY_CLASS` — процеси, які моніторять систему, такі як заставки екрану або аплікації, що періодично оновлюють дисплей; це – найнижчий клас пріоритету;
- `BELOW_NORMAL_PRIORITY_CLASS`;
- `NORMAL_PRIORITY_CLASS` — процеси після створення належать цьому класу за замовчуванням. При цьому варто зазначити, що якщо батьківський процес належить класу `IDLE_PRIORITY_CLASS` чи `BELOW_NORMAL_PRIORITY_CLASS`, то дочірні процеси за замовчуванням належатимуть класу пріоритету батьківського процесу;
- `ABOVE_NORMAL_PRIORITY_CLASS`;

- **HIGH\_PRIORITY\_CLASS** — використання процесів такого високого класу пріоритету потребує особливої уваги, адже якщо потік належатиме процесу такого класу пріоритету протягом тривалого періоду, інші потоки не зможуть отримати процесорний час. Доцільно задавати такий клас пріоритету тільки для процесів, які повинні реагувати на події, чутливі до часу, і тільки на період такого реагування;

- **REALTIME\_PRIORITY\_CLASS** — це найвищий клас пріоритету, який не варто застосовувати майже ніколи, оскільки потоки, що належать процесам цього класу, матимуть пріоритет, вищий за пріоритет уведення від миші, клавіатури та підкачування диску.

Клас пріоритету можна задати як під час створення процесу за допомогою функції `CreateProcess` (див. розділ 1.3.13), так і під час виконання процесу за допомогою функції `SetPriorityClass`. Остання функція має наступний формат:

```
BOOL WINAPI SetPriorityClass(  
    HANDLE hProcess,  
    DWORD dwPriorityClass  
);
```

Аргумент `hProcess` цієї функції відповідає за *дескриптор процесу* (process handle), клас пріоритету якого потрібно змінити, аргумент `dwPriorityClass` визначає новий клас пріоритету. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

Дізнатися клас пріоритету процесу під час його роботи можна за допомогою функції `GetPriorityClass`. Ця функція має наступний формат:

```
DWORD WINAPI GetPriorityClass(  
    HANDLE hProcess  
);
```

Аргумент `hProcess` цієї функції відповідає за дескриптор процесу, клас пріоритету якого потрібно дізнатися. У випадку успішного завершення, функція повертає клас пріоритету. У випадку помилки функція повертає 0.

### 1.3.7 Рівень пріоритету

У рамках кожного класу пріоритету виділяють такі рівні пріоритету:

- **THREAD\_PRIORITY\_IDLE**;
- **THREAD\_PRIORITY\_LOWEST**;
- **THREAD\_PRIORITY\_BELOW\_NORMAL**;
- **THREAD\_PRIORITY\_NORMAL**;
- **THREAD\_PRIORITY\_ABOVE\_NORMAL**;
- **THREAD\_PRIORITY\_HIGHEST**;
- **THREAD\_PRIORITY\_TIME\_CRITICAL**.

На момент створення всі потоки мають рівень **THREAD\_PRIORITY\_NORMAL**.

Рівень пріоритету потоку можна змінити в процесі його виконання за допомогою функції `SetThreadPriority`. Ця функція має наступний формат:

```
BOOL WINAPI SetThreadPriority(  
    HANDLE hThread,  
    int nPriority  
);
```

Аргумент `hThread` цієї функції відповідає за *дескриптор потоку* (thread handle), рівень пріоритету якого потрібно змінити, аргумент `nPriority` визначає новий рівень пріоритету. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

Дізнатися рівень пріоритету потоку під час його виконання можна за допомогою функції `GetThreadPriority`. Ця функція має наступний формат:

```
int WINAPI GetThreadPriority(  
    HANDLE hThread  
);
```

Аргумент `hThread` цієї функції відповідає за дескриптор потоку, рівень пріоритету якого потрібно дізнатися. У випадку успішного завершення, функція повертає рівень пріоритету. У випадку помилки функція повертає **THREAD\_PRIORITY\_ERROR\_RETURN**.

Типова стратегія полягає у використанні рівнів `THREAD_PRIORITY_ABOVE_NORMAL` чи `THREAD_PRIORITY_HIGHEST` для потоків, пов'язаних з уведенням від користувача. Такий підхід забезпечить ефективне реагування аплікації на запити користувача. Рівні фонових потоків, особливо тих, що активно використовують процесорний час, можна встановити в `THREAD_PRIORITY_BELOW_NORMAL` чи `THREAD_PRIORITY_LOWEST`. Такий підхід дозволить за потреби витиснути їх пріоритетнішими потоками.

У випадку, коли потік із вищим пріоритетом жде завершення виконання потоку з нижчим пріоритетом, потрібно явно призупиняти виконання потоку з вищим пріоритетом. Це можна зробити за допомогою функцій очікування (wait functions), критичної секції (critical section) (див. розділ 1.3.15), або функцій `Sleep`, `SleepEx` чи `SwitchToThread` (див. розділ 1.3.14). Застосування вказаних методів ефективніше за використання циклу, у якому потік із вищим пріоритетом постійно перевіряє стан потоку з нижчим пріоритетом, оскільки в останньому разі потік із вищим пріоритетом постійно залишається активним.

### 1.3.8 Базовий пріоритет

Базовий пріоритет потоку можна визначити на основі інформації про клас пріоритету процесу, якому він належить, та його рівень пріоритету згідно з таблицею 1.1.

Таблиця 1.1 – Таблиця визначення базового пріоритету потоку

Клас пріоритету	Рівень пріоритету	Базовий пріоритет
BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	4
	THREAD_PRIORITY_BELOW_NORMAL	5
	THREAD_PRIORITY_NORMAL	6
	THREAD_PRIORITY_ABOVE_NORMAL	7
	THREAD_PRIORITY_HIGHEST	8
	THREAD_PRIORITY_TIME_CRITICAL	15
NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	6
	THREAD_PRIORITY_BELOW_NORMAL	7
	THREAD_PRIORITY_NORMAL	8
	THREAD_PRIORITY_ABOVE_NORMAL	9
	THREAD_PRIORITY_HIGHEST	10

Клас пріоритету	Рівень пріоритету	Базовий пріоритет
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	2
	THREAD_PRIORITY_BELOW_NORMAL	3
	THREAD_PRIORITY_NORMAL	4
	THREAD_PRIORITY_ABOVE_NORMAL	5
	THREAD_PRIORITY_HIGHEST	6
	THREAD_PRIORITY_TIME_CRITICAL	15
Клас пріоритету	Рівень пріоритету	Базовий пріоритет
	THREAD_PRIORITY_TIME_CRITICAL	15
ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	8
	THREAD_PRIORITY_BELOW_NORMAL	9
	THREAD_PRIORITY_NORMAL	10
	THREAD_PRIORITY_ABOVE_NORMAL	11
	THREAD_PRIORITY_HIGHEST	12
	THREAD_PRIORITY_TIME_CRITICAL	15
HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	11
	THREAD_PRIORITY_BELOW_NORMAL	12
	THREAD_PRIORITY_NORMAL	13

Клас пріоритету	Рівень пріоритету	Базовий пріоритет
	THREAD_PRIORITY_ABOVE_NORMAL	14
	THREAD_PRIORITY_HIGHEST	15
	THREAD_PRIORITY_TIME_CRITICAL	15
REAL-TIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	16
	THREAD_PRIORITY_LOWEST	22
	THREAD_PRIORITY_BELOW_NORMAL	23
	THREAD_PRIORITY_NORMAL	24
	THREAD_PRIORITY_ABOVE_NORMAL	25
	THREAD_PRIORITY_HIGHEST	26
	THREAD_PRIORITY_TIME_CRITICAL	31

### 1.3.9 Динамічні пріоритети

Кожний потік має *динамічний пріоритет* — значення пріоритету, яке планувальник використовує для визначення наступного потоку для виконання. Відразу після створення потоку його динамічний пріоритет дорівнює базовому пріоритету. ОС може збільшувати та зменшувати динамічний пріоритет потоку, щоб забезпечити високий рівень його здатності до реагування та більш-менш рівномірний розподіл процесорного часу між потоками. При цьому ОС не збільшує пріоритет потоку, якщо його базовий пріоритет лежить у межах від 16 до 31 включно.

ОС збільшує динамічний пріоритет потоку для підвищення його рівня реагування наступним чином:

- коли процес класу пріоритету *NORMAL\_PRIORITY\_CLASS* виходить на передній план, планувальник збільшує клас пріоритету процесу, асоційованого з вікном переднього плану, до значення, вищого за або рівного значенню класу пріоритету будь-якого фонових процесу. Клас пріоритету зменшується до попереднього значення, коли процес більше не перебуває на передньому плані;
- коли вікно отримує введення (наприклад, повідомлення від таймера, миші чи клавіатури), планувальник збільшує пріоритет потоку, який володіє цим вікном;
- коли настала подія, яку чекає заблокований потік (наприклад, коли завершено операцію введення-виведення), планувальник збільшує його пріоритет.

Збільшивши динамічний пріоритет потоку, планувальник по завершенню кожного кванту часу зменшує цей пріоритет на одиницю доти, доки динамічний пріоритет не впаде до базового. Динамічний пріоритет потоку не може бути меншим за базовий.

### 1.3.10 Перемикання контексту

Планувальник підтримує *чергу* виконуваних потоків для кожного (динамічного) пріоритету. Такі потоки називають *готовими до виконання* (ready threads). Коли процесор стає доступним, ОС виконує *перемикання контексту*, яке передбачає виконання таких кроків:

- зберегти контекст потоку, який щойно призупинив виконання;
- помістити цей потік у кінець черги з відповідним пріоритетом;
- знайти чергу найвищого пріоритету, яка містить готові до виконання потоки;

г) вибрати з голови черги потік, завантажити його контекст та виконати його. Перемикання контексту, як правило, відбувається у випадку:

- завершення кванту часу поточного потоку;
- появи у відповідній черзі готового до виконання потоку з пріоритетом, вищим за пріоритет поточного потоку;
- блокування поточного потоку для очікування настання певної події (у цьому разі потік вивільнює решту свого кванту часу).

Певні потоки не вважаються готовими, зокрема:

- потоки, які ждуть синхронізації уведення або деякого об'єкта;
- потоки, виконання яких призупинено за допомогою функцій `SuspendThread` чи `SwitchToThread` (див. розділ 1.3.14);
- потоки, створені з прапорцем `CREATE_SUSPENDED` (див. розділ 1.3.13).

Доти, доки заблоковані потоки чи потоки, виконання яких призупинено, не стануть готові до виконання, планувальник не виділить їм процесорний час, не зважаючи на їхній пріоритет.

### 1.3.11 Інверсія пріоритету

Описаний вище алгоритм, за яким планувальник вибирає черговий потік для запуску, може в окремих випадках призводити до взаємоблокування потоків. У таких ситуаціях ОС здійснює *інверсію пріоритетів*.

Розгляньмо цей підхід на простому прикладі. Нехай маємо три потоки. Потік 1 має високий пріоритет і готовий до виконання, потік 2 має низький пріоритет та виконує код із критичної секції (див. розділ 1.3.15), а потік 3 має середній пріоритет. Потік 1 починає ждати звільнення спільного ресурсу, який займає потік 2. Оскільки потік 1 заблоковано, потік 3 отримує весь процесорний час. Оскільки потік 2 має пріоритет, менший за пріоритет потоку 3, він ніколи не покине критичну секцію, і потік 1 ніколи не продовжить своє виконання.

Планувальник розв'язує проблеми такого роду шляхом випадкового збільшення пріоритету готових до виконання потоків. У такому разі низькопріоритетні потоки отримають можливість покинути критичну секцію, і таким чином уможливити виконання заблокованих потоків.

### 1.3.12 Засоби Windows API для роботи з потоками

### 1.3.13 Створення потоків

Для створення потоків у Windows API передбачено функцію `CreateThread`. Ця функція має наступний формат:

```
HANDLE WINAPI CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

Прокоментуймо призначення аргументів цієї функції:

- `lpThreadAttributes` — указівник на структуру типу `SECURITY_ATTRIBUTES`, який для цілей даної лабораторної роботи можна встановити в значення `NULL`;
- `dwStackSize` — початковий розмір стеку потоку. Якщо це значення дорівнює 0, буде використано значення за замовчуванням;
- `lpStartAddress` — указівник на користувацьку функцію, яку повинен виконати створюваний потік. Фактично, цей указівник визначає початкову адресу потоку. У якості функції, на яку вказує цей аргумент, може бути будь-яка функція, що має наступний синтаксис:

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

Один процес може мати декілька потоків, які виконують одну й ту саму функцію;

- `lpParameter` — указівник на аргумент функції, на яку вказує `lpStartAddress`;
- `dwCreationFlags` — прапорці, які управляють створенням потоку. Якщо значення цього аргумента дорівнює 0, потік починає виконання відразу після створення, якщо значення дорівнює `CREATE_SUSPENDED`, потік створено в заблокованому стані;
- `lpThreadId` — указівник на змінну, у яку буде записано ідентифікатор потоку. Якщо цей аргумент дорівнює `NULL`, ідентифікатор потоку нікуди не буде записано.



Якщо функцію `CreateThread` завершено успішно, вона повертає *дескриптор* новоствореного потоку. Інакше, функція повертає значення `NULL`.

### Приклад 1.1.

Розгляньмо вихідний текст програми на мові програмування C, що демонструє створення нового потоку, який виконує локально визначену функцію `ExampleThread`. У програмі, потік створюють відразу після створення головного вікна програми. Потік виконує деяку тривалу обчислювальну операцію, роль якої в даному прикладі грає послідовне обчислення загальновідомої функції Еккермана (Ackermann's function). По завершенню виконання цієї операції програма видає повідомлення з текстом «The lengthy operation completed!» (рисунок 1.1).

Варто зазначити, що якби виконання цієї тривалої операції не було винесено в окремий потік, аплікація на час її виконання втратила б здатність реагувати на введення користувача (вікно аплікації неможливо було б перемістити, згорнути і навіть закрити). Винесення тривалих операцій в окремі потоки дозволяє забезпечити активність інтерфейсу аплікації протягом усього часу її виконання.

```
#include "stdafx.h"
#include <windows.h>
ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
DWORD WINAPI ExampleThread(LPVOID pvoid);
unsigned int ackermann(unsigned int m, unsigned int n);
//глобальні змінні
HINSTANCE hInst; //поточний екземпляр аплікації
TCHAR szAppName[] = TEXT("CreateThreadExample"); //заголовок головного вікна
TCHAR szWindowClass[] = TEXT("CreateThreadWindow"); //ім'я класу головного вікна
HWND hWnd; //дескриптор головного вікна
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);
    MSG msg;
    //реєструємо віконний клас головно вікна аплікації
    MyRegisterClass(hInstance);
    //виконуємо ініціалізацію аплікації
    if(!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }
    //цикл опрацювання повідомлень
    BOOL bRet;
    while((bRet = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0)
    {
        if(bRet == -1)
        {
            //in case of an error
            return FALSE;
        }
        else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    //повертаємо код виходу
    return (int) msg.wParam;
}
//функція для реєстрації нового віконного класу
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wndclass;
    wndclass.cbSize = sizeof(WNDCLASSEX);
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = NULL;
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szWindowClass;
    wndclass.hIconSm = NULL;
    return RegisterClassEx(&wndclass);
}
```

```

}
//функція для створення вікна
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    //зберігаємо дескриптор екземпляра програми в глобальній змінній
    hInst = hInstance;
    //створюємо вікно засобами Windows API
    hWnd = CreateWindow(szWindowClass, szAppName, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, 600, 300, NULL, NULL, hInstance, NULL);
    if(!hWnd) {
        //якщо сталася помилка
        return FALSE;
    }
    //виводимо вікно на екран
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    return TRUE;
}
//віконна процедура
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    RECT* lpRect = NULL;
    //аналізуємо тип повідомлення
    switch(message)
    {
        case WM_CREATE:
            //якщо треба опрацювати повідомлення створення
            //створюємо потік для тривалих обчислень
            CreateThread(
                NULL, //атрибути безпеки не потрібні
                0, //розмір стеку за замовчуванням
                ExampleThread, //вказівник на локальну функцію
                NULL, //ця функція не має аргументів
                0, //потік починає виконання відразу після створення
                NULL //нам не потрібний ідентифікатор потоку
            );
            return 0;
        case WM_SIZE:
            //якщо треба опрацювати повідомлення зміни розміру вікна
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;
        case WM_DESTROY:
            //якщо треба опрацювати повідомлення знищення вікна
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
//функція --- початкова адреса потоку
DWORD WINAPI ExampleThread(LPVOID pvoid)
{
    for(int i = 0; i < 10; i++)
    {
        for(unsigned int m = 0; m < 4; ++m)
        {
            for(unsigned int n = 0; n < 10; ++n)
            {
                ackermann(m, n);
            }
        }
    }
    MessageBox(hwnd, L"The lengthy operation completed!",
        L"Notification", MB_OK | MB_ICONINFORMATION);
    return 0; }
//функція Еккерманна
unsigned int ackermann(unsigned int m, unsigned int n)
{
    if(m == 0) {
        return n + 1; }
    if(n == 0) {
        return ackermann(m - 1, 1);
    }
    return ackermann(m - 1, ackermann(m, n - 1));
}

```

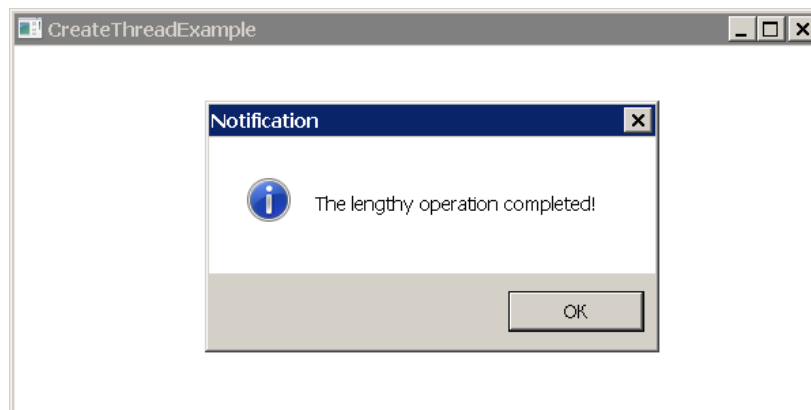


Рисунок 1.1 – Поява повідомлення в програмі з Прикладу 1.1

#### 1.3.14 Призупинення виконання потоку

Будь-який потік може призупинити та відновлювати виконання іншого потоку. Як було зазначено раніше, якщо потік призупинено, ОС не виділяє йому процесорний час.

Для призупинення дії потоку використовують функцію `Suspend-Thread`. Ця функція має наступний формат:

`DWORD WINAPI SuspendThread(HANDLE hThread);`

Єдиним аргументом цієї функції є дескриптор потоку, який потрібно призупинити. У випадку помилки функція повертає значення `-1`. У випадку успішного завершення, функція збільшує на 1 лічильник *призупинень* відповідного потоку та повертає значення цього лічильника до збільшення.

Для відновлення дії потоку використовують функцію `ResumeThread`. Формат цієї функції наступний:

`DWORD WINAPI ResumeThread(HANDLE hThread);`

Аргумент `hThread` цієї функції визначає дескриптор потоку, виконання якого потрібно відновити. У випадку помилки функція повертає значення `DWORD - 1`. У випадку успішного завершення, функція зменшує на одиницю лічильник *призупинень* відповідного потоку та повертає значення цього лічильника до зменшення.

Варто зазначити, що виконання потоку з дескриптором `hThread` буде відновлено тільки тоді, коли значення лічильника *призупинень* буде зменшено до 0.

Інший спосіб призупинити виконання потоку — задати під час його створення прапорець `CREATE_SUSPENDED` (див. розділ 1.3.13). У цьому разі він не починає виконання доти, доки якийсь інший потік не викличе функцію відновлення потоку `ResumeThread` з дескриптором потоку в якості свого аргумента.

Створення потоку в заблокованому стані з наступним відновленням його виконання з іншого потоку може бути доцільним для:

- ініціалізації стану потоку перед початком його виконання;
- одноразової синхронізації.

Варто зазначити, що використання функції `SuspendThread` не дозволяє досягти одноразової синхронізації, оскільки під час виконання аплікації неможливо визначити, у якій точці коду буде здійснено призупинення потоку. Натомість, застосування підходу з призупиненням виконання потоку в момент його створення гарантує, що потік буде призупинено в початковій точці коду.

Потік може тимчасово призупинити власне виконання на деякий інтервал часу за допомогою функцій `Sleep` та `SleepEx`. Функція `Sleep` має наступний формат:

`VOID WINAPI Sleep(DWORD dwMilliseconds);`

Єдиний аргумент цієї функції — величина інтервалу часу (у мілісекундах), протягом якого потік, який викликав функцію, буде призупинено. Розгляд функції `SleepEx` виходить за межі даних методичних рекомендацій.

Використання функцій `Sleep` та `SleepEx` корисно, коли потік очікує на введення від користувача. У такому разі потік може бути призупинено на період часу, достатній для того, щоб користувач зробив дії, які від нього вимагають.

Схожою на `Sleep` та `SleepEx` є функція `SwitchToThread`, яка дозволяє призупинити виконання потоку, який її викликав, але при цьому непотрібно вказувати інтервал часу, на який виконання потрібно призупинити. Формат цієї функції наступний:

BOOL WINAPI SwitchToThread(void);

У результаті успішного виконання функції, ОС призупиняє виконання потоку, який її викликав, та запускає на виконання деякий інший потік. У такому разі функція повертає ненульове значення. Якщо ж у системі немає жодного готового до виконання потоку, ОС не призупинить виконання потоку, який викликав цю функцію, і вона поверне значення 0.

### 1.3.15 Синхронізація під час виконання потоків

Для того, щоб упередити появу гонок (race conditions) та взаємоблокування (deadlocks) під час виконання декількох потоків, потрібно *синхронізувати* доступ потоків до спільних ресурсів. Синхронізацію також потрібно здійснювати, щоб забезпечити виконання взаємопов'язаного коду в правильній послідовності.

Наприклад, у деякій аплікації один потік обчислює координати певної фігури в деякій глобальній структурі даних, а інший потік використовує ці координати для малювання фігури. Якщо перший потік устиг змінити одну з координат, але не встиг змінити іншу (оскільки його квант часу завершився раніше), то другий потік може використати для малювання некоректні координати.

Windows API надає можливість використовувати низку об'єктів, дескриптори яких можна використовувати для синхронізації під час виконання декількох потоків:

- вхідні буфери консолі (console input buffers);
- події (events);
- м'ютекси (mutexes);
- процеси (processes);
- семафори (semaphores);
- потоки (threads);
- таймери (timers).

Деякі з цих об'єктів зручно використовувати для блокування виконання потоку до моменту настання певної події. Наприклад, дескриптор вхідного буфера консолі сигналізує, коли наявне незчитане уведення (натискання клавіші або клік мишею), а дескриптори процесів та потоків сигнализують, коли їх виконання завершено. У даній лабораторній роботі ми не розглядатимемо такі способи синхронізації.

Інші об'єкти зручно використовувати для захисту спільних ресурсів від одночасного доступу. Наприклад, декілька потоків можуть володіти дескриптором одного й того ж об'єкта м'ютекса. Перед доступом до спільного ресурсу потоки повинні викликати спеціальну функцію очікування (wait function) для того, щоб діждатися, коли стан м'ютекса буде сигналізований. Коли стан м'ютекса сигналізований, тільки один із очікуючих потоків може отримати доступ до ресурсу. Стан м'ютекса миттєво змінюється на несигналізований, і всі інші потоки залишаються в заблокованому стані. Коли потік завершує використання ресурсу, він повинен самостійно встановити стан м'ютекса в сигналізований, щоб інші потоки також могли отримати доступ до ресурсу.

У даній лабораторній роботі ми розглядатимемо інший спосіб здійснення синхронізації доступу до спільних ресурсів — використання *критичних секцій* (critical sections). Під критичною секцією потрібно розуміти ділянку коду, яку одночасно може виконувати тільки один потік.

Критичні секції працюють подібно м'ютексам, але цей підхід застосовний тільки для потоків, які належать одному процесу. Підхід із використанням критичних секцій забезпечує продуктивніший спосіб синхронізації, оскільки, на відміну від м'ютексів, операції з критичною секцією не потребують звернення до ядра ОС.

Для того, щоб використовувати критичну секцію, спочатку потрібно декларувати структуру типу CRITICAL\_SECTION. Після цього критичну секцію потрібно *ініціалізувати*. Це можна зробити за допомогою функції InitializeCriticalSection. Ця функція має наступний формат:

```
void WINAPI InitializeCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
);
```

Аргумент lpCriticalSection цієї функції — указівник на структуру типу CRITICAL\_SECTION. У результаті успішного виконання буде проініціалізовано критичну секцію, на яку вказує lpCriticalSection.

Після ініціалізації критичної секції, потік може увійти в неї шляхом викликання функції EnterCriticalSection. Ця функція має наступний формат:

```
void WINAPI EnterCriticalSection(  

```

LPCriticalSection

);

Аргумент `lpCriticalSection` цієї функції має інтерпретацію, аналогічну однойменному атрибуту попередньої функції.

Якщо деякий потік увійшов у деяку критичну секцію, будь-який інший потік, який намагатиметься викликати функцію `EnterCriticalSection` зі вказівником на структуру, що відповідає цій критичній секції, буде заблоковано. Це дозволяє забезпечити синхронізоване використання спільного ресурсу різними потоками. Заблокований потік продовжить своє виконання тільки в тому випадку, коли потік, що займав критичну секцію, покине її шляхом викликання функції `LeaveCriticalSection`. Формат цієї функції наступний:

```
void WINAPI LeaveCriticalSection(  
    LPCriticalSection  
);
```

Аргумент `lpCriticalSection` цієї функції має інтерпретацію, аналогічну однойменному атрибуту попередньої функції.

Варто зауважити, що якщо потік викличе функцію `lpCriticalSection` зі вказівником на структуру, що відповідає критичній секції, у яку він не увійшов раніше за допомогою функції `EnterCriticalSection`, то станеться помилка, яка може змусити інші потоки, які намагатимуться увійти в цю критичну секцію, бути постійно заблокованими.

Якщо критична секція більше не потрібна, видалити її можна за допомогою функції `DeleteCriticalSection`. Ця функція має наступний формат:

```
void WINAPI DeleteCriticalSection(  
    LPCriticalSection  
);
```

Аргумент `lpCriticalSection` цієї функції має інтерпретацію, аналогічну однойменному атрибуту попередньої функції.

У випадку успішного завершення, функція вивільнить усі системні ресурси, пов'язані з відповідною критичною секцією. Використання змінної критичної секції після її видалення за допомогою цієї функції може призвести до непередбачуваних наслідків. У випадку здійснення спроби видалення критичної секції в той час, як у ній перебуває деякий потік, стан заблокованих через цю критичну секцію потоків буде невизначеним.

У деяких випадках корисно організувати взаємодію потоків таким чином, щоб потік у випадку неможливості увійти в критичну секцію залишався готовим до виконання. Для цього потік повинен замість функції `EnterCriticalSection` викликати функцію `TryEnterCriticalSection`. Формат цієї функції наступний:

```
BOOL WINAPI TryEnterCriticalSection(  
    LPCriticalSection  
);
```

Аргумент `lpCriticalSection` цієї функції має інтерпретацію, аналогічну однойменному атрибуту попередньої функції. Якщо відповідна критична секція вільна, і потік може в неї зайти, чи якщо потік уже перебуває в цій критичній секції, функція повертає ненульове значення. Якщо ж деякий інший потік перебуває у відповідній критичній секції, то функція повертає 0.

#### Приклад 4.2.

Розгляньмо вихідний текст консольної програми Windows API на мові програмування C++, що демонструє використання критичної секції для синхронізації доступу потоків до спільного ресурсу.

У даній програмі спільним ресурсом є цілочисельна змінна `counter`. У програмі створюють два потоки, кожен із яких виконує одну й ту саму функцію `ExampleThread`. У цій функції потік спочатку збільшує значення `counter`, а потім виводить нове значення в консоль. Застосування критичних секцій забезпечує виведення в консоль послідовно зростаючих значень `counter` (рисунки 1.2). Якби критичні секції на було застосовано, то виконання певного потоку могло би бути призупинено в момент після оновлення значення `counter`, але до виведення відповідного значення в консоль. У такому разі значення, які було б виведено в консоль, не були б послідовно зростаючими (рисунки 1.3).

У цій програмі також використано функцію `WaitForSingleObject`.

Перший аргумент цієї функції — дескриптор об'єкта, стан якого відслідковує функція. Функція завершить своє виконання або якщо стан цього об'єкта стане сигналізований, або якщо сплине час, визначений як її другий аргумент. У програмі цю функцію використано для того, щоб головний потік спочатку діждався завершення обох дочірніх потоків, і тільки після цього продовжив своє виконання.

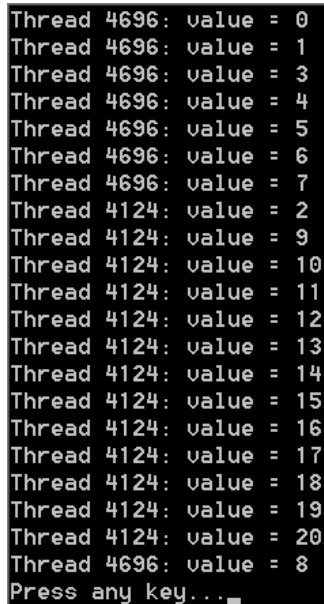
```
Thread 1212: counter = 1
Thread 768: counter = 2
Thread 1212: counter = 3
Thread 768: counter = 4
Thread 1212: counter = 5
Thread 768: counter = 6
Thread 1212: counter = 7
Thread 768: counter = 8
Thread 1212: counter = 9
Thread 768: counter = 10
Thread 1212: counter = 11
Thread 768: counter = 12
Thread 1212: counter = 13
Thread 768: counter = 14
Thread 1212: counter = 15
Thread 768: counter = 16
Thread 1212: counter = 17
Thread 768: counter = 18
Thread 1212: counter = 19
Thread 768: counter = 20
Thread 1212: counter = 21
Thread 768: counter = 22
Press any key..._
```

Рисунок 1.2 – Результат роботи програми з Прикладу 1.2 в синхронному режимі

```
#include "stdafx.h"
#include <Windows.h>
#include <iostream>
using namespace std;
int counter = 0;
CRITICAL_SECTION cs;
DWORD WINAPI ExampleThread(LPVOID pvoid);

int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE myhandle1, myhandle2;
    //ініціалізуємо критичну секцію
    InitializeCriticalSection(&cs);
    //створюємо два потоки, кожний із яких виконує функцію ExampleThread
    myhandle1 = CreateThread(NULL, 0, ExampleThread, NULL, 0, NULL);
    myhandle2 = CreateThread(NULL, 0, ExampleThread, NULL, 0, NULL);
    //дожидаємося, поки обидва потоки завершать своє виконання
    WaitForSingleObject(myhandle1, INFINITE);
    WaitForSingleObject(myhandle2, INFINITE);
    //закриваємо дескриптори потоків
    CloseHandle(myhandle1);
    CloseHandle(myhandle2);
    cout << "Press any key...";
    getchar();
    //видаляємо критичну секцію
    DeleteCriticalSection(&cs);
    //повертаємо код виходу
    return 0; }
//функція --- початкова адреса потоку
DWORD WINAPI ExampleThread(LPVOID pvoid)
{
    int value;
    while(counter <= 20)
    {
        //у циклі, поки лічильник не стане 20
        //пробуємо увійти в критичну секцію
        while(!TryEnterCriticalSection(&cs));
        //збільшуємо лічильник
        value = counter++;
        //виводимо його значення в консоль
        cout << "Thread " << GetCurrentThreadId() << ": value = " << value << endl;
        //виходимо з критичної секції
        LeaveCriticalSection(&cs);
    }
}
```

```
return 0; }
```



```
Thread 4696: value = 0
Thread 4696: value = 1
Thread 4696: value = 3
Thread 4696: value = 4
Thread 4696: value = 5
Thread 4696: value = 6
Thread 4696: value = 7
Thread 4124: value = 2
Thread 4124: value = 9
Thread 4124: value = 10
Thread 4124: value = 11
Thread 4124: value = 12
Thread 4124: value = 13
Thread 4124: value = 14
Thread 4124: value = 15
Thread 4124: value = 16
Thread 4124: value = 17
Thread 4124: value = 18
Thread 4124: value = 19
Thread 4124: value = 20
Thread 4696: value = 8
Press any key...
```

Рисунок 1.3 – Результат роботи програми з Прикладу 1.2 в асинхронному режимі

### 1.3.16 Завершення виконання потоку

Будь-який потік продовжує виконання до моменту настання однієї з наступних подій:

- цей потік викликає функцію `ExitThread`. Ця функція має наступний формат:

```
VOID WINAPI ExitThread(DWORD dwExitCode);
```

Аргумент `dwExitCode` цієї функції визначає код завершення для потоку (див. нижче);

- будь-який потік процесу, якому належить цей потік, викликає функцію `ExitProcess`. Ця функція має наступний формат:

```
VOID WINAPI ExitProcess(UINT uExitCode);
```

Аргумент `uExitCode` цієї функції визначає код завершення для процесу та всіх його потоків (див. нижче);

- функція, яку виконує цей потік, завершує свою роботу;

- будь-який інший потік викликає функцію `TerminateThread`, передавши в якості аргумента дескриптор цього потоку. Ця функція має наступний формат:

```
BOOL WINAPI TerminateThread(
```

```
    HANDLE hThread,
```

```
    DWORD dwExitCode
```

```
);
```

Аргумент `hThread` цієї функції визначає дескриптор потоку, виконання якого потрібно завершити, аргумент `dwExitCode` — код завершення для потоку (див. нижче). У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0;

- будь-який інший потік викликає функцію `TerminateProcess`, передавши в якості аргумента дескриптор процесу, якому належить цей потік. Ця функція має наступний формат:

```
BOOL WINAPI TerminateProcess(
```

```
    HANDLE hProcess,
```

```
    UINT uExitCode
```

```
);
```

Аргумент `hProcess` цієї функції визначає дескриптор процесу, виконання якого потрібно завершити, аргумент `uExitCode` — код завершення для процесу та всіх його потоків (див. нижче). У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0;

Дізнатися статус завершення потоку можна за допомогою функції `GetExitCodeThread`. Ця функція має наступний формат:

```
BOOL WINAPI GetExitCodeThread(
```

```
HANDLE hThread,  
LPDWORD lpExitCode  
);
```

Аргумент `hThread` цієї функції визначає дескриптор потоку, статус завершення якого потрібно дізнатися, аргумент `lpExitCode` — вказівник на змінну, у яку буде записано отриманий статус. У випадку успішного завершення, функція повертає ненульове значення. У випадку помилки функція повертає 0.

Поки потік виконується, його статус завершення дорівнює `STILL_ACTIVE`. Коли потік завершує виконання, його статус завершення дорівнює коду виходу. Якщо потік було завершено за допомогою функцій `ExitThread`, `ExitProcess`, `TerminateThread` чи `TerminateProcess`, його код виходу дорівнює відповідному аргументу кожної з цих функцій. Якщо потік завершено тому, що функція, яку він виконує, завершила свою роботу, його код виходу дорівнює значенню, яке повернула ця функція.

Функції `TerminateThread` та `TerminateProcess` потрібно застосовувати у виключних випадках, оскільки, на відміну від функцій `ExitThread` та `ExitProcess`, вони не інформують приєднані динамічні бібліотеки (DLL) про завершення виконання потоку та не вивільнюють початковий стек. Окрім того, дескриптори об'єктів, якими володів потік, виконання якого завершено таким чином, залишаються відкритими до завершення всього процесу.