МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ОЛЕСЯ ГОНЧАРА

Матвєєва Н.О., Пономарьов І.В.

РОЗРОБКА БАГАТОПОТОЧНИХ ТА ПАРАЛЕЛЬНИХ ПРОГРАМ НА ПЛАТФОРМІ .NET

Навчальний посібник

Дніпро

2022

УДК 004.432/[004.451.45+004.272.2]

Друкується за рішенням Вченої ради
Дніпровського національного університету імені Олеся Гончара
протокол № 10 від 29 червня 2021 року

Рецензенти:

Гаврилюк Володимир Іванович — доктор технічних наук, професор, завідувач кафедри інформаційних технологій і систем Національної металургійної академії України.

Гнатушенко Володимир Володимирович — доктор технічних наук, професор кафедри інформаційних систем та технологій Національного технічного університету «Дніпровська політехніка».

Матвєєва Н.О., Пономарьов І.В. Розробка багатопокових програм на платформі .NET: навчальний посібник — Дніпро, 2022. — 169с.

Розглянуті можливості паралельного програмування на мові C# для розробки ефективних, добре масштабованих додатків.

Наводяться особливості багатопоточного програмування на платформі .NET: способи створення нових потоків та процесів, їх взаємодії, обмін даними та різні механізми синхронізації. Докладно розглядається бібліотека розпаралелювання задач TPL, яка удосконалює багатопоточне програмування двома основними способами: спрощує створення і застосування багатьох потоків та дозволяє автоматично використовувати декілька процесорів. Описано можливості паралельної мови інтегрованих запитів PLINQ, що застосовуються для досягнення паралелізму даних всередині запиту.

Для студентів технічних спеціальностей, які входять до галузі знань 12 "Інформаційні технології".

ЗМІСТ

ВСТУП

У сучасних багатоядерних комп'ютерах програма розбивається на частини, які можуть виконуватися одночасно на кількох ядрах процесорів. Це дає можливість прискорити обробку обчислень, зменшити час відклику програми. Паралельне програмування є загальноприйнятим методом розробки добре масштабованих додатків.

У мові програмування C# присутні спеціальні засоби, а у платформі .Net визначено певні класи для підтримки паралельного виконання програми. Ця підтримка паралельного програмування полегшує розробку ефективних програм.

У загальному випадку, процес виконання програми може бути послідовним або паралельним.

Послідовний (однопотоковий) процес має тільки один потік управління, що характеризується зміною його лічильника команд. Потік - це особливого роду паралельне виконання деякої функції, яка запускається у головному процесі та виконується в тому ж адресному просторі, що і батьківський процес.

Однопоточний процес використовує код, дані в основній пам'яті, певні значення регістрів, стек, в якому зберігаються локальні змінні, та файли, з якими він працює.

Багатопоточний процес має декілька паралельних потоків, для кожного з яких операційна система створює свій стек і зберігає свої власні значення регістрів. Потоки працюють в загальній основний пам'яті і використовують той же адресний простір, що і процес-батько, а також поділяють код процесу і файли. Схема організації однопоточного та багатопоточного процесів зображена на рис. 1.

Багатопоточність має великі переваги над послідовним виконанням програми, а саме:

- збільшення швидкості виконання програми, так як потоки працюють в загальному просторі віртуальної пам'яті;
- використання загальних ресурсів. Потоки одного процесу використовують загальну пам'ять і файли;
- економія пам'яті та часу, тому що перемикання контексту на поток, для якого потрібна тільки зміна стеку та відновлення значення регістрів, значно швидше, ніж на звичайний процес.

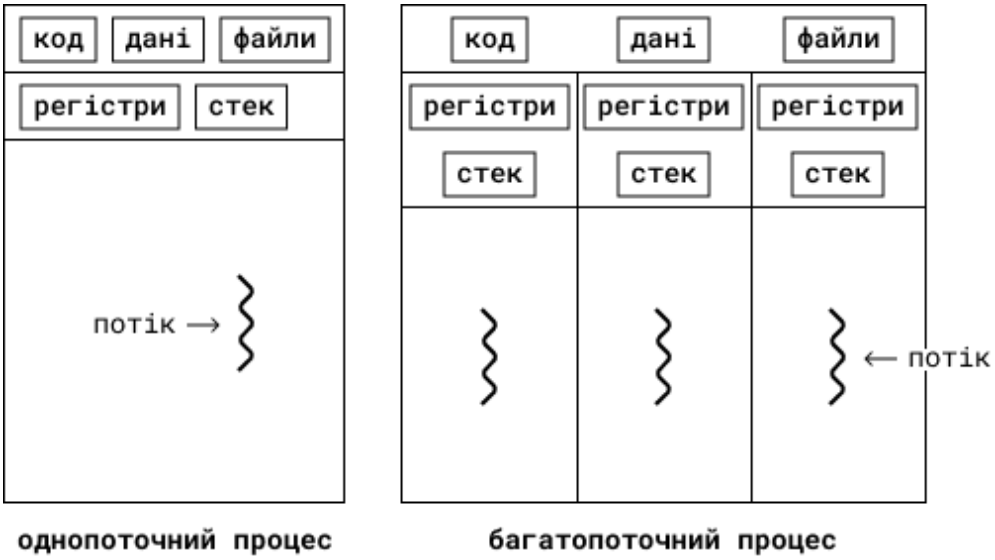


Рис. 1. Схема організації однопоточного та багатопоточного процесів

Головна перевага багатопотокової обробки полягає в тому, що вона дозволяє писати ефективні програми, завдяки можливості використовувати час простою, який неминуче виникає в ході виконання більшості програм. Як відомо, більшість пристроїв введення-виведення, будь то клавіатура, накопичувачі на дисках або пристрої, підключені до мережевих портів, працюють набагато повільніше, ніж центральний процесор. Тому більшу частину свого часу програмі доводиться очікувати відправки даних на пристрій вводу-виводу або прийому інформації з нього. А завдяки багатопотоковій обробці програма може вирішувати якусь іншу задачу під час вимушеного простою. Наприклад, в той час як одна частина програми виконує

читання текстової інформації, котра вводиться з клавіатури, інша її частина може відправляти файл через з'єднання з Інтернетом, а третя - здійснювати буферизацію чергового блоку даних, що відправляється.

1. ТЕОРЕТИЧНА ЧАСТИНА

1.1. Потоки

1.1.1. Визначення потоку

Потік (thread) являє собою незалежну послідовність інструкцій в програмі. Потоки дозволяють програмі одночасно виконувати різні роботи.

Наприклад, під час введення якогось коду мови програмування C# у вікні редактора Visual Studio окремим фоновим потоком проводиться аналіз на предмет синтаксичних помилок. Те ж саме відбувається і в засобі перевірки орфографії в Microsoft Word. Один потік очікує введення даних користувачем, а інший в цей час виконує у фоновому режимі деякий аналіз. Третій потік може записувати дані в тимчасовий файл, а четвертий - завантажувати додаткові дані з Інтернету [2].

У додатку, який функціонує на сервері, один потік завжди очікує надходження запиту від клієнта (потік-слухач). При отриманні запиту він відразу ж пересилає його окремому робочому потоку, який далі сам продовжує взаємодіяти з клієнтом. Потік-слухач після цього повертається до своїх обов'язків по очікуванню надходження наступного запиту від чергового клієнта.

Додаток, що виконується називається **процесом.** Кожен процес має виділену область у віртуальній пам'яті і містить один чи кілька потоків. Наявність хоча б одного потоку є обов'язковим для виконання будь-якого додатку. Потоки плануються до виконання операційною системою (ОС).

Потік має **пріоритет**, **лічильник команд**, який вказує на місце в програмі, де відбувається обробка, і **стек**, в якому зберігаються локальні змінні потоку. Стек у кожного потоку виглядає по-своєму, але **пам'ять для**

програмного коду і купа поділяються серед всіх потоків, які функціонують усередині одного процесу [2].

Це дозволяє потокам всередині одного процесу швидко взаємодіяти між собою, оскільки всі потоки процесу звертаються до однієї і тієї ж віртуальної пам'яті. Однак це також і ускладнює справу, оскільки дає можливість безлічі потоків змінювати одну і ту ж область пам'яті.

Розрізняють два різновиди багатозадачності:

- на основі процесів;
- на основі потоків.

При організації багатозадачності на основі процесів програма є найменшою одиницею коду, виконання якої може координувати планувальник завдань ОС. На комп'ютері можуть паралельно виконуватися дві і більше програм. Наприклад, можливо одночасно виконувати програми електронних таблиць, текстового редактора і перегляду вмісту в Інтернеті.

Потік являє собою координовану одиницю виконуваного коду. При організації багатозадачності на основі потоків, процес може мати більше одного потоку і це означає, що в одній програмі одночасно можуть вирішуватися два завдання і більше. Наприклад, текст може форматуватися в редакторі тексту одночасно з його виводом на друк, за умови, що обидві ці дії виконуються в двох окремих потоках.

Таким чином, відмінності в багатозадачності на основі процесів і потоків можуть бути зведені до наступного:

- багатозадачність на основі **процесів** організовується для **паралельного виконання програм;**
- багатозадачність на основі **потоків** - для **паралельного виконання окремих частин однієї програми**.

Потік може перебувати в одному з декількох станів:

- потік **виконуються**;
- **потік готовий до виконання**, як тільки він отримає час і ресурси центрального процесора (ЦП);
- **потік призупинений**, тобто тимчасово не виконуватися, але може бути відновленим в подальшому;
- **потік заблокований в очікуванні ресурсів** для свого виконання;
- **потік завершений**, коли його виконання закінчено і не може бути відновлено.

У середовищі .NET визначені два різновиди потоків:

- пріоритетний;
- фоновий.

За замовчуванням, коли потік створюється, він автоматично стає **пріоритетним**, але його можна зробити фоновим. Відмінність пріоритетних потоків від фонових полягає в тому, що фоновий потік автоматично завершується, якщо в його процесі зупинені всі пріоритетні потоки.

При організації багатозадачності на основі потоків виникає потреба в особливого роду режимі, який називається **синхронізацією** і дозволяє координувати виконання потоків певним чином. Для такої синхронізації в мові програмування C# передбачена окрема підсистема.

Всі процеси складаються хоча б з одного потоку, який зазвичай називають **основним**, оскільки саме з нього починається виконання програми. З основного потоку можна створити інші потоки.

У мові C# і середовищі .NET підтримуються обидва різновиди багатозадачності: **на основі процесів** і **на основі потоків**. Тому засобами C# можна створювати як процеси, так і потоки, а також керувати ними [7,8].

Класи, які підтримують багатопотокове програмування, визначені в просторі імен *System.Threading*. Мабуть, головним серед них є клас **Thread**, оскільки він представляє окремий потік.

Щоб програмно отримати посилання на потік, що виконується конкретним його примірником, необхідно викликати статичну властивість Thread.CurrentThread:

```
Thread currThread = Thread.CurrentThread;
```

При запуску додатку, написаного на C#, операційна система створює процес, а середовище CLR створює всередині цього процесу логічний контейнер, який називається доменом додатків (AppDomain) і всередині якого працює запущений додаток. На платформі .NET не існує прямої відповідності "один до одного" між доменами додатків і потоками. Фактично певний домен додатків може мати кілька потоків, що виконуються в кожен конкретний момент часу. Конкретний потік не прив'язаний до одного домену додатків протягом свого часу існування. Потоки можуть перетинати кордони доменів додатків, коли це заманеться планувальнику Windows та середовищу виконання CLR [2].

Хоча активні потоки можуть перетинати кордони домену додатків, кожен потік в кожен конкретний момент часу може виконуватися тільки всередині одного домену додатків (іншими словами, неможливо, щоб один потік працював в більш ніж одному домені додатків відразу). Щоб програмно отримати доступ до домену додатків, в якому працює поточний потік, треба викликати статичний метод Thread.GetDomain():

```
AppDomain ad = Thread.GetDomain();
```

Єдиний потік також в будь-який момент може бути переміщений в певний контекст, і він може переміщатися в межах нового контексту за примхою CLR. Для отримання поточного контексту, в якому виконується потік, використовується статична властивість Thread.CurrentContext (яке повертає об'єкт System.Runtime.Remoting.Contexts.Context):

Context ctx = Thread.CurrentContext;

1.1.2. Створення та завершення потоку

Для створення потоку досить отримати екземпляр об'єкта типу Thread, визначеного в просторі імен System.Threading. Нижче наведена найпростіша форма конструктора класу Thread:

public Thread (ThreadStart запуск)

де запуск - це ім'я методу, що викликається з метою почати виконання потоку, а ThreadStart - делегат, визначений в середовищі .NET, як показано нижче.

public delegate void ThreadStart()

Отже, метод, що вказується в якості точки входу до потіку, повинен **повертати тип void** і **не приймати ніяких аргументів**.

Новостворений потік не почне виконуватися до того часу, поки не буде викликаний його метод Start(), який визначається в класі Thread.

Одного разу почавшись, потік буде виконуватися до того часу, поки не відбудеться повернення з методу, на який вказує запуск. Після повернення з цього методу потік автоматично припиняється. Якщо ж спробувати викликати метод Start() для потоку, який вже почався, це призведе до генерування винятку ThreadStateException [2,9].

Іноді потік корисно перервати до його нормального завершення. Наприклад, отладчику може знадобитися перервати потік, який вийшов з-під контролю. Після переривання потік видаляється з системи і не може бути розпочато знову.

Для переривання потоку до його нормального завершення служить метод Thread.Abort(). Нижче наведена найпростіша форма цього методу.

public void Abort()

Метод Abort() створює необхідні умови для генерування винятку ThreadAbortException в тому потоці, для якого він був викликаний. Цей виняток призводить до переривання потоку і може бути перехоплено і в коді програми, але в цьому випадку він автоматично генерується ще раз, щоб зупинити потік. Метод Abort() не завжди здатний зупинити потік негайно, тому якщо потік потрібно зупинити перед тим, як продовжити виконання програми, то після методу Abort() слід відразу ж викликати метод Join(). Крім того, в найрідкісніших випадках методу Abort() взагалі не вдається зупинити потік. Це відбувається, наприклад, в тому випадку, якщо кодовий блок finally входить в нескінченний цикл [2].

Починаючи з .NET 5 метод Thread.Abort позначений як застарілий. Виклик цього методу попереджає компілятора. Під час виконання метод викликає виключення PlatformNotSupportedException.

Запит на передчасне переривання може бути перевизначений в самому потоці. Для цього необхідно спочатку перехопити в потоці виключення ThreadAbortException, а потім викликати метод ResetAbort(). Завдяки цьому виключається повторне генерування винятку по завершенні обробника виключення, який перериває цей потік. Нижче наведена форма оголошення методу ResetAbort().

public static void ResetAbort()

Виклик методу ResetAbort() може завершитися невдало, якщо в потоці відсутній належний режим надійного скасування передчасного переривання потоку.

1.1.3. Призупинення та відновлення потоку

У початкових версіях середовища .NET Framework потік можна було призупинити викликом методу Thread.Suspend() і відновити викликом методу Thread.Resume(). Але тепер, обидва ці методи вважаються застарілими і не рекомендуються до застосування. Пояснюється це зокрема тим, що користуватися методом Suspend() насправді небезпечно, так як з його допомогою можна призупинити потік, який в даний момент утримує блокування, що перешкоджає його зняттю, а отже, призводить до взаімоблокування. Застосування обох методів може стати причиною серйозних ускладнень на рівні системи. Тому для припинення та поновлення потоку слід використовувати інші засоби синхронізації, в тому числі м'ютекс і семафор.

1.1.4. Пріоритети потоків

У кожного потоку є свій пріоритет, який певною мірою визначає, наскільки часто потік отримує доступ до ЦП. Взагалі кажучи, фонові потоки отримують доступ до ЦП рідше, ніж високопріоритетні. Таким чином, протягом заданого проміжку часу фоновому потоку буде доступно менше часу ЦП, ніж високопріоритетним. Час ЦП, що отримується потоком, справляє визначальний вплив на характер його виконання і взаємодії з іншими потоками, виконуваними в даний момент в системі [2].

Слід мати на увазі, що, крім пріоритету, на частоту доступу потоку до ЦП впливають і інші фактори. Так, якщо високопріоритетний потік очікує доступу до деякого ресурсу, наприклад для введення з клавіатури, він блокується, а замість нього виконується фоновий потік. У подібній ситуації фоновий потік може отримувати доступ до ЦП частіше, ніж високопріоритетний потік протягом певного періоду часу. І нарешті, конкретне планування завдань на

рівні операційної системи також впливає на час ЦП, що виділяється для потоку.

Коли породжений потік починає виконуватися, він отримує пріоритет, який встановлюється за умовчанням. Пріоритет потоку можна змінити за допомогою властивості Priority, що є членом класу Thread. Нижче наведена загальна форма даного властивості:

public ThreadPriority Priority {get; set; }

де *ThreadPriority* позначає перерахування, в якому визначаються наведені нижче значення пріоритетів [3,9].

1.1.5. Визначення моменту закінчення потоку

В класі Thread є два способи для визначення моменту закінчення потоку. Доступна тільки для читання властивість IsAlive визначається наступним чином.

public bool IsAlive {get;}

Властивість IsAlive повертає логічне значення true, якщо потік, для якого вона викликається, як і раніше виконується.

Ще один спосіб відстеження моменту закінчення складається у виклику методу Join(). Нижче наведена його найпростіша форма.

public void Join()

Метод Join() очікує поки потік, для якого він був викликаний, не завершиться. Його ім'я відображає принцип очікування до того часу, поки потік, що викликає не приєднається до викликаного методу. Якщо ж даний

потік не було розпочато, то генерується виключення ThreadStateException. В інших формах методу Join() можна вказати максимальний період часу, протягом якого слід очікувати завершення зазначеного потоку [2,9].

1.1.6. Передача аргументу потоку

Аргумент передається потоку за допомогою методу Start().

public void Start (object параметр)

Об'єкт, що вказується в якості аргументу *параметр*, автоматично передається методу, що виконує роль точки входу в потік. Отже, для того щоб передати аргумент потоку, досить передати його методу Start().

Для застосування параметризованої форми методу Start() буде потрібно наступна форма конструктора класу Thread:

public Thread (ParameterizedThreadStart запуск)

де запуск позначає метод, що викликається з метою почати виконання потоку.

В даному випадку ParameterizedThreadStart є делегатом, який декларується в такий спосіб.

public delegate void ParameterizedThreadStart (object obj)

Делегат приймає аргумент типу object. Тому для правильного застосування даної форми конструктора класу Thread, у методі в якості точки входу в потік повинен бути параметр типу object [3,9].

1.1.7. Властивість IsBackground

В середовищі .NET визначені два різновиди потоків: **пріоритетний і фоновий**. Єдина відмінність між ними полягає в тому, що процес не завершиться до того часу, поки не закінчиться пріоритетний потік, тоді як фонові потоки завершуються автоматично після закінчення всіх пріоритетних потоків. За замовчуванням створюваний потік стає пріоритетним, але його можна зробити фоновим, використовуючи властивість IsBackground в класі Thread:

public bool IsBackground {get; set; }

Для того щоб зробити потік **фоновим**, досить привласнити логічне значення **true** властивості **IsBackground**. А логічне значення **false** вказує на те, що потік є **пріоритетним** [2].

1.2. Процеси

1.2.1. Визначення процесу

При програмуванні на C# найчастіше організовується багатозадачність на основі потоків. Але там, де це доречно, можна організувати і багатозадачність на основі процесів. У цьому випадку замість запуску іншого потоку в одній і тій же програмі, одна програма починає виконання іншої. Це робиться за допомогою **класу Process**, визначеного в просторі імен **System.Diagnostics**.

У Windows під **процесом** розуміється об'єкт ядра, якому належать системні ресурси, використовувані виконуваним додатком. Тому можна сказати, що в Windows **процесом є додаток, що виконується**. Виконання кожного процесу починається з первинного потоку. Під час свого виконання процес може створювати інші потоки. Виконання процесу закінчується при завершенні роботи всіх його потоків. Кожний процес в операційній системі Windows володіє наступними ресурсами:

- віртуальним адресним простором;
- робочою безліччю сторінок в реальній пам'яті;
- маркером доступу, що містить інформацію для системи безпеки;
- таблицею для зберігання дескрипторів об'єктів ядра.

Крім дескриптора, кожен процес в Windows має свій ідентифікатор, який є унікальним для процесів, що виконуються в системі. Ідентифікатори процесів використовуються, головним чином, службовими програмами, які дозволяють користувачам системи відстежувати роботу процесів [2,10].

Іноді процесу потрібно знати свій дескриптор, щоб змінити якісь свої характеристики. Наприклад, процес може змінити свій пріоритет.

Псевдодескриптор процесу відрізняється від справжнього дескриптора процесу тим, що він може використовуватися тільки поточним процесом і не може успадковуватися іншими процесами. Псевдодескриптор процесу не потрібно закривати після його використання. З псевдодескриптором процесу можна отримати справжній дескриптор процесу.

1.2.2. Запуск та завершення процесів

Найпростіший спосіб запустити інший процес можливо за допомогою метода Start(), визначеного у класі Process. Нижче наведена одна з найпростіших форм цього методу:

public static Process Start (string ім'я_файлу)

де *ім'я_файлу* позначає конкретне ім'я файлу, який повинен виконуватися або ж пов'язаний з виконуваним файлом.

Процес, який створює новий процес, називається батьківським процесом (parent process) по відношенню до створюваного процесу. Новий же процес,

який створюється іншим процесом, називається дочірнім процесом (child process) по відношенню до батьківського процесу [2].

Коли створений процес завершується, слід викликати метод Close(), щоб звільнити пам'ять, виділену для цього процесу. Нижче наведена форма оголошення методу Close().

public void Close()

1.3. Синхронізація

1.3.1. Визначення синхронізації

Синхронізація процесів - це приведення двох або декількох процесів до фіксованого порядку їх протікання, для уникнення конкуренції потоків або взаємного блокування у випадку роботи зі спільними ресурсами. У програмуванні розглядаються паралельні процеси, які є програмами, що виконуються процесором [2].

Усі потоки, що належать одному процесу, поділяють деякі загальні ресурси - такі, як адресний простір оперативної пам'яті або відкриті файли. Ці ресурси належать всьому процесу, а значить, і кожному його потоку. Отже, кожний потік може працювати з цими ресурсами без будь-яких обмежень. Але якщо один потік ще не закінчив працювати з будь-яким загальним ресурсом, а система переключилася на інший потік, який використовує цей же ресурс, то результат роботи цих потоків може надзвичайно сильно відрізнятися від задуманого. Такі конфлікти можуть виникнути і між потоками, що належать різним процесам. Завжди, коли два або більше потоків використовують будь-який загальний ресурс, виникає ця проблема [1].

Саме тому необхідний механізм, що дозволяє потокам погоджувати свою роботу з загальними ресурсами. Цей механізм отримав назву механізму **синхронізації потоків**.

Цей механізм являє собою набір об'єктів операційної системи, які створюються і управляються програмно, є загальними для всіх потоків в системі (деякі - для потоків, що належать одному процесу) і використовуються для координування доступу до ресурсів. Як ресурс може виступати все, що може бути загальним для двох і більше потоків: глобальна змінна програми (яка може бути доступна з потоків, що належать одному процесу), файл на диску, порт, запис в базі даних, об'єкт GDI.

Об'єктів синхронізації існує декілька, найважливіші з них:

- взаємовиключення (mutex),
- подія (event),
- семафор (semaphore),
- блокування (lock),
- монітор (monitor).

Кожен з цих об'єктів реалізує свій спосіб синхронізації. Також в якості об'єктів синхронізації можуть використовуватися самі процеси і потоки (коли один потік чекає завершення іншого потоку або процесу), а також файли, комунікаційні пристрої, консольне введення і повідомлення про зміну [2].

Будь-який об'єкт синхронізації може перебувати в так званому **сигнальному стані**. Для кожного типу об'єктів цей стан має різний зміст. Потоки можуть перевіряти поточний стан об'єкта та / або чекати зміни цього стану і таким чином узгоджувати свої дії. Гарантується, що коли потік працює з об'єктами синхронізації (створює їх, змінює стан) операційна система не перерве його виконання, поки він не завершить цю дію [11].

1.3.2. М'ютекс

М’ютекс (mutual exclusion) є взаємовиключним синхронізуючим об’єктом.

A mutex makes it possible to ensure that only one thread at a time has access to the object.

A monitor is an additional "superstructure" over a mutex.

A Mutex can be either named or unnamed. If a Mutex is named, it is eligible to be a system-wide Mutex that can be accessed from multiple processes. If a Mutex is unnamed, it is an anonymous Mutex which can only be accessed within the process in which it is created.

The name may be prefixed with **Global** or **Local** to specify a namespace. When the Global namespace is specified, the synchronization object may be shared with any processes on the system. When the Local namespace is specified, which is also the default when no namespace is specified, the synchronization object may be shared with processes in the same session. On Windows, a session is a login session, and services typically run in a different non-interactive session. Session-local synchronization objects may be appropriate for synchronizing between processes with a parent/child relationship where they all run in the same session.

```
var mutexName = "Global\\" + path.Replace("\\", ""); // mutex names don't like \

            // create a global mutex
            using (var mutex = new Mutex(false, mutexName))
            {
                Console.WriteLine("Waiting for mutex");
                var mutexAcquired = false;
                try
                {
                    // acquire the mutex (or timeout after 60 seconds)
                    // will return false if it timed out
                    mutexAcquired = mutex.WaitOne(60000);
                }
                catch(AbandonedMutexException)
                {
                    // abandoned mutexes are still acquired, we just need
                    // to handle the exception and treat it as acquisition
                    mutexAcquired = true;
                }


            // if it wasn't acquired, it timed out, so can handle that how ever we want
                if (!mutexAcquired)
                {
                    Console.WriteLine("I have timed out acquiring the mutex and can
handle that somehow");
                    return;
```

```
}

// otherwise, we've acquired the mutex and should do what we need to do,
// then ensure that we always release the mutex
try
{
    DoWork(path);
}
finally
{
    mutex.ReleaseMutex();
}
```

Це означає, що він може бути отриманий потоками тільки по черзі. М'ютекс призначений для тих ситуацій, в яких загальний ресурс може бути одночасно використаний **тільки в одному потоці** (рис. 2). Припустимо, що системний журнал спільно використовується в декількох процесах, але тільки в одному з них дані можуть записуватися в файл цього журналу в будь-який момент часу. Для синхронізації процесів в даній ситуації ідеально підходить м’ютекс [2].

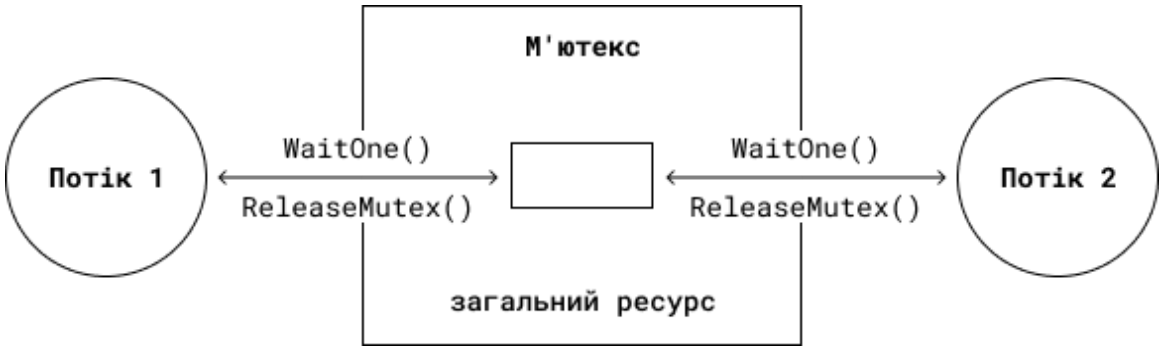


Рис. 2. М'ютекс

М'ютекс підтримується в класі System.Threading.Mutex. У нього є кілька конструкторів. Нижче наведені два найбільш уживаних конструктора.

public Mutex()

public Mutex (bool initiallyOwned)

У першій формі конструктора створюється м'ютекс, яким спочатку ніхто не володіє. А в другій формі вихідним станом мьютекса заволодіває потік, що викликає, якщо параметр initiallyOwned має логічне значення true. В іншому випадку м'ютексом ніхто не володіє.

Для того щоб отримати м'ютекс, в коді програми слід викликати метод WaitOne() для цього мьютекса. Метод WaitOne() успадковується класом Mutex від класу Thread.WaitHandle. Нижче наведена його найпростіша форма.

public bool WaitOne();

Метод WaitOne() очікує до того часу, поки не буде отримано м'ютекс, для якого він був викликаний. Отже, цей метод блокує виконання потоку, що викликає до того часу, поки не стане доступним вказаний м'ютекс. Він завжди повертає логічне значення true.

Коли ж в коді більше не потрібно володіти м'ютексом, він звільняється за допомогою виклику методу ReleaseMutex(), форма якого наведена нижче.

public void ReleaseMutex()

У цій формі метод ReleaseMutex() звільняє мьютекс, для якого він був викликаний, що дає можливість іншому потоку отримати даний м'ютекс.

Для застосування мьютекса з метою синхронізувати доступ до загального ресурсу згадані вище методи WaitOne() і ReleaseMutex() використовуються так, як показано в наведеному нижче фрагменті коду.

```
Mutex myMtx = new Mutex();
// ...
myMtx.WaitOne();          // очікувати отримання мьютекса
// Отримати доступ до загального ресурсу.
```

myMtx.ReleaseMutex(); // звільнити м'ютекс

Викликом методу WaitOne() виконання відповідного потоку призупиняється до того часу, поки не буде отримано м'ютекс. А при виклику методу ReleaseMutex() м'ютекс звільняється, й потім може бути отриманий іншим потоком. Завдяки такому підходу до синхронізації одночасний доступ до загального ресурсу обмежується тільки одним потоком [3,11].

1.3.3. Семафор

Its distinctive feature is that it uses a counter to create the synchronization mechanism.

The counter tells us how many threads can simultaneously access the shared resource.

Семафор подібний м'ютексу, за винятком того, що він надає одночасний доступ до загального ресурсу не одному, а **декільком потокам** (рис. 3). Тому семафор придатний для синхронізації цілого ряду ресурсів.

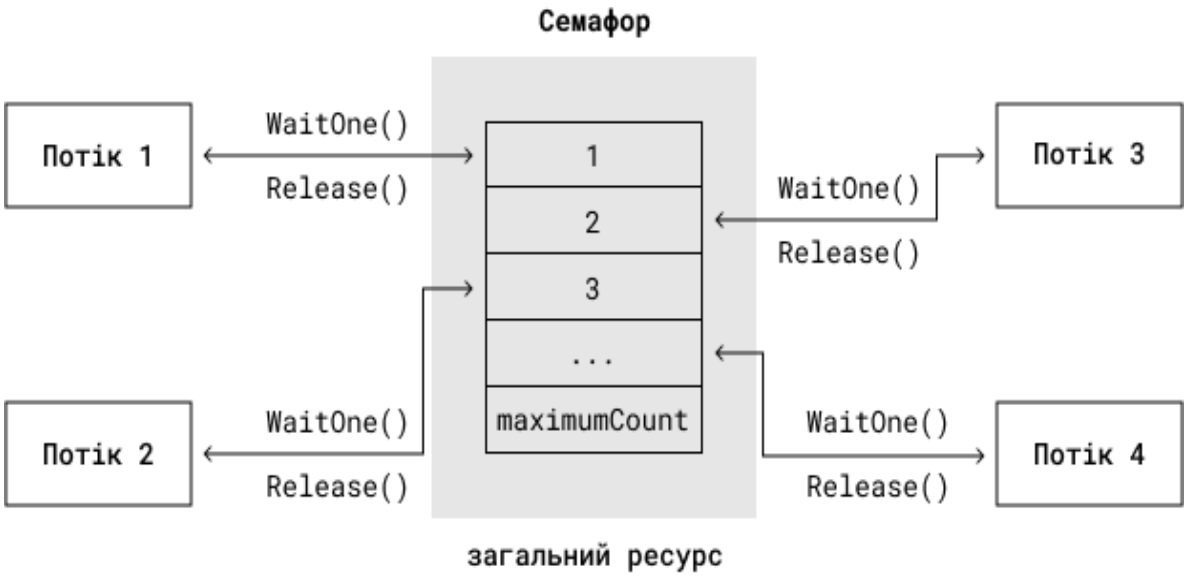


Рис. 3. Семафор

Семафор управляє доступом до загального ресурсу, використовуючи для цієї мети лічильник. Якщо значення лічильника більше нуля, то доступ до ресурсу дозволений. А якщо це значення дорівнює нулю, то доступ до ресурсу заборонено. За допомогою лічильника ведеться підрахунок кількості дозволів. Отже, для доступу до ресурсу потік повинен отримати дозвіл від семафора [2].

Зазвичай потік, якому потрібен доступ до загального ресурсу, намагається отримати дозвіл від семафора. Якщо значення лічильника семафора більше нуля, то потік отримує дозвіл, а лічильник семафора декрементується. В іншому випадку потік блокується до того часу, поки не отримає дозвіл. Коли ж потоку більше не потрібен доступ до загального ресурсу, він вивільняє дозвіл, а лічильник семафора інкрементирується. Якщо дозволу чекає інший потік, то він отримує його в цей момент. Кількість одночасно дозволених доступів вказується при створенні семафора. Так, якщо створити семафор, який одночасно дозволяє тільки один доступ, то такий семафор буде діяти як м'ютекс.

Семафори особливо корисні в тих випадках, коли загальний ресурс складається з групи або пулу ресурсів. Наприклад, пул ресурсів може складатися з цілого ряду мережевих з'єднань, кожне з яких служить для передачі даних. Потоку, якому потрібно мережеве з'єднання, все одно, яке саме з'єднання він отримає. В даному випадку семафор забезпечує зручний механізм управління доступом до мережевих з'єднань [2,11].

Семафор реалізується в класі System.Threading.Semaphore, у якого є декілька конструкторів. Нижче наведена найпростіша форма конструктора даного класу.

public Semaphore(int initialCount, int maximumCount)

де *initialCount* - це початкове значення для лічильника дозволів семафора, тобто кількість доступних дозволів;

maximumCount - максимальне значення даного лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор застосовується таким же чином, як і м'ютекс. З метою отримання доступу до ресурсу в коді програми викликається метод WaitOne() для семафора. Цей метод успадковується класом Semaphore від класу WaitHandle. Метод WaitOne() очікує до того часу, поки не буде отримано семафор, для якого він викликається. Таким чином, він блокує виконання потоку, що викликає до того часу, поки вказаний семафор не надасть дозвіл на доступ до ресурсу [1].

Якщо коду більше не потрібно володіти семафором, він звільняє його, викликаючи метод Release(). Нижче наведено дві форми цього методу.

public int Release()
public int Release(int releaseCount)

У першій формі метод Release() вивільняє тільки один дозвіл, а в другій формі - кількість дозволів, які визначаються параметром releaseCount. В обох формах даний метод повертає підраховану кількість дозволів, які існували до вивільнення.

В .NET пропонується два класи з функціональністю семафора:

Semaphore,
SemaphoreSlim.

Клас Semaphore може бути іменований, використовувати ресурси в масштабі всієї системи і забезпечувати синхронізацію між різними процесами. Клас SemaphoreSlim є полегшеною версією класу Semaphore, яка оптимізована для забезпечення більш короткого часу очікування [11].

1.3.4. Події

Події є ще одним ресурсом для забезпечення синхронізації в масштабі всієї системи.

Для використання системних подій з керованого коду, .NET пропонує класи ManualResetEvent, AutoResetEvent, ManualResetEventSlim і CountdownEvent, які знаходяться в просторі імен System.Threading [2,7,11].

Ці класи є похідними від класу EventWaitHandle, що знаходиться на верхньому рівні ієрархії класів, і застосовуються в тих випадках, коли **один потік чекає появи деякої події в іншому потоці.** Як тільки така подія з'являється, другий потік повідомляє про нього перший потік, дозволяючи тим самим відновити його виконання (рис. 4).

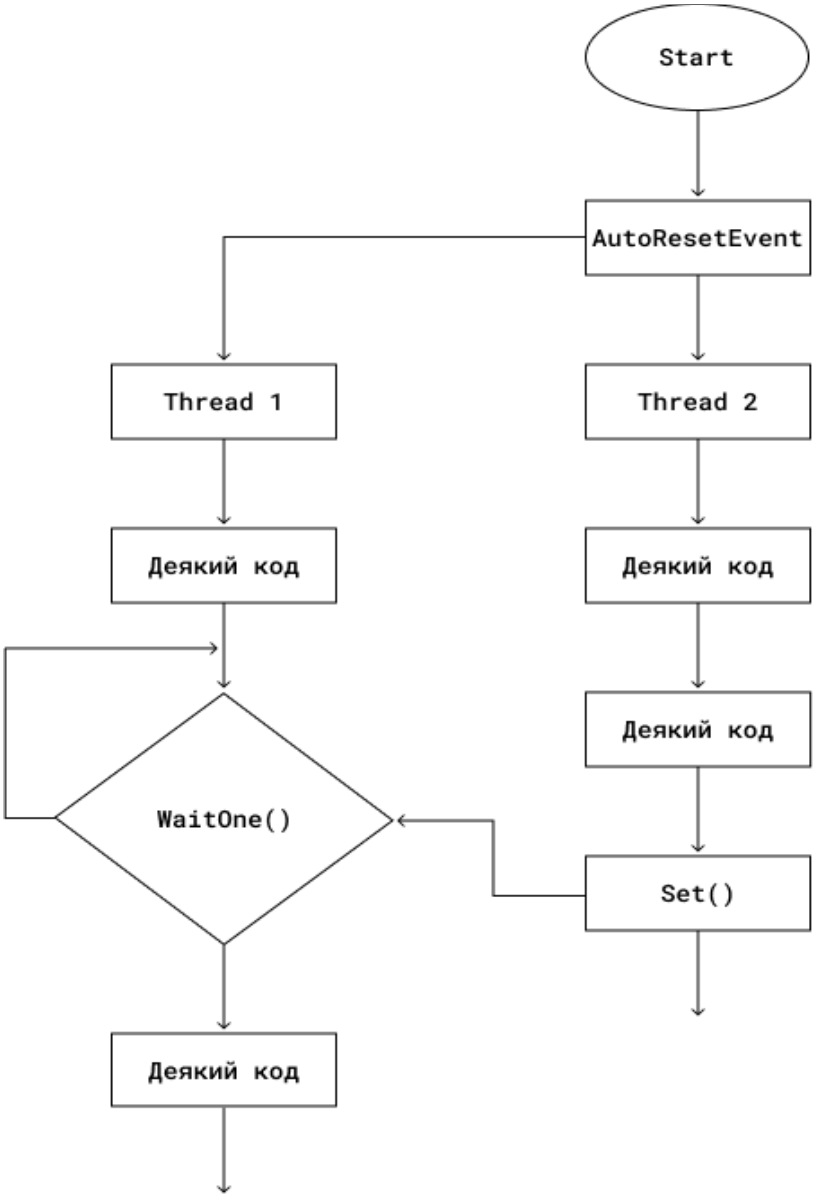


Рис. 4. Події

Нижче наведені конструктори класів ManualResetEvent і AutoResetEvent.

public ManualResetEvent(bool initialState)
public AutoResetEvent(bool initialState)

Якщо в обох формах параметр initialState має логічне значення **true**, то **про подію спочатку повідомляється**. А якщо він має логічне значення false, то про подію спочатку не буде повідомлено.

Для події типу ManualResetEvent порядок застосування наступний. Потік, що очікує деяку подію, викликає метод WaitOne() для подієвого об'єкта, що представляє дану подію. Якщо подієвий об'єкт знаходиться в сигнальному стані, то відбувається негайне повернення з методу WaitOne(). В іншому випадку виконання потоку, що викликає призупиняється до того часу, поки не буде отримано повідомлення про подію. Як тільки подія відбудеться в іншому потоці, цей потік **встановить подієвий об'єкт в сигнальний стан**, викликавши метод **Set()**. Тому метод Set() слід розглядати як повідомляє про те, що **подія відбулася**.

Після установки подієвого об'єкта в сигнальний стан станеться негайне повернення з методу WaitOne(), і перший потік відновить своє виконання. А в результаті виклику методу **Reset()** подієвий об'єкт повертається **в несигнальний стан.**

Подія AutoResetEvent відрізняється від події типу ManualResetEvent лише способом установки в початковий стан. Якщо, для події типу ManualResetEvent подієвий об'єкт залишається в сигнальному стані доти, поки не буде викликаний метод Reset(), то для події типу AutoResetEvent подієвий об'єкт автоматично переходить в несигнальному стан, як тільки потік, який чекає цю подію, отримає повідомлення про нього і відновить своє виконання.

Тому якщо застосовується подія типу AutoResetEvent, то викликати метод Reset() необов'язково [2-4].

Подія ManualResetEventSlim перекладається в сигнальний стан викликом методу Set(), а за допомогою Reset() повертається назад в несигнальному стан. У разі виклику методу Set() при наявності безлічі потоків, які чекають переходу події в сигнальний стан, очікування всіх цих потоків негайно припиняється. У разі, якщо потік просто викликає метод WaitOne(), а подія вже знаходиться в сигнальному стані, потік, який чекав, може відразу ж продовжити роботу [11].

1.3.5. Клас Monitor і блокування

Ключове слово lock служить в C# швидким способом доступу до засобу синхронізації, визначеним у класі Monitor, який знаходиться в просторі імен System.Threading. В цьому класі визначено, зокрема, ряд методів для управління синхронізацією. Наприклад, для отримання **блокування об'єкта** викликається метод **Enter(),** а для **зняття блокування** - метод **Exit().** Нижче наведені загальні форми цих методів.

public static void Enter (object obj)
public static void Exit (object obj)

де obj позначає синхронізований об'єкт. Якщо ж об'єкт недоступний, то після виклику методу Enter() потік, що викликає, очікує до того часу, поки об'єкт не стане доступним. Проте методи Enter() і Exit() застосовуються рідко, оскільки оператор lock автоматично надає еквівалентні засоби синхронізації потоків. Саме тому оператор lock виявляється "кращим" для отримання блокування об'єкта при програмуванні на C# [2].

У класі Monitor визначено метод TryEnter(), одна із загальних форм якого наведена нижче.

public static bool TryEnter (object obj)

Метод повертає логічне значення true, якщо потік, що викликає, отримує блокування для об'єкта obj, а інакше він повертає логічне значення false.

Але в будь-якому випадку потоку, що викликає, доведеться чекати своєї черги. За допомогою цього методу можна реалізувати альтернативний варіант синхронізації потоків, якщо необхідний об'єкт тимчасово недоступний [11].

1.3.6. Методи Wait(), Pulse() і PulseAll()

Методи Wait(), Pulse() і PulseAll() визначені в класі Monitor і можуть викликатися тільки з заблокованого фрагмента блоку. Вони застосовуються таким чином.

Коли виконання потоку тимчасово заблоковано, він викликає метод Wait(). В результаті потік переходить в стан очікування, а блокування з відповідного об'єкта знімається, що дає можливість використовувати цей об'єкт в іншому потоці. Надалі потік, що очікує, активізується, коли інший потік увійде в аналогічний стан блокування, і викликає метод Pulse() або PulseAll(). При виклику методу **Pulse()** поновлюється **виконання першого потоку**, що очікує своєї черзі на отримання блокування. А виклик методу **PulseAll()** сигналізує про **зняття блокування всім потокам**, що очікують [2].

Нижче наведено дві найбільш використовувані форми методу Wait().

public static bool Wait (object obj)
public static bool Wait (object obj, int міллісекунд_простою)

У першій формі очікування триває аж до повідомлення про звільнення об'єкта, а в другій формі - як до повідомлення про звільнення об'єкта, так і до закінчення періоду часу, на який вказує кількість міллісекунд_простою. В обох формах obj позначає об'єкт, звільнення якого очікується.

Нижче наведені загальні форми методів Pulse() і PulseAll().

public static void Pulse (object obj)
public static void PulseAll (object obj)

де obj позначає звільняється об'єкт.

Якщо методи Wait(), Pulse() і PulseAll() викликаються з коду, що знаходиться за межами синхронізованого коду, наприклад з блоку lock, то генерується виняток SynchronizationLockException [3,11].

1.3.7. Атрибут MethodImplAttribute

Метод може бути повністю синхронізований з допомогою атрибута MethodImplAttribute. Такий підхід може стати альтернативою оператору lock в тих випадках, коли метод потрібно заблокувати повністю. Атрибут MethodImplAttribute визначено в просторі імен System.Runtime. CompilerServices. Нижче наведено конструктор, застосовуваний для подібної синхронізації.

public MethodImplAttribute (MethodImplOptions methodImplOptions)

де *methodImplOptions* позначає атрибут реалізації. Для синхронізації методу досить вказати атрибут MethodImplOptions.Synchronized. Цей атрибут викликає блокування всього методу для поточного екземпляра об'єкта, доступного за посиланням this. Якщо ж метод відноситься до типу static, то

блокується його тип. Тому даний атрибут непридатний для застосування у відкритих об'єктах або класах [11].

1.3.8. Клас Interlocked

Цей клас є в якості альтернативи іншим засобам синхронізації, коли потрібно тільки змінити значення загальної змінної. Методи, доступні в класі Interlocked, гарантують, що їх дія буде виконуватися як єдина, неперервна операція. Це означає, що ніякої синхронізації в даному випадку взагалі не вимагається. У класі Interlocked надаються статичні методи для складання двох цілих значень, инкрементирования і декрементірованія цілого значення, порівняння і установки значень об'єкта, обміну об'єктами і отримання 64-разрядного значення. Всі ці операції виконуються без переривання [3].

1.4. Відображення файлу у пам’яті

1.4.1. Концепція механізму відображення файлу у пам’яті

Механізм, що дозволяє динамічно виконувати підключення бібліотек називається відображенням вмісту файлу (file mapping) у віртуальну пам'ять процесу.

ОС створює файли підкачки з віртуальними сторінками, які система відображає в адресні простори процесів.

В ОС Windows реалізований механізм, який дозволяє відображати в адресний простір процесу не тільки вміст файлів підкачки, але і вміст звичайних файлів. Тобто в цьому випадку файл або його частину розглядається як набір віртуальних сторінок процесу, які мають послідовні логічні адреси (рис. 5).

Файл, відображений в адресний простір процесу, називається поданням або видом файлу (file view). Після відображення файлу в адресний простір

процесу доступ до виду може здійснюватися за допомогою вказівника, як до звичайних даних в адресному просторі процесу.

Кілька процесів можуть одночасно відображати один і той же файл в свій адресний простір. В цьому випадку ОС забезпечує узгодженість вмісту файлу для всіх процесів, якщо доступ до цих даних здійснюється як до області віртуальної пам'яті процесу. Тобто для доступу до файлу, який відображений в пам'ять, не використовується функція WriteFile.

Така узгодженість даних, що зберігаються в файлі, відображеному в пам'ять декількома процесами, називається когерентністю даних. Когерентність даних для файлу, відображеного в пам'ять, не підтримується в тому випадку, якщо цей файл відображається в адресний простір процесів, які виконуються на інших комп'ютерах в локальній мережі.

Основним призначенням механізму відображення файлів в пам'ять є завантаження програми (файлу з розширенням .exe) на виконання, в адресному просторі процесу, і динамічне підключення бібліотек функцій під час виконання цієї програми.

Крім того, механізм відображення файлів в пам'ять дозволяє здійснювати обмін даними між процесами, беручи до уваги те, що система забезпечує когерентність даних в файлі, який відображається в пам'ять.

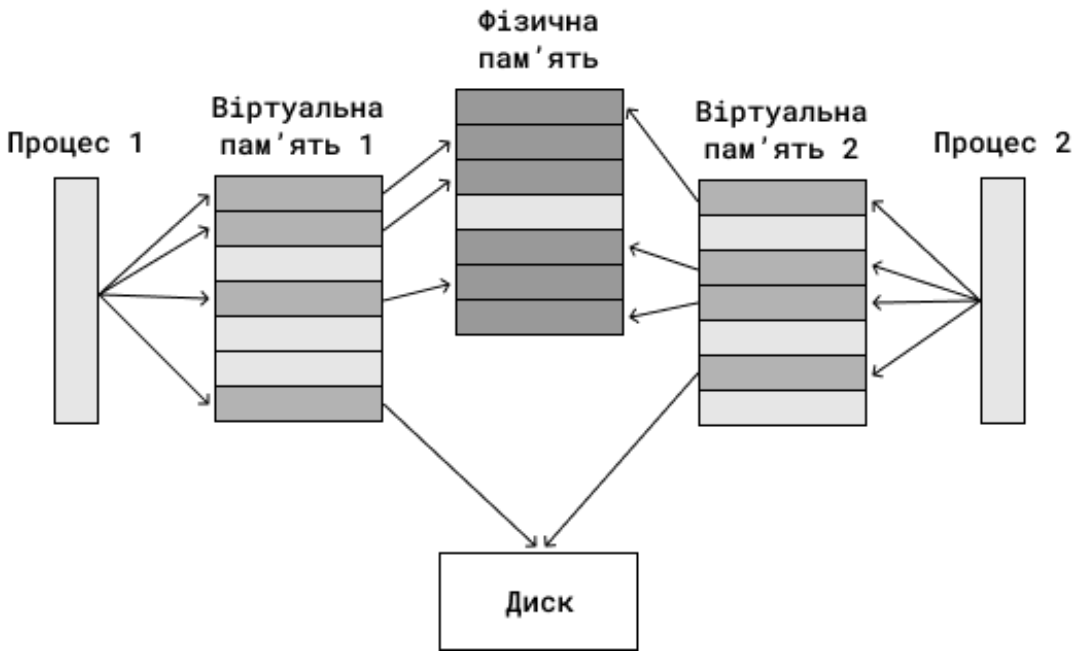


Рис. 5. Відображення файлу в пам'ять

Відображення файлу в пам'ять представляє вміст файлу у віртуальній пам'яті. Це зіставлення файлу і області пам'яті дозволяє додатку, включаючи безліч процесів, змінювати файл шляхом читання і запису прямо в пам'ять [12].

1.4.2. Класи простору імен System.IO.MemoryMappedFiles

Простір імен System.IO.MemoryMappedFiles надає класи для використання зіставленого у пам’яті файлу, який відображає вміст файлу у логічному адресному просторі програми.

До класів цього простору імен відносяться:

- MemoryMappedFile;
- MemoryMappedFileSecurity;
- MemoryMappedViewAccessor;
- MemoryMappedViewStream.

Клас **MemoryMappedFile** представляє зіставлений в пам’яті файл.

public class MemoryMappedFile : IDisposable

Файл, зіставлений у пам'яті, відображає вміст файлу у логічному адресному просторі програми. Файли, зіставлені у пам’яті, дозволяють програмістам працювати з надзвичайно великими файлами, оскільки пам'ять може бути використана паралельно, і вони дозволяють повний, довільний доступ до файлу без необхідності пошуку. Зіставлені у пам’яті файли також можуть бути доступними для різних процесів.

Файл, зіставлений у пам'яті, пов'язаний з ім'ям.

Можливо створювати декілька представлень зіставленого в пам’яті файлу, включаючи представлення частин файлу. Можливо зіставити одну й ту

саму частину файлу з більш ніж однією адресою, аби створити паралельну пам'ять. Щоб два представлення залишалися паралельними, вони повинні бути створені з одного зіставленого в пам’яті файлу. Створення двох файлів відображення одного і того самого файлу з двома представленнями не забезпечує паралельності.

Клас **MemoryMappedFileSecurity** представляє дозволи, які можуть бути надані для доступу до файлів і операцій із зіставленими у пам’яті файлами.

public class MemoryMappedFileSecurity :
System.Security.AccessControl.ObjectSecurity
<System.IO.MemoryMappedFiles. MemoryMappedFileRights>

Клас MemoryMappedFileSecurity визначає права доступу до файлу, відображеного у пам'яті, і як перевіряються спроби доступу. Цей клас представляє права доступу та аудиту як сукупність правил.

Клас MemoryMappedFileSecurity є абстракцією базової системи безпеки файлів Microsoft Windows. У цій системі вона приховує багато деталей DACL та SACLs. Цей клас використовується для завантаження, додавання або зміни правил доступу, що відображають DACL та SACL каналу.

Щоб застосувати нові або змінені правила доступу або правила аудиту до зіставленого у пам’яті файлу, потрібно скористатися методом SetAccessControl. Метод GetAccessControl надає можливість отримати доступ або правила перевірки з існуючого файлу.

Клас **MemoryMappedViewAccessor** представляє собою представлення зіставленого в пам’яті файлу з **довільним доступом**.

public sealed class MemoryMappedViewAccessor :
System.IO.UnmanagedMemoryAccessor

Для отримання представлення використовується метод CreateViewAccessor об’єкту класу MemoryMappedFile.

Клас **MemoryMappedViewStream** представляє собою представлення зіставленого в пам’яті файлу як потік з **послідовним доступом**.

public sealed class MemoryMappedViewStream :
System.IO.UnmanagedMemoryStream

Для отримання цього потоку використовується метод CreateViewStream класу MemoryMappedFile [12].

1.4.3. Постійні зіставлені в пам’яті файли. Метод CreateFromFile()

Постійні файли є зіставленими в пам'яті файлами, які **пов'язані з вихідним файлом на диску**. Дані зберігаються в вихідний файл на диск, коли останній процес закінчив роботу з файлом. Ці зіставлені в пам'яті файли застосовуються для роботи з дуже великими файлами.

Метод CreateFromFile створює зіставлений у пам'яті файл за визначеним шляхом або за файловим потоком (клас FileStream) існуючого файлу на диску. CreateFromFile(String) створює зіставлений в пам’яті файл, з файлу на диску.

public static System.IO.MemoryMappedFiles.MemoryMappedFile
CreateFromFile (string path);

Параметром передається шлях до існуючого файлу на диску.

Перевантаження методу:

- CreateFromFile(String, FileMode) - створює зіставлений в пам’яті файл, що має вказаний режим доступу, з файлу на диску.

- CreateFromFile(String, FileMode, String) - створює зіставлений в пам’яті файл, що має вказаний режим доступу і ім’я, з файлу на диску.
- CreateFromFile(String, FileMode, String, Int64) - створює зіставлений в пам’яті файл, що має вказаний режим доступу, ім’я і ємність, з файлу на диску.
- CreateFromFile(String, FileMode, String, Int64, MemoryMappedFileAccess) - створює зіставлений в пам’яті файл, що має вказаний режим доступу, ім’я, ємність і тип доступу, з файлу на диску [12].

1.4.4. Непостійні зіставлені в пам’яті файли. Метод CreateNew()

Непостійні файли є зіставленими в пам'яті файлами, які не пов'язані з файлом на диску (рис. 6). Коли останній процес закінчив роботу з файлом, дані втрачаються і файл видаляється функцією збору сміття. Такі файли підходять для створення загальної пам'яті для взаємодії між процесами (IPC).

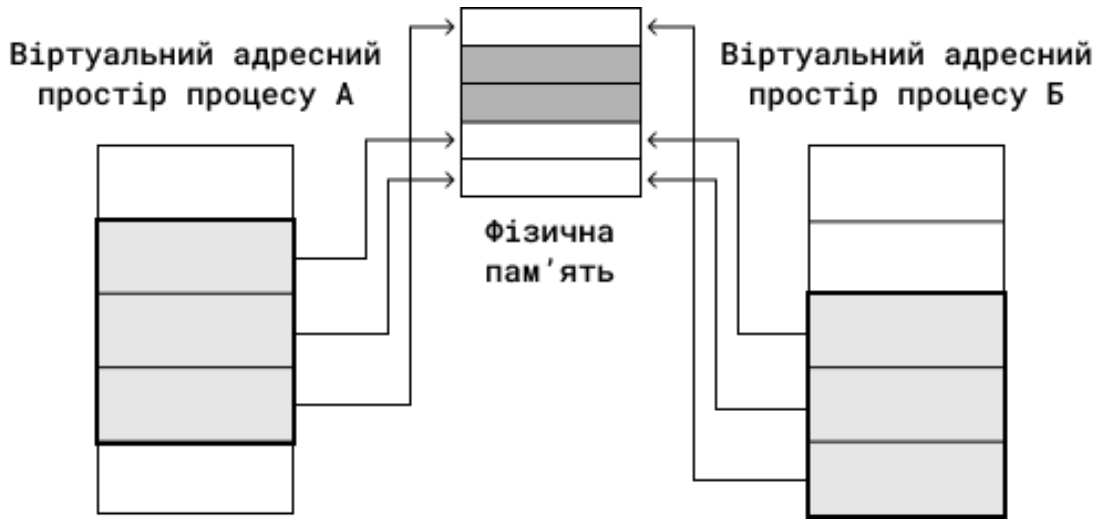


Рис. 6. Непостійні зіставлені в пам’яті файли

CreateNew(String, Int64) - створює зіставлений в пам’яті файл, що має вказану ємність в системній пам’яті.

public static System.IO.MemoryMappedFiles.MemoryMappedFile
CreateNew (string mapName, long capacity);

mapName – ім’я, що призначається для файлу, зіставленого в пам'яті;

capacity – максимальний розмір у байтах, який буде виділено для зіставленого у пам’яті файлу.

Перевантаження методу:

- CreateNew(String, Int64, MemoryMappedFileAccess) - створює зіставлений в пам’яті файл, що має вказану ємність і режим доступу в системній пам’яті;
- CreateNew(String, Int64, MemoryMappedFileAccess, MemoryMappedFileOptions, HandleInheritability) - створює зіставлений в пам’яті файл, що має вказане ім’я, ємність, режим доступу, опції виділення пам’яті і можливість до успадковування (значення яке визначає, чи може дескриптор на зіставлений в пам’яті файл бути успадкований дочірнім процесом);
- CreateNew(String, Int64, MemoryMappedFileAccess, MemoryMappedFileOptions, MemoryMappedFileSecurity, HandleInheritability) - створює зіставлений в пам’яті файл, що має вказане ім’я, ємність, режим доступу, опції виділення пам’яті, права безпеки і можливість до успадковування [12].

1.4.5. Метод OpenExisting()

Метод OpenExisting використовується для того, щоб отримати об’єкт MemoryMappedFile вже створеного зіставленого в пам’яті файлу (постійного чи непостійного).

OpenExisting(String) - відкриває існуючий зіставлений в пам’яті файл, що містить вказане ім'я в системній пам'яті.

public static System.IO.MemoryMappedFiles.MemoryMappedFile OpenExisting (string mapName);

mapName – ім’я зіставленого в пам’яті файлу, що необхідного відкрити.

Перевантаження методу:

- OpenExisting(String, MemoryMappedFileRights) - відкриває існуючий зіставлений в пам’яті файл, що має вказане ім'я і права доступу в системній пам'яті.
- OpenExisting(String, MemoryMappedFileRights, HandleInheritability) - відкриває існуючий зіставлений в пам’яті файл, що має вказане ім'я, права доступу і можливість до успадковування.

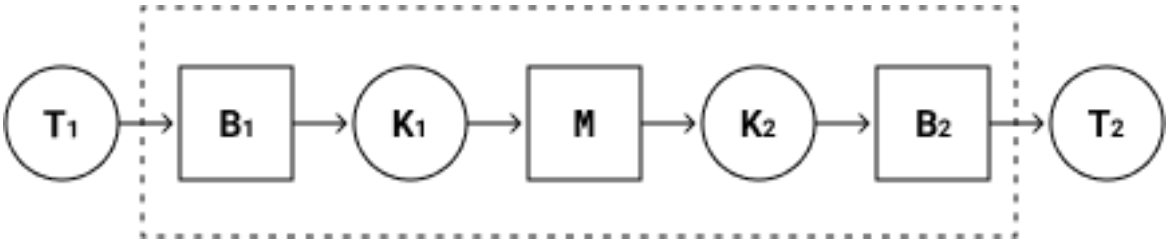
1.5. Канали передачі даних

1.5.1. Способи передачі даних між процесами

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

Якщо потоки виконуються в одному процесі, то для обміну даними між ними можна використовувати глобальні змінні і засоби синхронізації потоків. Насправді все складніше в тому випадку, якщо потоки виконуються в різних процесах - потоки не можуть звертатися до загальних змінних і для обміну даними між ними існують спеціальні засоби операційної системи.

Для обміну даними між процесами створюється канал передачі даних, організація якого схематично показана на рис. 7.



T1, T2 - користувацькі потоки,

B1, B2 – буфери,

K1, K2 - потоки ядра операційної системи,

M - загальна пам'ять.

Рис. 7. Схема каналу передачі даних

Канал даних включає вхідний і вихідний буфери пам'яті, потоки ядра операційної системи і загальну пам'ять, доступ до якої мають обидва потоки ядра. Працює канал передачі даних наступним чином:

- перший потік ядра ОС читає дані з вхідного буфера B1 і записує їх в загальну пам'ять M;
- другий потік ядра читає дані із загальної пам'яті M і записує їх в буфер B2.

Користувацькі потоки T1 і T2 за допомогою виклику функцій ядра операційної системи мають доступ до буферів B1 і B2 відповідно. Тому пересилання даних з потоку T1 в потік T2 відбувається наступним чином:

- користувацький потік T1 записує дані в буфер B1, використовуючи спеціальну функцію ядра операційної системи;
- потік K1 ядра операційної системи читає дані з буфера B1 і записує їх в загальну пам'ять M;

- потік К2 ядра операційної системи читає дані із загальної пам'яті М і записує їх в буфер В2.
- користувацький потік Т2 читає дані з буфера В2.

Обмін даними може бути організований тільки через ланцюжок взаємодіючих потоків, які обмінюються між собою даними через загальну, тільки для них, пам'ять.

Таким ж чином можлива організація каналу передачі даних по мережі. Тільки в цьому випадку загальна пам'ять М, може розглядатися як середовище, що передає, влаштування якого аналогічно влаштуванню каналу передачі даних [1].

1.5.2. Зв’язки між процесами

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

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

Топологією зв'язку є конфігурація зв'язків між процесами-відправниками та адресатами.

З точки зору топології розрізняють наступні види зв'язків:

- 1 —> 1 — між собою пов’язані тільки два процеси;
- 1 —> N — один процес пов’язаний с N процесами;
- N —> 1 — кожен з N процесів пов’язаний з одним процесом;
- N —> M — кожен з N процесів пов’язаний з кожним з M процесів.

Види зв'язків схематично показані на рис. 8, де процеси зображені колами, зв'язки – дугами, а прямокутники позначають засоби передачі даних.

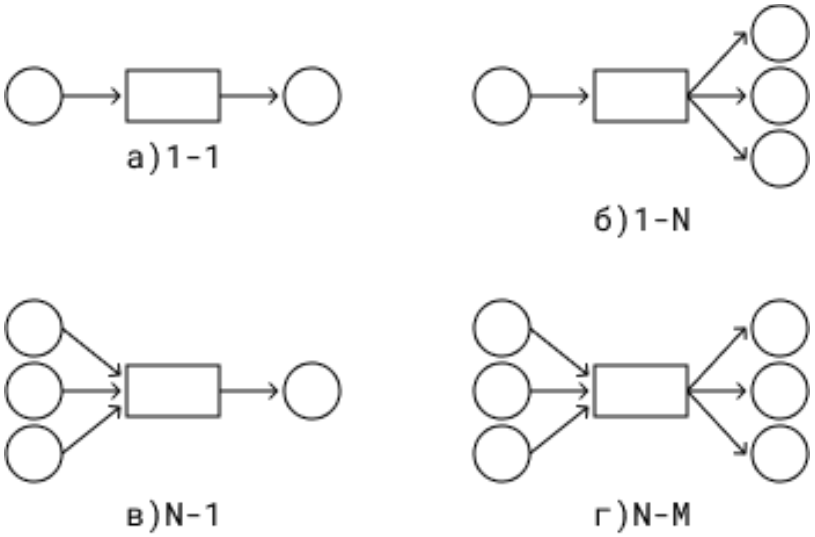


Рис. 8. Топології зв’язків між процесами

При розробці систем з обміном даними між процесами, перш за все, повинна бути обрана топологія зв'язків і напрямки передачі даних по цих зв`язках. Після цього в програмах реалізуються обрані зв'язки між процесами з використанням спеціальних функцій операційної системи, які призначені для встановлення зв'язку між процесами. Ці функції, а також функції для обміну даними між процесами забезпечує система передачі даних, яка зазвичай є частиною ядра операційної системи [1].

1.5.3. Анонімні канали

Анонімні канали забезпечують **міжпроцесну взаємодію на локальному комп'ютері**. Анонімні канали використовують менше системних ресурсів, ніж іменовані канали, але їх можливості обмежені. Анонімні канали є **неіменованими односторонніми** і не можуть використовуватися для взаємодії в мережі. Вони дозволяють використовувати **тільки один екземпляр сервера**.

Анонімні канали зручно використовувати для організації взаємодії **між потоками** або **між батьківськими і дочірніми процесами** - в цьому випадку

дескриптор каналу можна просто передати дочірньому процесу при його створенні.

В .NET анонімні канали реалізуються за допомогою класів AnonymousPipeServerStream і AnonymousPipeClientStream.

public sealed class AnonymousPipeServerStream:System.IO.Pipes.PipeStream
public sealed class AnonymousPipeClientStream:System.IO.Pipes.PipeStream

Анонімні канали допомагають організувати безпечний та захищений зв'язок між дочірнім та батьківським процесами. Клас AnonymousPipeServerStream дозволяє батьковіму процесові надсилати або отримувати інформацію з дочірнього процесу.

Значення InOut для PipeDirection не підтримується, бо анонімні канали визначені лише як однонаправлені. Анонімні канали не підтримують режими читання повідомлень.

Клієнтська сторона анонімного каналу має бути створена за допомогою дескриптора каналу, який надається серверною стороною каналу. Дескриптор отримується за допомогою методу GetClientHandleAsString. Потім рядок передається як параметр при створенні клієнтського процесу. З клієнтського процесу він потім передається конструктору AnonymousPipeClientStream як параметр pipeHandleAsString.

Об'єкт класу AnonymousPipeServerStream повинен видалити копію дескриптора клієнта, використовуючи метод DisposeLocalCopyOfClientHandle, для того, аби бути сповіщенним про існування клієнта.

Конструктори класу AnonymousPipeServerStream:

- AnonymousPipeServerStream() – ініціалізує новий екземпляр класу AnonymousPipeServerStream.

- AnonymousPipeServerStream(PipeDirection) – ініціалізує новий екземпляр класу AnonymousPipeServerStream з вказанням напрямку каналу.
- AnonymousPipeServerStream(PipeDirection, HandleInheritability) – ініціалізує новий екземпляр класу AnonymousPipeServerStream з вказанням напрямку каналу та режиму наслідування.
- AnonymousPipeServerStream(PipeDirection, HandleInheritability, Int32) – ініціалізує новий екземпляр класу AnonymousPipeServerStream з вказанням напрямку каналу, режиму наслідування та розміру буферу.
- AnonymousPipeServerStream(PipeDirection, HandleInheritability, Int32, PipeSecurity) – ініціалізує новий екземпляр класу AnonymousPipeServerStream з вказанням напрямку каналу, режиму наслідування, розміру буферу та безпеки каналу.
- AnonymousPipeServerStream(PipeDirection, SafePipeHandle, SafePipeHandle) – ініціалізує новий екземпляр класу AnonymousPipeServerStream з визначених дескрипторів каналу.

Конструктори AnonymousPipeClientStream:

- AnonymousPipeClientStream(String) - ініціалізує новий екземпляр класу AnonymousPipeClientStream з вказанням рядкового представленням дескриптору каналу.
- AnonymousPipeClientStream(PipeDirection, String) - ініціалізує новий екземпляр класу AnonymousPipeClientStream з вказанням напрямку каналу та рядкового представлення дескриптору каналу.
- AnonymousPipeClientStream(PipeDirection, SafePipeHandle) - ініціалізує новий екземпляр класу AnonymousPipeClientStream з визначених дескрипторів каналу [13].

1.5.4. Іменовані канали

Іменовані канали забезпечують міжпроцесну взаємодію **між сервером і одним або декількома клієнтами.** Іменовані канали можуть бути **односторонніми** або **двосторонніми**. Вони підтримують зв'язок за допомогою **повідомлень** і дозволяють **кільком клієнтам підключатися** до серверного процесу **одночасно**, використовуючи одне і те ж **ім'я каналу**. Іменовані канали також підтримують персональні налаштування, що дозволяють процесам що підключаються використовувати власні дозволи на віддалених серверах.

В .NET іменовані канали реалізуються за допомогою класів NamedPipeServerStream і NamedPipeClientStream.

```
public sealed class NamedPipeServerStream : System.IO.Pipes.PipeStream
public sealed class NamedPipeClientStream : System.IO.Pipes.PipeStream
```

Іменовані канали забезпечують односторонній або дуплексний канал зв'язку між сервером та одним або кількома клієнтами. Іменовані канали можуть бути використані для міжпроцесного зв'язку локально або **через мережу.** Одне ім'я каналу може використовуватися декількома об'єктами NamedPipeClientStream.

Будь-який процес може діяти і як сервер іменованого каналу, чи як клієнт, чи як і сервер, і клієнт одночасно.

Конструктори класу *NamedPipeServerStream*:

- NamedPipeServerStream(String) - ініціалізує новий екземпляр класу NamedPipeServerStream з вказанням імені каналу.
- NamedPipeServerStream(String, PipeDirection) - ініціалізує новий екземпляр класу NamedPipeServerStream з вказанням імені каналу та його напрямку.
- NamedPipeServerStream(String, PipeDirection, Int32) - ініціалізує новий екземпляр класу NamedPipeServerStream з вказанням імені каналу, його напрямку та максимального числа екземплярів сервера.

- NamedPipeServerStream(String, PipeDirection, Int32, PipeTransmissionMode) - ініціалізує новий екземпляр класу NamedPipeServerStream з вказанням імені каналу, його напрямку, максимального числа екземплярів сервера та режиму передачі.
- NamedPipeServerStream(String, PipeDirection, Int32, PipeTransmissionMode, PipeOptions) - ініціалізує новий екземпляр класу NamedPipeServerStream з вказанням імені каналу, його напрямку, максимального числа екземплярів сервера, режиму передачі та опцій каналу.

Конструктори класу NamedPipeClientStream:

- NamedPipeClientStream(String) - ініціалізує новий екземпляр класу NamedPipeClientStream з вказанням ім’я каналу.
- NamedPipeClientStream(String, String) - ініціалізує новий екземпляр класу NamedPipeClientStream з вказанням імен каналу та серверу.
- NamedPipeClientStream(String, String, PipeDirection) - ініціалізує новий екземпляр класу NamedPipeClientStream з вказанням імен каналу і серверу та напрямку каналу.
- NamedPipeClientStream(String, String, PipeDirection, PipeOptions) - ініціалізує новий екземпляр класу NamedPipeClientStream з вказанням імен каналу і серверу, напрямку каналу та опцій каналу [13].

1.6. Бібліотека розпаралелювання задач TPL

Для сучасних багатоядерних машин, які дозволяють паралельно виконувати одразу декілька процесів, стандартних засобів роботи з потоками в .NET вже не вистачає. Тому до фреймворку .NET була додана бібліотека паралельних задач **TPL** (*Task Parallel Library*), основний функціонал якої розташовується в просторі імен System.Threading.Tasks. Дана бібліотека дозволяє розпаралелити задачи та виконувати їх відразу на декількох

процесорах, якщо на цільовому комп'ютері є декілька ядер. Крім того, спрощується сама робота щодо створення нових потоків.

Бібліотека розпаралелювання завдань (TPL) удосконалює багатопоточне програмування двома основними способами:

- спрощує створення і застосування багатьох потоків;
- дозволяє автоматично використовувати декілька процесорів.

Таким чином, TPL відкриває можливості для автоматичного масштабування додатків з метою ефективного використання ряду доступних процесорів. Завдяки цим особливостям бібліотеки TPL рекомендується в більшості випадків до застосування для організації багатопотокової обробки.

Два підходи до паралельного програмування.

Застосовуючи бібліотеку TPL, паралелізм у програму можна ввести двома основними способами:

- **паралелізмом даних**,
- **паралелізмом завдань**.

При **паралелізмі даних** одна операція над сукупністю даних розбивається на два або більше паралельно виконуваних потоки, в кожному з яких обробляється частина даних. Так, якщо змінюється кожен елемент масиву, то, застосовуючи паралелізм даних, можна організувати паралельну обробку різних областей масиву в двох або більше потоках. Такі паралельно виконувані дії можуть привести до значного прискорення обробки даних у порівнянні з послідовним підходом.

Незважаючи на те, що паралелізм даних був завжди можливий і за допомогою класу Thread, побудова масштабованих рішень засобами цього класу вимагало чималих зусиль і часу. Ситуація змінилась з появою бібліотеки TPL, за допомогою якої масштабований паралелізм даних без особливих зусиль вводиться до програми.

При **паралелізмі завдань** дві або більше операції виконуються паралельно. Отже, такий спосіб введення паралелізму є різновидом паралелізму, який у минулому досягався засобами класу Thread. До переваг TPL відноситься простота застосування і можливість автоматично масштабувати виконання коду на декількох процесорів [2-4,14].

1.6.1. Клас Task

В основу TPL покладено клас Task визначений в просторі імен System.Threading.Tasks. Клас Task описує окрему задачу, яка запускається асинхронно в одному з потоків з пулу потоків. А в класі Thread інкапсулюється потік виконання. На системному рівні потік як і раніше залишається елементарною одиницею виконання, яку можна планувати засобами операційної системи. Але відповідність екземпляра об'єкта класу Task і потоку виконання не обов'язково виявляється взаємно-однозначною. Крім того, виконанням задач керує планувальник задач, який працює з пулом потоків. Це, наприклад, означає, що декілька задач можуть розділяти один і той же потік [15].

Створення задачі

Створити нову задачу у вигляді об'єкта класу Task і почати її виконання можна декількома різними способами.

Одним із способів є створення об'єкту типу Task за допомогою конструктора і запуск його за допомогою виклику методу Start().

У класі Task визначено кілька конструкторів. Найпростіший конструктор:

public Task (Action дія)

де *дія* позначає точку входу до коду, який представляє задачу, тоді як *Action* - делегат, визначений у просторі імен System. Найпростіша форма делегата Action виглядає наступним чином.

public delegate void Action()

Таким чином, точкою входу слугує метод, котрий **не приймає ніяких параметрів і не повертає ніяких значень**. Як тільки задача буде створена, її можна запустити на виконання, викликавши метод Start(). Нижче наведена одна з її форм.

public void Start()

Після виклику методу Start() планувальник задач запланує виконання задачи.

За замовчуванням задача виконується в фоновому потоці. Отже, при завершенні створюваного потоку завершується і сама задача. Після того, як задача завершена, її не можливо перезапустити.

Задачу можна створити і відразу ж почати її виконання, викликавши метод StartNew(), визначений в класі TaskFactory. У класі TaskFactory існують різні методи, які спрощують створення задач і управління ними. За замовчуванням об'єкт класу TaskFactory можна отримати з властивісті Factory, яка доступна тільки для читання в класі Task. Використовуючи цю властивість, можна викликати усякі методи класу TaskFactory. Метод StartNew() існує у множині форм. Нижче наведена найпростіша форма його оголошення.

public Task StartNew(Action action)

де *action* — точка входу до виконуваємої задачи. Спочатку в методі StartNew() автоматично створюється екземпляр об'єкта типу Task для дії, що визначається параметром *action*, а потім планується запуск задачи на виконання. Отже, необхідність у виклику методу Start() тепер відпадає.

Наприклад, наступний виклик методу StartNew() призведе до створення і запуску задачи tsk однією дією:

Task tsk = Task.Factory.StartNew(MyTask);

Після цього оператора відразу ж почне виконуватися метод MyTask(). Метод StartNew() виявляється більш ефективним в тих випадках, коли задача створюється і відразу ж починає виконання [2,3,15,16].

Застосування ідентифікатора задачі

У класі Task відсутня властивість Name для зберігання імені задачи. Але замість цього існує властивість Id для зберігання **ідентифікатора задачи**, за яким можна розпізнавати задачи. Властивість Id доступна тільки для читання і відноситься до типу int. Вона оголошується наступним чином.

public int Id {get;}

Кожна задача при створені отримує ідентифікатор. Значення ідентифікаторів унікальні, але не впорядковані. Тому один ідентифікатор задачи може з'явитися перед іншим, навіть при наявності не меншого значення. Ідентифікатор для задачи, яка виконується в даний час, можна знайти за допомогою властивості CurrentId. Вона доступна тільки для читання, відноситься до типу static і оголошується наступним чином.

public static Nullable <int> CurrentID {get;}

Властивість повертає задачу, яка виконується в даний час, або ж порожнє значення, якщо наведений код не є задачою.

Застосування методів очікування

Організувати очікування завершення задач можна за допомогою методів очікування, котрі спеціально задані в класі Task.

Найпростішим з них вважається метод Wait(), який припиняє виконання потоку, поки не завершиться задача, що викликалася. Нижче наведена найпростіша форма оголошення цього методу.

public void Wait()

При виконанні цього методу можуть бути згенеровані два винятки. Перший з них ObjectDisposedException генерується у випадку, якщо завдання звільнюється за викликом методу Dispose(). А другий виняток - AggregateException генерується у випадку, якщо задача сама генерує виняток або ж скасовується виконання. Як правило, відстежується і обробляється саме цей виняток. Задача може згенерувати декілька винятків. Наприклад, задача має породжені завдання, всі подібні винятки збираються в єдиний виняток типу AggregateException. Для з'ясовування, що ж сталося насправді, досить проаналізувати внутрішні винятки, котрі пов'язані з цим сукупним винятком [16].

Застосування лямбда-виразів у якості задачі

Крім використання звичайного методу в якості задачи, існує і інший, більш раціональний підхід: вказати лямбда-вираз як окрему задачу, яка розв’язується. Лямбда-вирази є особливою формою анонімних функцій, тому

можуть виконуватися як окремі задачи. Лямбда-вирази виявляються особливо корисними в тих випадках, коли єдиним призначенням методу є розв'язання одноразової задачи. Лямбда-вирази можуть становити окреме завдання або викликати інші методи. Так чи інакше, застосування лямбда-виразу як задачи може стати привабливою альтернативою іменованого методу.

Створення продовження задачі

Однією з дуже зручних особливостей бібліотеки TPL є можливість створювати продовження задачі. Продовження - це одна задача, яка автоматично починається після завершення другої задачі. Створити продовження можна, зокрема, за допомогою методу ContinueWith(), визначеного в класі Task. Нижче наведена найпростіша форма його оголошення.

public Task ContinueWith (Action<Task> дія_продовження)

де *дія_продовження* позначає задачу, яка буде запущена на виконання після закінчення виконання попередньої задачи. У делегата Action є єдиний параметр типу Task. Отже, варіант делегата Action, застосовуваного в даному методі, виглядає наступним чином.

public delegate void Action<in T> (T obj)

В даному випадку узагальнений параметр T позначає клас Task [15,16].

Повернення значення із задачі

Задача може повертати значення. І це дуже зручно з двох причин. По-перше, за допомогою задачі можна обчислити деякий результат. Подібним

чином підтримуються паралельні обчислення. По-друге, процес, що викликається, виявиться заблокованим до тих пір, поки не буде отримано результат. Це означає, що для організації очікування результату не потрібно ніякої особливої синхронізації.

Для того щоб повернути результат з задачи, досить створити цю задачу, використовуючи узагальнену форму Task класу Task. Нижче наведені два конструктора цієї форми класу Task.

public Task(Func<TResult> функція)
public Task(Func<Object, TResult> функція, Object стан)

де *функція* позначає виконуємий делегат типу Func. Тип Func використовується саме в тих випадках, коли задача повертає результат. У першому конструкторі створюється задача без аргументів, а в другому конструкторі - задача, що приймає аргумент типу Object, який передається як *стан*. Існують також інші конструктори даного класу.

Є також інші варіанти методу StartNew(), котрі доступні в узагальненій формі класу TaskFactory<TResult> і підтримують повернення результату з задачі. Нижче наведені варіанти даного методу, які застосовуються паралельно з тільки що розглянутими конструкторами класу Task.

public Task<TResult> StartNew (Func<TResult> функція)
public Task<TResult> StartNew(Func<Object, TResult> функція, Object стан)

Значення, що повертається задачею, отримується з властивості Result у класі Task, яке визначається наступним чином.

public TResult Result {get; internal set;}

Аксессор set є внутрішнім для даної властивості, і тому вона виявляється доступною в зовнішньому коді тільки для читання. Отже, задача отримання результату блокує код, який викликається, до тих пір, поки результат не буде підраховано [2,3,16].

Скасування задачі та обробка винятка AggregateException

У середовищі .NET впроваджена підсистема, що забезпечує структурований і дуже зручний спосіб скасування задачі. Вона ґрунтується на понятті ознаки скасування. Ознаки скасування підтримуються в класі Task за допомогою фабричного методу StartNew().

Скасування задачі, як правило, виконується в такий спосіб. Спочатку отримується ознака скасування з джерела ознак відміни. Потім ця ознака передається задачі, після чого вона повинна контролювати ознаку на предмет отримання запиту на скасування. Якщо отримано запит на скасування, задача повинна завершитися. В одних випадках цього виявляється досить для простого припинення задачі без будь-яких додаткових дій, а в інших — із задачі повинен бути викликаний метод ThrowIfCancellationRequested() для ознаки скасування. Завдяки цьому в коді стає відомо, що задача скасована.

Ознака скасування є екземпляром об'єкта типу CancellationToken, тобто структури, визначеної в просторі імен System.Threading. У структурі CancellationToken визначено декілька властивостей і методів.

Доступна тільки для читання властивість IsCancellationRequested оголошується наступним чином.

public bool IsCancellationRequested { get; }

Вона повертає логічне значення true, якщо скасування задачі було запрошене для виклику ознаки, а інакше — логічне значення false.

Метод ThrowIfCancellationRequested() оголошується наступним чином.

public void ThrowIfCancellationRequested()

Якщо ознака скасування, для якої викликається цей метод, отримала запит на скасування, то в даному методі генерується виняток OperationCanceledException. В іншому випадку ніяких дій не виконується.

У коді, який виконує дії скасування, можна організувати відстеження згаданого винятку з метою переконання, що скасування задачі дійсно відбулося. Як правило, з цією метою спочатку перехоплюється виняток AggregateException, а потім його внутрішній виняток аналізується за допомогою властивості InnerException або InnerExceptions. (Властивість InnerExceptions являє собою колекцію винятків) Ознака скасування виходить з джерела ознак відміни, яке являє собою об'єкт класу CancellationTokenSource, котрий визначений в просторі імен System.Threading. Для того щоб отримати цей показник, потрібно створити спочатку екземпляр об'єкта типу CancellationTokenSource (для цього можна скористатися конструктором класу CancellationTokenSource, який викликається за замовчуванням). Ознака скасування, пов'язана з даним джерелом, виявляється доступною через властивість Token, яка достапна тільки для читання та оголошується наступним чином.

public CancellationToken Token { get; }

Це і є ознакою, яка передається скасованій задачі. Для скасування в задачі повинна бути отримана копія ознаки скасування і організований контроль цієї ознаки з метою відслідковування самого скасування. Таке відстеження можна організувати трьома способами: опитуванням, методом зворотного виклику і за допомогою дескриптора очікування. Найпростіше організувати опитування, і тому розглянемо саме цей спосіб.

З метою опитування в задачі перевіряється згадана вище властивість IsCancellationRequested ознаки скасування. Якщо ця властивість містить логічне значення true, значить, скасування було запрошене, і задача повинна бути завершена.

Для створення задачи, з якої викликається метод ThrowIfCancellationRequested(), у випадку скасування задачі, звичайно потрібно передати ознаку скасування як самій задачі, так і конструктору класу Task. Передавання виконується безпосередньо або ж опосередковано через метод StartNew(). Передача ознаки скасування самої задачі дозволяє змінити стан задачі, яка скасовується, в запиті на скасування з зовнішнього коду. Далі буде використана наступна форма методу StartNew().

public Task StartNew (Action<Object> action, Object стан, CancellationToken ознака_скасування)

Ознака скасування передається через параметри, що позначені як *стан* і *ознака_скасування*. Це означає, що ознака скасування буде передана як делегату, який реалізує завдання, так і самому примірнику об'єкта типу Task. Нижче наведена форма, яка підтримує делегат Action.

public delegate void Action<in T> (T obj)

В даному випадку узагальнений параметр T позначає тип Object. В силу цього об'єкт obj повинен бути приведений всередині задачи до типу CancellationToken.

По завершенні роботи з джерелом ознак відміни слід звільнити його ресурси, викликавши метод Dispose(). Факт скасування задачи можна перевірити найрізноманітнішими способами. Один з підходів: перевірка значення властивості IsCanceled для екземпляра об'єкта типу Task. Якщо отримано логічне значення true, то задача була скасована [2,15,16].

1.6.2. Клас Parallel

Крім класу Task, який використовується так само, як і клас Thread, бібліотека TPL включає клас Parallel, який спрощує паралельне виконання коду і надає методи, котрі раціоналізують обидва види паралелізму: даних та задач.

Клас Parallel є статичним, і в ньому визначені методи For(), ForEach() і Invoke(). У кожного з цих методів є різні форми. Зокрема, метод For() виконує розпаралелювання циклу for, а метод ForEach() – розпаралелювання циклу foreach, і обидва методи підтримують паралелізм даних. А метод Invoke() підтримує паралельне виконання двох або більше методів. Ці методи дають перевагу реалізації на практиці поширених методик паралельного програмування, при цьому не вдаючись до управління задачами або потоками явно [2-4,17].

Розпаралелювання задач методом Invoke()

Метод Invoke(), визначений в класі Parallel, дозволяє виконувати один або декілька методів, котрі вказуються у вигляді його аргументів. Він також встановлює масштаб виконання коду, використовуючи доступні процесори, якщо є така можливість. Нижче наведена найпростіша форма його оголошення.

public static void Invoke (params Action[] actions)

Виконуємі методи повінні бути сумісними з делегатом Action.
Делегат Action оголошується наступним чином.

public delegate void Action()

Отже, кожен метод, який передається методу Invoke() як аргумент, не повинен ні приймати параметрів, ні повертати значення.

Метод Invoke() спочатку ініціює виконання, а потім очікує завершення всіх переданих йому методів.

Якщо система підтримує декілька процесорів, то передбачається, що методи будуть виконуватися паралельно, хоча це не гарантовано. Крім того, відсутня можливість вказати порядок виконання методів від першого і до останнього, і цей порядок може бути не таким же, як і в списку аргументів [2,4,17].

Застосування методу For()

У TPL паралелізм даних підтримується, зокрема, за допомогою методу For(), визначеного в класі Parallel. Цей метод існує в декількох формах. Найпростіша форма наведена нижче.

public static ParallelLoopResult For (int fromInclusive, int toExclusive, Action<int> body)

де fromInclusive позначає початкове значення, яке відповідає змінній управління циклом; його називають також ітераційним, або індексним, значенням; а toExclusive - значення, на одиницю більше кінцевого. На кожному кроці циклу змінна управління циклом збільшується на одиницю. Отже, цикл поступово просувається від початкового значення fromInclusive до кінцевого значенням toExclusive мінус одиниця. Циклічно виконуємий код позначається методом, який передається через параметр body. Цей метод повинен бути сумісним з делегатом Action <int>, який декларуються так.

public delegate void Action<in T> (T obj)

Для методу For() узагальнений параметр T повинен бути типу int. Значення, яке передається через параметр obj, буде наступним значенням змінної управління циклом. А метод, який передається через параметр body, може бути іменованим або анонімним. Метод For() повертає екземпляр об'єкта типу ParallelLoopResult, котрий описує стан завершення циклу. Для простих циклів цим значенням можна знехтувати.

Метод For() при наявності можливості дозволяє распараллелити виконання коду в циклі. А це, в свою чергу, може привести до підвищення продуктивності. Наприклад, процес перетворення масиву в циклі можна розділити на різні частини, які зможуть перетворюватися одночасно. Слід мати на увазі, що підвищення продуктивності не гарантується через відмінність у кількості доступних процесорів для різних середовищ виконання, а також через розпаралелювання дрібних циклів, що може скласти витрати, які перевищують заощаджений час.

Метод For() повертає екземпляр об'єкта типу ParallelLoopResult - це структура, в якій визначаються дві наступні властивості.

public bool IsCompleted {get;}
public Nullable<long> LowestBreakIteration {get;}

Властивість IsCompleted матиме логічне значення true, якщо виконані усі кроки циклу. Тобто, при нормальному завершенні циклу ця властивість буде мати логічне значення true. Якщо ж виконання циклу перерветься завчасно, то ця властивість отримає логічне значення false. Властивість LowestBreakIteration приймає найменше значення змінної управління циклом, якщо цикл перерветься завчасно при виклику методу ParallelLoopState.Break().

Для доступу до об'єкту типу ParallelLoopState слід використовувати форму методу For(), делегат якого приймає в якості другого параметра

поточний стан циклу. Нижче ця форма методу For() приведена в найпростішому вигляді.

public static ParallelLoopResult For (int fromInclusive, int toExclusive, Action<int, ParallelLoopState> body)

В даній формі делегат Action, який описує тіло циклу, визначається наступним чином.

public delegate void Action<in T1, in T2> (T1 arg1, T2 arg2)

Для методу For() узагальнений параметр T1 повинен бути типу int, а узагальнений параметр T2 - типу ParallelLoopState. Кожного разу, коли делегат Action викликається, поточний стан циклу приймає в якості аргументу arg2.

Для передчасного завершення циклу слід скористатися методом Break(), котрий викликається для екземпляра об'єкта типу ParallelLoopState всередині тіла циклу, що визначається параметром body. Метод Break() оголошується наступним чином.

public void Break()

Виклик методу Break() формує запит на найбільш раннє припинення паралельно виконуваного циклу, що може статися через декілька кроків циклу після виклику методу Break(). Але всі кроки циклу до виклику методу Break() все ж таки виконуються.

Слід мати на увазі, що окремі частини циклу можуть і не виконуватися паралельно. Так, якщо виконано декілька кроків циклу, то це ще не означає, що всі ці крокі представляють перші значення змінної управління циклом. Переривання циклу, при паралельному виконанні методу For(), нерідко виявляється корисним при пошуку даних. Так, якщо остаточне значення

знайдено, то продовжувати виконання циклу немає ніякої потреби. Переривання циклу може виявитися корисним і для випадку, якщо під час чергової операції зустрілися недостовірні дані.

Якщо потрібно зупинити цикл, котрий паралельно виконується методом For(), не звертаючи особливої уваги на будь-які кроки циклу, які ще можуть бути в ньому виконані, то для цього краще скористатися методом Stop(), ніж методом Break() [2-4,17].

Застосування методу ForEach()

Використовуючи метод ForEach(), можна створити варіант циклу foreach, який розпаралелюється. Існує декілька форм методу ForEach(). Нижче наведена найпростіша форма його оголошення.

public static ParallelLoopResult ForEach<TSource>
(IEnumerable<TSource> source, Action<TSource> body)

де *source* позначає колекцію даних, які обробляються в циклі, а *body* - метод, який буде виконуватися на кожному кроці циклу. У всіх масивах, колекціях і інших джерелах даних підтримується інтерфейс IEnumerable<T>. Метод, який передається через параметр body, приймає в якості свого аргументу значення або посилання на кожен опрацьований в циклі елемент масиву, але не його індекс. А в результаті повертаються відомості щодо стану циклу.

Як і для методу For(), паралельне виконання циклу методом ForEach() можна зупинити, викликавши метод Break() для екземпляра об'єкта типу ParallelLoopState, який передається через параметр body, за умови, що використовується наведена нижче форма методу ForEach() [2-4,17].

public static ParallelLoopResult ForEach<TSource>

(IEnumerable<TSource> source, Action<TSource, ParallelLoopState> body)

1.7. Можливості PLINQ

Однією з компонент Microsoft .NET, яка додає нативні можливості виконання запитів даних до різних мов, є **LINQ** (*Language Integrated Query* - запити, інтегровані в мову). LINQ розширює можливості мови C#, додаючи до неї вирази запитів, які схожі на твердження SQL та можуть бути використані для зручного отримання та обробки даних масивів, XML документів, реляційних баз даних та сторонніх джерел. LINQ визначає набір імен методів, а також правила перекладу, котрі має використовувати компілятор для перекладу текучих виразів у звичайні, використовуючи їх назву, лямбда-вирази та анонімні типи.

Починаючи з четвертої версії, .NET Framework включає в себе PLINQ (Parallel LINQ), яке реалізує виконання LINQ запитів паралельно. PLINQ може виконувати частини запиту одночасно в різних потоках, що дозволяє швидше отримати результат.

PLINQ є паралельним варіантом мови інтегрованих запитів LINQ і тісно пов'язаний з бібліотекою TPL. PLINQ застосовується головним чином для досягнення паралелізму даних всередині запиту [2-4,18].

1.7.1. Клас ParallelEnumerable

Основу PLINQ становить клас ParallelEnumerable, визначений у просторі імен System.Linq. Це статичний клас, який має багато методів розширення та підтримує паралельне виконання операцій. Він є паралельним варіантом стандартного для LINQ класу Enumerable. Багато його методів є розширенням класу ParallelQuery, а деякі з них повертають об'єкт типу ParallelQuery. У класі ParallelQuery інкапсулюється послідовність операцій, котрі підтримує

паралельне виконання. Існує як узагальнений, так і неузагальнений варіанти даного класу.

Розпаралелювання запиту методом AsParallel()

Найзручнішим засобом PLINQ є можливість легко створювати паралельний запит, для цього потрібно лише викликати метод AsParallel() для джерела даних.

Метод AsParallel() визначений у класі ParallelEnumerable та повертає джерело даних, яке інкапсульоване в екземплярі об'єкта типу ParallelQuery. Це дозволяє підтримувати методи розширення паралельних запитів. Після виклику даного методу запит розподіляє джерело даних на частини та оперує з кожною з них для отримання максимальної вигоди від розпаралелювання. Якщо розпаралелювання виявляється неможливим або неприйнятним, то запит зазвичай виконується послідовно. Таким чином, додавання у вихідний код єдиного виклику методу AsParallel() виявляється достатнім для перетворення послідовного запиту LINQ на паралельний запит LINQ. Для простих запитів це єдина необхідна умова.

Існують як узагальнені, так і неузагальнені форми методу AsParallel(). Нижче наведена найпростіша узагальнена його форма:

public static ParallelQuery AsParallel(this IEnumerable source)
public static ParallelQuery<TSource> AsParallel<TSource>
(this IEnumerable<TSource> source)

де *TSource* означає тип елементів у послідовному джерелі даних *source*.

Застосування методу AsOrdered()

За замовчуванням порядок формування результуючої послідовності в паралельному запиті зовсім не обов'язково має відображати порядок формування вихідної послідовності. Більш того, результуючу послідовність слід розглядати як практично невпорядковану. Якщо ж результат повинен відображати порядок організації джерела даних, його потрібно запросити спеціально за допомогою методу AsOrdered(), який визначено в класі ParallelEnumerable. Нижче наведені узагальнена та неузагальнена форми цього методу.

public static ParallelQuery AsOrdered(this ParallelQuery source)
public static ParallelQuery<TSource> AsOrdered<TSource>
(this ParallelQuery<TSource> source)

де *TSource* означає тип елементів у джерелі даних *source*. Метод *AsOrdered()* можна викликати тільки для об'єкта типу *ParallelQuery*, оскільки він є методом розширення класу *ParallelQuery* [18].

Скасування паралельного запиту

Паралельний запит скасовується так само, як і задача. В обох випадках скасування спирається на структуру CancellationToken, яка отримується з класу CancellationTokenSource. В результаті отримана ознака скасування передається запиту за допомогою методу WithCancellation(). Скасування паралельного запиту здійснюється методом Cancel(), який викликається для джерела ознак скасування.

Головна відмінність скасування паралельного запиту від скасування задачі полягає у наступному: коли паралельний запит скасовується, він генерує виняток OperationCanceledException, а не AggregateException. Але для випадків, коли запит здатний згенерувати декілька винятків, виняток

OperationCanceledException може бути об'єднаний у сукупний виняток AggregateException. Тому відстежувати краще обидва види винятків.

Нижче наведено форму оголошення методу WithCancellation().

public static ParallelQuery<TSource> WithCancellation (this ParallelQuery source, CancellationToken CancellationToken)

де *source* означає запит, який визивається, а *CancellationToken* — ознаку скасування. Цей метод повертає запит, який підтримує зазначену ознаку скасування [2-4].

1.7.2. Інші засоби та ефективність PLINQ

У PLINQ доступні і багато інших засобів, що допомагають підлаштовувати паралельні запити під конкретну ситуацію. Так, при виклику методу WithDegreeOfParallelism() можна вказати максимальну кількість процесорів, котрі виділяються для обробки запиту, а при виклику методу AsSequential() — запитати послідовне виконання частини паралельного запиту. Якщо потік, що викликається, очікує результатів від циклу foreach, не потрібно блокувати, то для цього можна скористатися методом ForAll(). Всі ці методи визначені в класі ParallelEnumerable. А для випадків, коли PLINQ повинен за умовчанням підтримувати послідовне виконання, можна скористатися методом WithExecutionMode(), передавши йому як параметр ознаку ParallelExecutionMode.ForceParallelism.

Не всі запити виконуються швидше тільки тому, що вони розпаралелені. Недоліки, пов'язані зі створенням паралельних потоків та управлінням їх виконанням, можуть "перекрити" всі переваги, які дає розпаралелювання. Взагалі кажучи, якщо джерело даних виявляється досить дрібним, а необхідна обробка даних — дуже короткою, то впровадження паралелізму може і не призвести до прискорення обробки запиту [2,4,18].

2. ЛАБОРАТОРНИЙ ПРАКТИКУМ

Лабораторна робота № 1. Потоки

Мета роботи: вивчити базову структуру концепції багатопоточності мови C#; навчитися створювати та запускати дочірні потоки.

Одним з ключових аспектів у сучасному програмуванні є багатопоточність. Ключовим поняттям при роботі з багатопотоковістю є потік. Потік предствляет деяку частину коду програми. При виконанні програми кожному потоку виділяється певний квант часу. І за допомогою багатопоточності можна виділити у додатку декілька потоків, які будуть виконувати різні завдання одночасно.

2.1.1. Створення потоку

Приклад 1.1. Програма створює окремий потік за допомогою методу Run().

```
using System;
using System.Threading;

namespace l1_1
{
    class MyThread
    {
        public int Count;

        public void Run()
        {
            Console.WriteLine("Новий потік запущено.");
            Console.Write("Підрахунок: ");
            do
            {
                Thread.Sleep(500);
                Console.Write(Count + " ");
                Count++;
            } while (Count <= 10);
```

```
            Console.WriteLine("\nНовий потік закінчено.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyThread mt = new MyThread();

            Thread newThrd = new Thread(mt.Run);  // створюється потік

            Console.WriteLine("\t Головний потік підрахунок = " +
mt.Count);

            newThrd.Start();       // запуск нового потоку

            for (int i = 20; i <= 30; i++)
            {
                Thread.Sleep(500);
                Console.WriteLine("\t Головний потік = " + i);
            }

            newThrd.Join();     // очікування завершення нового потоку

            Console.WriteLine("\t Головний потік підрахунок = " +
mt.Count);

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу MyThread та об'єкт класу Thread, який відповідає за створенна та управління потоками. У конструктор класу Thread передається метод Run(). Запуск нового потоку починається після виклику методу Start() з класу Thread.

У створенному потоці метод Run() виконує підрахунок від 0 до 10, включно. На початку роботи методу виводиться рядок «Новий потік запущено.», що означає, що другий потік почав віконуватися. Виведення чисел проводиться з частотою 0.5 сек. Така частота реалізована за допомогою методу

Sleep(). Після закінчення роботи методу виводиться рядок «Новий потік закінчено.», що означає, що другий потік закінчив виконання. Одночасно головний потік виводить текст " Головний потік = " та числа від 20 до 30 з частотою 0.5 сек.

Метод Join() у основному потоці виконує роль очікування, поки новий потік не завершить роботу.

Результат роботи програми наступний:

Головний потік підрахунок = 0
Новий потік запущено.
Підрахунок: Головний потік = 20
0 Головний потік = 21
1 Головний потік = 22
2 Головний потік = 23
3 Головний потік = 24
4 Головний потік = 25
5 Головний потік = 26
6 Головний потік = 27
7 Головний потік = 28
8 Головний потік = 29
9 Головний потік = 30
10
Новий потік закінчено.
Головний потік підрахунок = 11

2.1.2. Стан потоків

Приклад 1.2. Програма виводить стан потоку за допомогою властивості ThreadState.

```
using System;
using System.Threading;

namespace l1_2
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread newThrd;

            newThrd = Thread.CurrentThread;

            Console.WriteLine(newThrd.ThreadState);

            newThrd.IsBackground = true;      // робимо потік фоновим

            Console.WriteLine(newThrd.ThreadState);

            newThrd.IsBackground = false; // робимо потік пріоритетним

            Console.WriteLine(newThrd.ThreadState);

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу Thread, який відповідає за створена та управління потоками. Цьому об’єкту присвоюється значення головного потоку.

За допомогою властивості ThreadState у вікно виводиться стан потоку. Для того щоб зробити потік **фоновим**, присвоюється логічне значення **true** властивості **IsBackground**. Знову виводиться стан потоку. Властивості **IsBackground** присвоюється логічне значення **false** та знову виводиться стан потоку.

Результат роботи програми наступний:

Running

Background

Running

2.1.3. Передача параметра у потік

Приклад 1.3.1. Передача параметра у потік за допомогою делегата **ParameterizedThreadStart**.

```
using System;
using System.Threading;

namespace l1_3
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 4;
                                        // створення нового потоку

            Thread myThread = new Thread(new
ParameterizedThreadStart(Count));

            myThread.Start(number);  // передаємо змінну у новий потік

            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine("Головний потік: "+ i * i);

                Thread.Sleep(300);
            }

            Console.ReadLine();
        }
        public static void Count(object x)
        {
            for (int i = 1; i <= 10; i++)
            {
                int n = (int)x;

                Console.WriteLine("\tДругий потік: "+ i * n);

                Thread.Sleep(400);
            }
        }

    }
```

```
}
```

Опис роботи програми.

Після створення потоку методу myThread.Start(number); передається змінна, значення якої необхідно передати у потік.

При використанні ParameterizedThreadStart стикаємося з обмеженням: можна запускати в другому потоці тільки такий метод, який в якості єдиного **параметра** приймає об'єкт типу **object**.

Тому в даному випадку треба додатково привести передане значення до типу int, щоб його використовувати в обчисленнях [13].

Результат роботи програми наступний:

```
Другий потік: 4
Головний потік: 1
Головний потік: 4
    Другий потік: 8
Головний потік: 9
    Другий потік: 12
Головний потік: 16
    Другий потік: 16
Головний потік: 25
Головний потік: 36
    Другий потік: 20
Головний потік: 49
    Другий потік: 24
Головний потік: 64
    Другий потік: 28
Головний потік: 81
Головний потік: 100
    Другий потік: 32
```

Другий потік: 36

Другий потік: 40

Якщо треба передати не один, а кілька параметрів різного типу, то на допомогу приходить класовий підхід.

Приклад 1.3.2. Передача кілька параметрів різного типу у потік за допомогою делегата **ParameterizedThreadStart**.

```
using System;
using System.Threading;

namespace l1_4
{
    public class Counter
    {
        public int x;
        public int y;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Counter counter = new Counter();
            counter.x = 4;
            counter.y = 5;

            Thread myThread = new Thread(new
ParameterizedThreadStart(Count));

            myThread.Start(counter);
        }

        public static void Count(object obj)
        {
            for (int i = 1; i <= 10; i++)
            {
                Counter c = (Counter)obj;

                Console.WriteLine("Другий потік: "+ i * c.x * c.y);

            }
        }
    }
}
```

Опис роботи програми.

Спочатку визначається спеціальний клас Counter, об'єкт якого буде передаватися в другій потік та в методі Main() передамо його в другій потік.

Результат роботи програми наступний:

Другий потік: 20
Другий потік: 40
Другий потік: 60
Другий потік: 80
Другий потік: 100
Другий потік: 120
Другий потік: 140
Другий потік: 160
Другий потік: 180
Другий потік: 200

Але тут знову є одне обмеження: метод Thread.Start не є тіпобезпечним, тобто можливо передати в нього будь-який тип, і потім приводити переданий об'єкт до потрібного типу.

Для вирішення даної проблеми рекомендується оголошувати всі використовувані методи і змінні в спеціальному класі, а в основній програмі запускати потік через ThreadStart.

Приклад 1.3.3. Передача кілька параметрів різного типу у потік за допомогою спеціального класу.

```
using System;
using System.Threading;

namespace l1_5
```

```
{
    public class Counter
    {
        private int x;
        private int y;

        public Counter(int _x, int _y)
        {
            this.x = _x;
            this.y = _y;
        }

        public void Count()
        {
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine("Другий потік: " + i * x * y);

                Thread.Sleep(400);
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Counter counter = new Counter(5, 4);

            Thread myThread = new Thread(new
ThreadStart(counter.Count));

            myThread.Start();
        }
    }
}
```

2.1.4. Запуск кількох потоків з параметром

Приклад 1.4. Запуск кількох потоків з параметром.

```
using System;
using System.Threading;

namespace l_1_4
{
    class MyThread
    {
        public int Count;
```

```
    public void Run(Object x)
    {
        Console.WriteLine("Thread " + x + " is started.");
        do
        {
            Thread.Sleep(2 + 5 * (int)x);
            Console.WriteLine(" Thread- " + x + " - " + Count);
            Count++;
        } while (Count <= 10);

        Console.WriteLine("\nThread " + x + " is finished.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        int n = 3;
        Console.WriteLine("\t\tMain start");

        MyThread[] mt = new MyThread[n];

        for (int i = 0; i < n; i++)
        {
            mt[i] = new MyThread();
        }

        Thread[] newThrd = new Thread[n];

        for (int i = 0; i < n; i++)
        {
            newThrd[i] = new Thread(mt[i].Run);
        }

        for (int i = 0; i < n; i++)
        {
            newThrd[i].Start(i);
        }
        for (int i = 0; i < 20; i++)
        {
            Console.WriteLine("\t\tMain ");
            Thread.Sleep(7);
        }
        for (int i = 0; i < n; i++)
        {
            newThrd[i].Join();
        }

        Console.WriteLine("\t\tMain end");
```

```
            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

Створюється масив mt посилань на екземпляри класу MyThread з n елементів. Для кожного елемента масиву створюється екземпляр класу MyThread. Потім створюється масив newThrd посилань на екземпляри класу Thread з n елементів. Для кожного елемента масиву newThrd створюється потік з зазначенням виконання методу Run.

Після створення масиву потоків, у циклі за допомогою методу Start запускаються на виконання всі потоки, яким передається змінна `i`, значення якої необхідно передати у потік. У циклі ця змінна приймає значення від 0 до n-1.

В методі Run класу MyThread передане значення параметру явно наводиться до типу int, щоб його використовувати в обчисленнях [13].

Результат роботи програми наступний:

```
Main start
            Main
Thread 2 is started.
Thread 1 is started.
Thread 0 is started.
            Main
 Thread- 1 - 0
 Thread- 0 - 0
 Thread- 2 - 0
 Thread- 1 - 1
```

```
        Main
Thread- 0 - 1
        Main
Thread- 1 - 2
Thread- 2 - 1
Thread- 0 - 2
        Main
Thread- 1 - 3
Thread- 0 - 3
Thread- 2 - 2
Thread- 2 - 3
Thread- 1 - 4
        Main
Thread- 0 - 4
        Main
Thread- 2 - 4
Thread- 1 - 5
Thread- 0 - 5
        Main
Thread- 0 - 6
Thread- 1 - 6
Thread- 2 - 5
Thread- 1 - 7
Thread- 0 - 7
        Main
Thread- 2 - 6
Thread- 1 - 8
Thread- 0 - 8
        Main
```

```
Thread- 0 - 9
Thread- 1 - 9
Thread- 2 - 7
        Main
Thread- 1 - 10

Thread 1 is finished.
Thread- 0 - 10
        Main

Thread 0 is finished.
Thread- 2 - 8
        Main
Thread- 2 - 9
        Main
        Main
Thread- 2 - 10

Thread 2 is finished.
        Main
        Main
        Main
        Main
        Main
        Main end
```

2.1.5. Здійснення потокобезпечних викликів елементів управління у проектах Windows Forms

При використанні багатопоточності для поліпшення продуктивності додатків Windows Forms під час виклику елементів управління необхідно дотримуватися принципів безпеки потоків.

Доступ до елементів управління Windows Forms по суті не є потокобезпечним. При наявності двох або більше потоків, які контролюють стан елемента управління, цей елемент управління можна перевести в неузгоджений стан. Крім того, можуть виникнути інші помилки, пов'язані з потоками, включаючи стани гонок і взаємоблокировок. Дуже важливо забезпечити потокобезпечний доступ до елементів управління.

Виклик елемента управління з будь-яких інших потоків, за винятком потоку, в якому було створено елемент управління, без використання методу Invoke є порушенням безпеки.

Використання методів Invoke/BeginInvoke

Ці методи виконують зазначені делегати в тому потоці, в якому елемент управління був створений. Метод *Invoke* викликає делегат синхронно, метод *BeginInvoke* - асинхронно.

Щоб визначити, чи потрібно використовувати метод *Invoke* потрібно використовувати властивість InvokeRequired.

1. Якщо властивість InvokeRequired повертає true, викликається метод Invoke з делегатом, котрий фактично викликає елемент управління.

2. Якщо властивість InvokeRequired повертає false, викликається елемент управління безпосередньо.

Наприклад, оголошується делегат:

```
delegate void Del(string text);
```

та викликається метод Invoke

```
textBox1.Invoke(new Del((s) => textBox1.Text=s), "newText");
```

Замість оголошення нових делегатів можна використовувати готові, Action або Func.

Приклад готового, потокобезпечного методу:

```
void SetTextSafe(string newText)
{
     if (textBox1.InvokeRequired)
               textBox1.Invoke(new Action<string>((s) =>
          textBox1.Text = s), newText);
     else
          textBox1.Text = newText;
}
```

Приклад 1.5.1. Робота потоку з параметром з виведенням значення у елемент керуванням на формі у додатку Windows Forms.

Опис роботи програми.

На **формі** після натискання на кнопку виконується розрахунок в окремому потоці: складаються значення двох текстбоксів і результат виводиться у третій текстбокс.

Обчислювальні операції повинні виконуватися в одному потоці, а відображення в іншому.

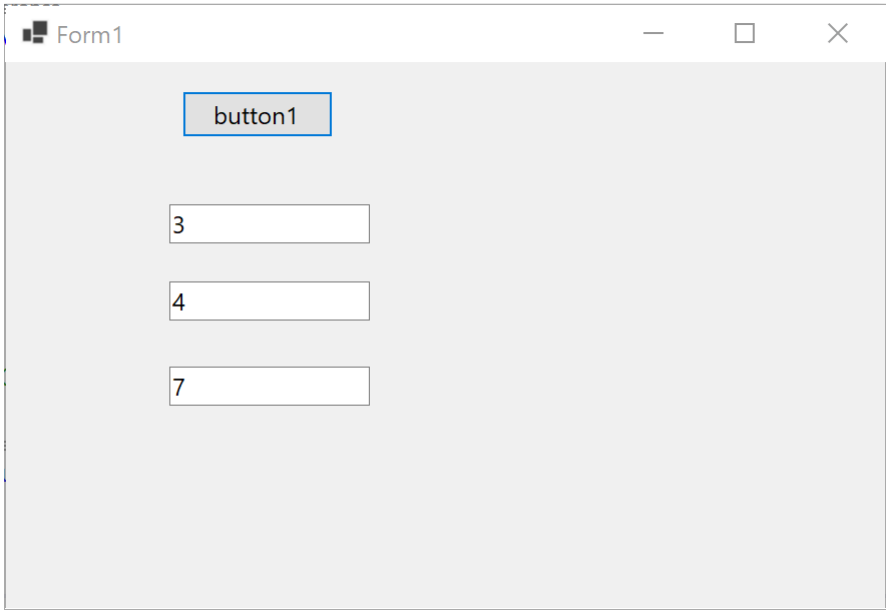
У потоці існують обмеження, що метод Start() може приймати тільки один аргумент типу object. Тому доведеться "упакувати" всі дані і передати в потік, а всередині потоку "розпакувати" їх.

```
using System.Threading;
using System.Windows.Forms;
```

```
namespace l_1_5_1
{
public partial class Form1: Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        Thread thread = new Thread(Raschet);

            //  запаковуємо

         Data temp;
        temp.a = int.Parse(textBox1.Text);
        temp.b = int.Parse(textBox2.Text);

            // Починаємо виконання потоку та передаємо дані

         thread.Start(temp);
    }

    private void Raschet(object input)
    {
        // Розпаковуємо всередині потоку

         Data data = (Data)input;
        int res = data.a + data.b;
        Invoke(new Action(()=> textBox3.Text= res.ToString()));
    }

      // Окремий тип для пакування/розпакування

     struct Data
       {
       public int a;
       public int b;
     }
   }
}
```

Дані, які потрібно повернути з потоку, записуються через метод Invoke() - при запуску нового потоку він працює одночасно з основним потоком. А основний потік, в свою чергу, вже закінчив обробку події натискання на кнопку і чекає подальших подій від користувача.

Результат роботи програми наступний:



Приклад 1.5.2. Запуск кілька потоків з параметрами з виведенням значенння у елементи керування ListBox та Label на формі у додатку Windows Forms.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;

namespace Forms_thread
{

    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

        }

        private void button1_Click(object sender, EventArgs e)
        {
```

```
            int CountThreads = 4;

            Thread[] Threads = new Thread[CountThreads];

            for (int p = 1; p < CountThreads; p++)
            {
                Threads[p] = new Thread(dosomething);
                Threads[p].Start(p*100);
            }

        }

        void dosomething(object data)
        {
            int i = (int)data;

            for (int j=0;j< 3;j++)
            {
                if (label1.InvokeRequired)
                    label1.Invoke(new Action(() => label1.Text =
i.ToString()));
                if (listBox1.InvokeRequired)
                     listBox1.Invoke(new Action(()
=>listBox1.Items.Add(i)));
                Thread.Sleep(300);
                i++;
            }
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            for (int i = 0; i < 2; i++)
            {
                Thread mythr = new Thread(dosomething);
                mythr.Start(i * 100);
            }
        }

    }
}
```

Опис роботи програми.

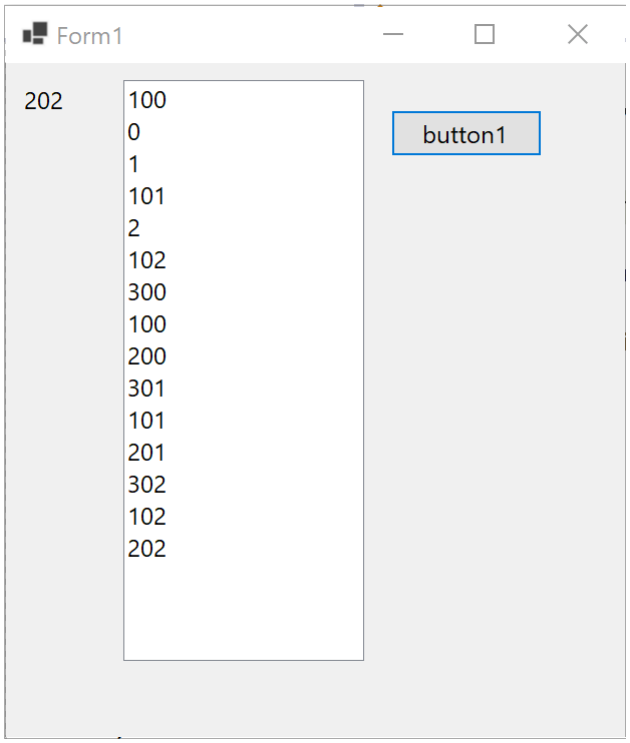
Коли завантажиться форма один за іншим запустяться два потоки, що виконують метод dosomething з передачею йому параметра. Кожен потік отримує своє значення параметру.

У методі dosomething передане значення параметру явно наводиться до типу int. Далі у циклі записується це значення у елементи керування ListBox та Label на формі. При цьому виконується перевірка значення властивості

InvokeRequired та для запису викликається метод Invoke. Значення параметру інкрементується.

При натисканні на кнопку створюється ще CountThreads потоків, що виконують метод dosomething з параметром data. У кожний потік передається своє значення цього параметра.

Результат роботи програми наступний:



Приклад 1.5.3. Створення кілька елементів керування ListBox на формі Form у додатку Windows Forms.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```

```
namespace control
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        int x;

        List<Label> labels;

        private void button1_Click(object sender, EventArgs e)
        {
            Random r = new Random();

            x = Convert.ToInt16(textBox1.Text);

            labels = new List<Label>(){label1, label2, label3};

            ListBox[] lb = new ListBox[x];

            for (int i = 0; i < x; i++)
            {
                lb[i] = new ListBox();
                lb[i].Visible = true;
                lb[i].Show();
                lb[i].Items.Add(i);
                lb[i].Location = new Point(200+i*150, i*150);
                lb[i].BackColor = Color.FromArgb(r.Next(255),
r.Next(255), r.Next(255));

                this.Controls.Add(lb[i]);
            }

            foreach ( Label s in labels )
            {
                {
                    s.Visible = true;
                }
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            int n=this.Controls.Count;

            for (int i = n-1; i >= n-x; i--)
            {
                Controls.RemoveAt(i);
            }
            foreach (Label s in labels)
```

```
                {
                    {
                        s.Visible = false;
                    }
                }
            }
        }
}
```

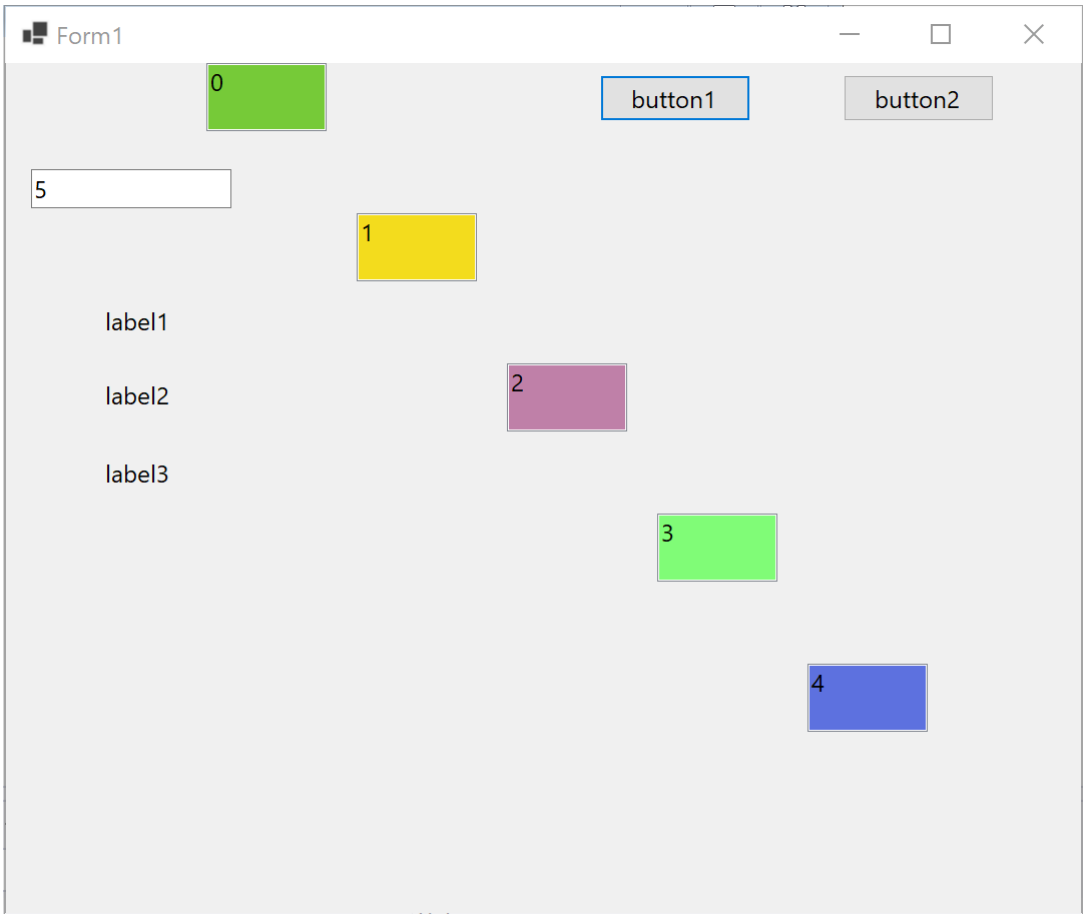
Опис роботи програми.

Коли завантажиться форма один за іншим запустяться два потоки, що виконують метод dosomething з передачею йому параметра. Кожен потік отримує своє значення параметру.

У методі dosomething передане значення параметру явно наводиться до типу int. Далі у циклі записується це значення у елементи керування ListBox та Label на формі. При цьому виконується перевірка значення властивості InvokeRequired та для запису викликається метод Invoke. Значення параметру інкрементується.

При натисканні на кнопку створюється ще CountThreads потоків, що виконують метод dosomething з параметром data. У кожний потік передається своє значення цього параметра.

Результат роботи програми наступний:



2.1.6. Інші приклади роботи потоків у проектах Form

Приклад 1.6.1.

```
class Class1
    {
        Form1 f1;
        public Class1(Form1 x)
        {
            f1 = x;
        }
        public void dosomething(object data)
        {
            int i = (int)data;
            f1.Invoke(f1.myDelegate, new Object[]{i});
        }
    }
//-----------------------------------------------
public partial class Form1: Form
    {
```

```
        public Form1()
        {
            InitializeComponent();
        }
        public delegate void AddListItem(int x);
        public AddListItem myDelegate;

        public void AddListItemMethod(int x)
        {
            String myItem;
            Thread.Sleep(600 - x);
            for (int i = 1; i < 6; i++)
            {
                myItem = "MyListItem" + (x+i).ToString();
                listBox2.Items.Add(myItem);
                listBox2.Update();
                Thread.Sleep(600-x);
            }
        }

       private void button1_Click(object sender, EventArgs e)
        {
            Class1 cc = new Class1(this);
            myDelegate = new AddListItem(AddListItemMethod);
            int CountThreads = 5;

            Thread[] Threads = new Thread[CountThreads + 1];
            for (int p = 1; p <= CountThreads; p++)
            {
                Threads[p] = new Thread(cc.dosomething);
                Threads[p].Start(p * 100);
            }
        }

//-------------------------------------------------

class MyThread
    {
        public int Count;
        Form1 ff;
        public MyThread(Form1 f1)
        {
            ff = f1;
        }
        public void Run(object x)
        {
            int n = (int)x;
            if (ff.lb.InvokeRequired)
            {
                ff.lb.Invoke(new Action(() => {
ff.lb.Items.Add("Thread "+n+" is started."); }));
            }
            do
```

```
            {
                Thread.Sleep(30);
               if (ff.lb.InvokeRequired)
                {
                    ff.lb.BeginInvoke(new Action(() => {
ff.lb.Items.Add((Count + n*100)+" "); }));
                }
                Count++;
            } while (Count <= 10);
            if (ff.lb.InvokeRequired)
            {
                ff.lb.Invoke(new Action(() => {
ff.lb.Items.Add("Thread "+n+" is finished."); }));
            }
        }
    }
}

//-----------------------------------------------
```

```
public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        public ListBox lb;

        private void button1_Click(object sender, EventArgs e)
        {
            lb = listBox1;
            MyThread mt = new MyThread(this);
            Thread[] newThrd = new Thread[3];
            for(int i=0;i<3;i++)
            {
                newThrd[i] = new Thread(mt.Run);
            }

           listBox1.Items.Add("main Count = " + mt.Count);

            for (int i = 0; i < 3; i++)
            {
                newThrd[i].Start(i);
                Thread.Sleep(200);
            }
            for (int i = 0; i < 3; i++)
            {
                if (!newThrd[i].IsAlive)
                {
                    listBox1.Items.Add("main Count="+ mt.Count);
                }
                else
                {
```

```
                    listBox1.Items.Add("potok " +i+ " - " +
newThrd[i].ThreadState);
                }
            }
        }
    }
}
```

Опис роботи програми.

У формі описані делегат і метод, який йому відповідає за записом в listBox2.

Створюється екземпляр делегату із зазначенням методу, потім екземпляр класу. Класу в конструкторі передаємо посилання на форму. Потім створюється n потоків (виконуватимуть метод cc.dosomething), запускаються і передаються параметри. У методі cc.dosomething - викликається делегат з параметром.

Приклад 1.6.2. Один потік записує у Control (Label)

```
using System.Threading;

namespace form_potok_1
{
    public partial class Form1: Form
    {
        private Label lb;
        public Form1()
        {
            InitializeComponent();

            lb = new Label { Parent = this };
            new Thread(Work) { IsBackground = true }.Start();

                                    //запускаємо потік
        }
        void Work()
        {
            var counter = 0;
                                     //цикл зчитування даних
            while (true)
            {
                counter++;
                Thread.Sleep(200);
                SetText(lb, counter.ToString());
```

```
                                //виводимо показання у Label
            }
        }
        private void SetText(Control ctrl, string text)
        {
            if (InvokeRequired)
            {
                BeginInvoke((Action)(() => SetText(ctrl,
text)));
                return;
            }
            ctrl.Text = text;
        }
    }
}
```

Приклад 1.6.3. У проекті Form головний потік створює n дочірних потоків.

```
   public partial class Form1: Form
    {
        int n = 3;
        private Label [] lb;
        public Form1()
        {
            InitializeComponent();
            lb = new Label[n+1];
            for (int i = 0; i < n+1; i++)
            {
                Point p = new Point(i * 100, 10);
                lb[i] = new Label { Parent = this };
                lb[i].Location=p;
            }
            Thread[] tt = new Thread[n];
            for (int i=0;i<n;i++)
            {
                tt[i]= new Thread(Work) { IsBackground = true };
                tt[i].Start(i);
            }
            Thread mm = new Thread(MainWork);
            mm.Start();
        }
        void Work(object num)
        {
            int x = (int)num;
            var counter = 0;
                                    //цикл зчитування даних
            while (true)
            {
                counter++;
```

```
                Thread.Sleep(100+100*x);
                SetText(lb[x], counter.ToString());

                                        //виводимо показання у Label
            }
        }
        private void SetText(Control ctrl, string text)
        {
            if (InvokeRequired)
            {
                BeginInvoke((Action)(() => SetText(ctrl,
text)));
                return;
            }
            ctrl.Text = text;
        }
        void MainWork()
        {
            var counter = 0;
            while (true)
            {
                counter++;
                Thread.Sleep(100);
                GetText(lb[n], counter.ToString());

                                        //виводимо показання у Label
            }
        }
        private void GetText(Control ctrl, string text)
        {
            text+=" Main";
            if (InvokeRequired)
            {
                BeginInvoke((Action)(() => SetText(ctrl,
text)));
                return;
            }
            ctrl.Text = text;
        }
    }
```

ЗАВДАННЯ

1. Виконайте послідовний виклик методу Run() у прикладі 1.1 (без створення потоку). Наочно покажіть різницю між роботою програми з другим потоком і без створення нового потоку.

2. Проаналізуйте та виконайте приклади 1.1-1.4 з практичної частини. Наочно покажіть паралельність виконання потоків.
3. Виконайте приклади 1.5-1.6 в проектах C# Windows Forms.
4. Виконайте свій варіант відповідно до списку у підгрупи в проекті C# Windows Forms.

Варіант 1

Написати програму, яка створює N потоків, кожному з яких виділяється 1/N частина вікна програми. Всі потоки фарбують свою частину вікна випадковими кольорами з різним інтервалом часу.

Варіант 2

Написати програму, яка створює потік по натисненню однієї з клавіш клавіатури. Кожному створеному таким чином потоку відповідає окружність у вікні застосунку, яка з’являється у випадковому місці вікна застосунку і рухається або по вертикалі, або по горизонталі. Коли окружність досягне межі вікна, вона змінює напрямок свого руху на протилежний.

Варіант 3

Розробити програму, яка ділить робочу область вікна на N частин (потоків). Всі частини через заданий проміжок часу змінюють свій колір. При натисканні миші в одній з частин, інтервал зміни кольору в цій частині збільшується на певний інтервал часу.

Варіант 4

Написати програму, яка запускає новий потік при натисканні лівої клавіші миші. Потік починає виводити зростаючу числову послідовність в поточну позицію курсору миші. При натисканні правої клавіші миші програма видаляє потік, координати якого ближче всього до положення курсору.

Варіант 5

Написати програму, в якій перший потік при кожному натисканні на кнопку «Завантажити» читає обраний файл з зображенням і виводить його у вікно програми. При кожному натисканні на кнопку «Поворот», другий потік виконує поворот зображення на 90 градусів.

Варіант 6

Написати програму, яка читає всі текстові файли, що знаходяться в папці з файлом, який виконується, і виводить вміст на екран у випадкову позицію з різним інтервалом. Для читання файлів використовувати декілька потоків.

Варіант 7

Написати програму, в якій з певним інтервалом часу створюється потік, який керує переміщенням прямокутника в межах вікна (всього N потоків). При натисканні миші всередині кожного з прямокутників, потік, якому належить цей прямокутник, видаляється. Максимальна кількість потоків задається і може буди змінена.

Варіант 8

Написати програму, в якій створюється кілька потоків, кожний з яких переміщає прямокутник горизонтально по вікну програми. При натисканні клавіші пробіл, створюється новий потік, який, в свою чергу, створює прямокутник, який рухається вертикально. При зіткненні прямокутника з будь-яким іншим, обидва знищуються; при виході прямокутника за межі вікна потік також знищується.

Варіант 9

Написати програму, яка створює три потоки, кожному з яких виділяється третина вікна застосунку. Перший потік виводить у свою область зростаючу числову послідовність 0,1,2,…, другий – послідовність чисел

Фібоначчі. Третій потік заповнює свою область вікна прямокутниками випадкового розміру і кольору.

Варіант 10

Написати програму, яка при натисканні правої клавіші миші створює потік, який виводить зростаючий ряд в позицію курсору, при натисканні лівої клавіші миші створює ряд, що зменшується. Максимальна кількість потоків задається і може буди змінена.

Варіант 11

Розробити програму, в якій на формі містяться дві геометричні фігури. Вони слідують за курсором по одній з осей: одна з фігур — по Y , інша — по X. Кожна фігура керується своїм потоком.

Варіант 12

Написати програму, яка містить два потоки, кожен з яких керує рухом однієї з двох куль. Перша куля рухається горизонтально, друга – вертикально. Швидкість куль різна. Коли куля досягне межі клієнтської області вікна, вона змінює напрямок руху на протилежний.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке потік?
2. Чим відрізняється виконання багатопотокової програми від однопотокової?
3. Як створити і запустити потік?
4. Як завершити, призупинити і відновити виконання потоку?
5. Як визначити закінчення потоку?
6. Як передати аргумент потоку?
7. Пріоритети потоків. Як поміняти пріоритет потоку?
8. Який метод в новому потоці запускає метод Start() класу Thread?

9. Що робить метод Join() класу Thread?

Лабораторна робота № 2. Процеси

Мета роботи: вивчити принципи застосування пріоритетності запуску та роботи процесів на мові C#, навчитися створювати дочірній процес та очікувати завершення його роботи.

2.2.1. Створення процесів

Приклад 2.1. Створення процесів

Програма реалізує створення нового процесу за допомогою класу Process. Для його створення попередньо треба написати програму, яка буде викликатися як дочірній процес.

Лістинг 2.1. Дочірній процес

```
using System;
using System.Threading;

namespace l_2_1_p
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Дочірній процес створений: ");
            for (int i = 1; i <= 10; i++)
            {
                Console.Write(i + " ");
                Thread.Sleep(200);
            }
            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

Програма, котру буде викликати батьківський процес виводить строку «Дочірній процес створений.» на початку своєї роботи. Далі виводить числа від 1 до 10 включно з частотою 0.2 сек. Така частота реалізована за допомогою методу Sleep(), що знаходиться у класі Thread. Після завершення циклу програма очікує введення будь-якої клавіші. Після цього вона закривається.

Результат роботи програми наступний:

Дочірній процес створений: 1 2 3 4 5 6 7 8 9 10

Лістинг 2.1.2. Програма процесу, який створює процес на новій консолі

```
using System;
using System.Diagnostics;
using System.IO;

namespace l_2_1_m
{
    class Program
    {
        static void Main(string[] args)
        {
            Process newProc = new Process();

            newProc.StartInfo.FileName =
@"D:\Igor\Univer\Projects\Metod_SPO\l_2_1_p\l_2_1_p\bin\Debug\netcorea
pp3.1\l_2_1_p.exe";
            newProc.StartInfo.UseShellExecute = true;
            newProc.Start();
            Console.WriteLine("Новий процес створений.\n");
            newProc.WaitForExit();
            newProc.Close();

            Console.WriteLine("Новий процес закінчений.\n");

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці батьківського процесу створюється об'єкт класу Process, що відповідає за створення та управління процесами. Властивості FileName цього об’єкта присвоюється шлях до виконуючого файлу дочірнього процесу. Прапор UseShellExecute структури StartInfo встановлюється в true, щоб встановити створення нового вікна консолі для нового процесу. Запуск нового процесу починається після виклику методу Start() з класу Process.

Після виклику дочірнього процесу у вікно батьківського процесу виводиться рядок «Новий процес створений.», що означає, що новий процес створився.

Метод WaitForExit() слугує для очікування завершення роботи дочірнього процесу. Після його завершення процес закривається функцією Close(). У вікно батьківського процесу виводиться рядок «Новий процес закінчений.».

Результат роботи програми наступний:

Новий процес створений.

Новий процес закінчений.

2.2.2. Псевдодескриптори процесів

Приклад 2.2. Отримання псевдодескриптору процесу

Програма реалізує виведення псевдодескриптору процесу з використанням методу GetCurrentProcess().

```
using System;
using System.Diagnostics;

namespace l_2_2
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Process newProc;

            newProc = Process.GetCurrentProcess();

            Console.WriteLine(newProc);

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці цункції Main() створюється об'єкт класу Process, що відповідає за створення та управління процесами. Цьому об’єкту присвоюється значення головного процесу.

За допомогою об’єкта newProc у вікно виводиться значення дескриптору процесу.

Результат роботи програми наступний:

System.Diagnostics.Process (l_2_2)

2.2.3. Обслуговування потоків

Приклад 2.3. Обслуговування потоків. Отримання і зміна пріоритетності потоків.

Програма реалізує зміну пріоритету потоку з пріоритету за замовчуванням на пріоритет Lowest, а потім Highest.

```
using System;
using System.Threading;

namespace l_2_3
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Thread newThrd;

            newThrd = Thread.CurrentThread;

            Console.WriteLine("The priority level of the thread = " +
newThrd.Priority);

            newThrd.Priority = ThreadPriority.Lowest;

            Console.WriteLine("The priority level of the thread = " +
newThrd.Priority);

            newThrd.Priority = ThreadPriority.Highest;

            Console.WriteLine("The priority level of the thread = " +
newThrd.Priority);

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу Thread, що відповідає за створення та управління потоками. Цьому об’єкту присвоюється значення головного потоку.

Після цього у консоль виводиться значення приорітету потоку. Далі пріоритет потоку змінюється на низький за допомогою властивості Priority. Щоб змінити пріоритет на низький йому присвоюється значення Lowest. Та знову у консоль виводиться значення пріоритету потоку. Така ж процедура проводиться при зміні пріоритету на Highest.

Програма очікує свого завершення до введення будь-якої клавіші. Після цього вона закривається.

Результат роботи програми наступний:

The priority level of the thread = Normal

The priority level of the thread = Lowest

The priority level of the thread = Highest

ЗАВДАННЯ

1. Проаналізуйте та виконайте усі приклади розглянутих програм.
2. Виконайте усі приклади програм для 2, 3 та N процесів. Наочно покажіть паралельність виконання процесів.
3. Виконайте усі приклади програм для 2, 3 та N процесів в проектах C# Windows Forms.
4. Виконайте свій варіант відповідно до списку в підгрупи в проекті C# Windows Forms:

Варіант 1

Написати програму, яка буде дочірнім процесом. Вона повинна отримувати 3 параметра і виводити їх на екран, також постійно виводити на екран показання лічильника.

Запустити дочірній процес з передачею йому 3 параметрів. Після введення символу 'y' закрити дочірній процес.

Варіант 2

Написати програму, яка запускає в якості дочірнього процесу Блокнот (notepad). Після закриття Блокнота, з’являється повідомлення, що «Блокнот закритий».

Варіант 3

Написати програму, яка запускає в якості дочірнього процесу редактор тексту (winword). Після закриття редактора, видається повідомлення, що «Редактор закритий».

Варіант 4

Написати програму нескінченного лічильника, яка буде постійно виводити на екран показання лічильника. Запустити дочірній процес лічильник. Після введення символу 't' завершити дочірній процес.

Варіант 5

Написати програму, яка запускає N дочірніх процесів Блокнот (notepad). Після закриття і-го Блокнота, видає повідомлення, що «і-ий Блокнот закритий».

Варіант 6

Написати програму, яка запускає N дочірніх процесів редактор тексту (winword). Після закриття усіх редакторів, програма видає повідомлення «Всі процеси завершені».

Варіант 7

Написати програму, яка запускає в якості дочірнього процесу Блокнот (notepad) і передає йому в якості аргументу *name* назву файлу для редагування. Після закриття Блокнота, видає повідомлення, що «Блокнот закритий - файл *name* відредагований».

Варіант 8

Написати програму, яка запускає в якості дочірнього процесу програму роботи з таблицями (Excel) і передає йому в якості аргументу *name* назву файлу для редагування. Після закриття Excel, видає повідомлення, що «Excel закритий - файл *name* відредагований».

Варіант 9

Написати програму, яка запускає в якості дочірнього процесу редактор зображень (mspaint) і передає йому в якості аргументу *name* назву файлу із зображенням для редагування. Після закриття редактора, видає повідомлення, що «Редактор закритий - файл *name* відредагований».

Варіант 10

Написати програму, яка запускає N дочірніх процесів калькуляторів (calc). Після закриття і-го калькулятора, видає повідомлення, що «і-ий калькулятор закритий».

Варіант 11

Написати програму, яка запускає в якості дочірніх процесів 3 редактора зображень (mspaint), 5 блокнот (notepad), 2 редактора тексту (winword), 1 програму роботи з таблицями (Excel) і передає їм в якості аргументів name1, name2, ... файли для редагування. Після закриття кожного дочірнього процесу, видається повідомлення, що «Закрито «назва програми» - файл nameX відредагований».

Варіант 12

Написати програму, яка дозволяє запускати процеси, використовуючи для цього обрані на диску файли. Користувач може задавати ім'я файлу, що запускається, в командний рядок. Програма стежить за всіма запущеними нею процесами і виводить на вимогу користувача наступну інформацію: ім'я процесу, значення ідентифікатора процесу, час виконання процесу.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке процес?

2. Який клас і який простір імен використовується для створення процесу?
3. Як створити і запустити процес?
4. Як зупинити виконання процесу?
5. Як визначити завершення процесу?
6. Що запускає метод Start() класу Process?

Лабораторна робота № 3. Синхронізація

Мета роботи: ознайомитися з об’єктами синхронізації та навчитися використовувати їх.

2.3.1. Об'єкти синхронізації

Приклад 3.1. Використання lock для синхронізації роботи потоків

Програма реалізує виконання деякої роботи укладенної в структуру lock.

Лістинг 3.1 Приклад використання lock

```
using System;
using System.Threading;

namespace l_3_1
{

    class Program
    {
            static object lockOn = new object();

            static public void Run()
            {
                Console.WriteLine("Новий потік запущено.");


                lock (lockOn)
                {
                    Console.Write("Підрахунок: ");
                    for (int i = 0; i <= 10; i++)
                    {
                        Console.Write(i + " ");
                        Thread.Sleep(200);
                    }
                Console.WriteLine();
            }
            }
            static void Main(string[] args)
        {
            Thread newThrd = new Thread(Run);
            Thread newThrd1 = new Thread(Run);
```

```
            newThrd.Start();
            newThrd1.Start();
            newThrd.Join();
            newThrd1.Join();

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється два об'єкти класу Thread, що відповідає за створення та управління потоками. У конструктори класу Thread передається метод Run(). Запуск нових потоків починається після виклику методу Start() з класу Thread.

У створеному потоці метод Run() виконує цикл for(), в якому змінна count збільшується на одиницю кожні 0.2 сек. від 0 до 10 включно та виводиться на консоль. Така частота реалізована за допомогою методу Sleep(). На початку роботи методу виводиться рядок «Новий потік запущено.», що означає, що новий потік почав віконуватися.

Створюється об'єкт блокування lockOn. Цикл for вкладається в структуру lock з передачею у останню об'єкта lockOn. Виконання циклу for() в методі Run() починається виконуватися у одному потоці і блокується для виконання у другому. Після закінчення цього циклу в першому дочірньому потоці, він виконується у другому.

Метод Join() у основному потоці застосовується для очікування завершення нових потоків.

Результат роботи програми наступний:

Новий потік запущено.

Новий потік запущено.

Підрахунок: 0 1 2 3 4 5 6 7 8 9 10

Підрахунок: 0 1 2 3 4 5 6 7 8 9 10

Якщо не використовувати блокування за допомогою lock (закоментувати рядок lock (lockOn)), вивод значень змінної count двома дочірними потоками буде одночасно.

Результат роботи програми без блокування наступний:

Новий потік запущено.

Новий потік запущено.

Підрахунок: Підрахунок: 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10

2.3.2. М'ютекси

Приклад 3.2. Виконання процесів без використання м'ютекса

Програма створює новий процес за допомогою класу Process. Для його створення попередньо треба написати програму, яка буде викликатися як дочірній процес. Ця програма не використовує синхронізацію.

Лістинг 3.2. Дочірній процес без використання м'ютекса

```
using System;
using System.Threading;

namespace l_3_2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Дочірній процес.");

            for (int j = 10; j <= 20; j++)
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.Write(j + " ");
                    Thread.Sleep(100);
                }
                Console.WriteLine();
```

```
            }
            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

Програма, котру буде викликати батьківський процес виводить числа від 10 до 20 включно по 10 разів у кожному рядку з частотою 0.1 сек. Така частота реалізована за допомогою методу Sleep(), яка знаходиться у класі Thread. Після завершення циклу програма очікує до введення будь-якої клавіші та закривається.

Результат роботи програми наступний:

Дочірній процес.

10 10 10 10 10 10 10 10 10 10 10

11 11 11 11 11 11 11 11 11 11 11

12 12 12 12 12 12 12 12 12 12 12

13 13 13 13 13 13 13 13 13 13 13

14 14 14 14 14 14 14 14 14 14 14

15 15 15 15 15 15 15 15 15 15 15

16 16 16 16 16 16 16 16 16 16 16

17 17 17 17 17 17 17 17 17 17 17

18 18 18 18 18 18 18 18 18 18 18

19 19 19 19 19 19 19 19 19 19 19

20 20 20 20 20 20 20 20 20 20 20

У головному потоці батьківського процесу створюється об'єкт класу Process, що відповідає за створення та управління процесами. Властивості FileName цього об’єкта присвоюється шлях до виконуючого файлу дочірнього

процесу. Запуск нового процесу починається після виклику методу Start() з класу Process.

Батьківський процес виводить числа від 1 до 10 включно по 10 разів у кожному рядку з частотою 0.1 сек. Така частота реалізована за допомогою методу Sleep(), що знаходиться у класі Thread. Після завершення циклу програма очікує до введення будь-якої клавіші. Після цього вона закривається.

Листинг 3.2.1 Батьківський процес без використання м'ютекса

```
using System;
using System.Diagnostics;
using System.Threading;

namespace l_3_2_p
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Батьківський процес.");
            string appName =
@"D:\Igor\Univer\Projects\Metod_SPO\l_3_2\l_3_2\bin\Debug\netcoreapp3.
1\l_3_2.exe";
            Process newProc = new Process();
            newProc.StartInfo.FileName = appName;
            newProc.Start();

            for (int j = 0; j < 10; ++j)
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.Write(j + " ");
                    Thread.Sleep(100);
                }
                Console.WriteLine();
            }

            Console.ReadKey();

        }
    }
}
```

Результат роботи програми без використання синхронізації:

Батьківський процес.

0 Дочірній процес.

10 0 10 0 10 0 10 0 10 0 10 0 10 0 10 0 10 0 10

1 10 1

11 1 11 1 11 1 11 1 11 1 11 1 11 1 11 1 11

2 11 2 11 2

12 2 12 2 12 2 12 2 12 2 12 2 12 2 12

3 12 3 12 3 12 3

13 3 13 3 13 3 13 3 13 3 13 3 13

4 13 4 13 4 13 4 13 4

14 4 14 4 14 4 14 4 14 4 14

5 14 5 14 5 14 5 14 5 14 5

15 5 15 5 15 5 15 5 15

6 15 6 15 6 15 6 15 6 15 6 15 6

16 6 16 6 16 6 16

7 16 7 16 7 16 7 16 7 16 7 16 7 16 7

17 7 17 7 17

8 17 8 17 8 17 8 17 8 17 8 17 8 17 8 17 8

18 8 18

9 18 9 18 9 18 9 18 9 18 9 18 9 18 9 18 9 18 9

19

19 19 19 19 19 19 19 19 19 19

20 20 20 20 20 20 20 20 20 20 20

Ряди чисел з батьківського та дочірного процесів виводяться не по порядку.

Приклад 3.2-1: Робота процесів з використанням синхронізації м'ютексом

Програма повинна реалізувати створення нового процесу за допомогою класу Process. Для його створення попередньо треба написати програму з використанням синхронізації м'ютексом, яка буде викликатися як дочірній процес.

Листинг 3.2-1. Дочірній процес з використанням м'ютекса

```
using System;
using System.Threading;

namespace l_3_2_1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Дочірній процес.");

            if (args.Length < 1)
            {
                Console.WriteLine("No arguments.");
                Console.ReadKey();
                return;
            }

            string mtxName = args[0];
            bool init = false;
            Mutex newMutex = new Mutex(init, mtxName);

            for (int j = 10; j <= 20; j++)
            {
                newMutex.WaitOne();
                for (int i = 0; i < 10; i++)
                {
                    Console.Write(j + " ");
                    Thread.Sleep(100);
                }
                Console.WriteLine();
                newMutex.ReleaseMutex();
                Thread.Sleep(10);
            }

            Console.ReadKey();
```

```
        }
    }
}
```

Опис роботи програми.

Програма, котру буде викликати батьківський процес виводить числа від 10 до 20 включно по 10 разів у кожній строці з частотою 0.1 сек. Така частота реалізована за допомогою методу Sleep(), що знаходиться у класі Thread. Після завершення циклу програма очікує до введення будь-якої клавіші. Після цього вона закривається.

До виконання циклу проводиться перевірка на наявність переданого об’єкта м'ютекса у аргументах головній методу Main(). Якщо його немає, то виводиться рядок «No arguments.», а після натискання будь-якої клавіші програма закривається.

У циклі виводу чисел до виводу кожного рядку викликається метод WaitOne(), який очікує своєї черги на виконання. А після виводу кожного рядку викликається метод ReleaseMutex(), що встановлює м'ютекс у доступний режим для іншого процесу.

Листинг 3.2-1.1. Батьківський процес з використанням м'ютекса

```
using System;
using System.Diagnostics;
using System.Threading;

namespace l_3_2_1_p
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Батьківський процес.");

            bool initOwn = false;
            string mtxName = "MyMutex";
```

```
            string appName =
@"C:\Users\ponel\source\repos\l_3_2_1\l_3_2_1\bin\Debug\netcoreapp3.1\
l_3_2_1.exe";

            Mutex newMutex = new Mutex(initOwn, mtxName);

            Process newProc = new Process();
            newProc.StartInfo.FileName = appName;
            newProc.StartInfo.Arguments = mtxName;
            newProc.Start();

            for (int j = 0; j < 10; ++j)
            {
                newMutex.WaitOne();
                for (int i = 0; i < 10; i++)
                {
                    Console.Write(j + " ");
                    Thread.Sleep(100);
                }
                Console.WriteLine();
                newMutex.ReleaseMutex();
                Thread.Sleep(10);
            }

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці батьківського процесу створюється об'єкт класу Process, що відповідає за створення та управління процесами. Властивості FileName цього об’єкта присвоюється шлях до виконуючого файлу дочірнього процесу. Запуск нового процесу починається після виклику методу Start() з класу Process. Також створюється об'єкт класу Mutex, що відповідає за синхронізацію.

Батьківський процес виводить числа від 1 до 10 включно по 10 разів у кожному рядку з частотою 0.1 сек. Така частота реалізована за допомогою методу Sleep(), що знаходиться у класі Thread. Після завершення циклу програма очікує до введення будь-якої клавіші. Після цього вона закривається.

У циклі виводу чисел до написання кожної строки викликається метод WaitOne(), яка очікує своєї черги на виконання. А після кожної строки викликається ReleaseMutex(), що встановлює м'ютекс у доступний режим для іншого процесу.

Результат роботи програми з використанням м'ютекса:

Батьківський процес.
0 Дочірній процес.
0 0 0 0 0 0 0 0 0 0
10 10 10 10 10 10 10 10 10 10
1 1 1 1 1 1 1 1 1 1
11 11 11 11 11 11 11 11 11 11
2 2 2 2 2 2 2 2 2 2
12 12 12 12 12 12 12 12 12 12
3 3 3 3 3 3 3 3 3 3
13 13 13 13 13 13 13 13 13 13
4 4 4 4 4 4 4 4 4 4
14 14 14 14 14 14 14 14 14 14
5 5 5 5 5 5 5 5 5 5
15 15 15 15 15 15 15 15 15 15
6 6 6 6 6 6 6 6 6 6
16 16 16 16 16 16 16 16 16 16
7 7 7 7 7 7 7 7 7 7
17 17 17 17 17 17 17 17 17 17
8 8 8 8 8 8 8 8 8 8
18 18 18 18 18 18 18 18 18 18
9 9 9 9 9 9 9 9 9 9
19 19 19 19 19 19 19 19 19 19
20 20 20 20 20 20 20 20 20 20

Ряди чисел з батьківського та дочірного процесів виводяться по черзі.

2.3.3. Події

Приклад 3.3.

Програма повинна реалізувати синхронізацію між головним потоком та створеним потоком за допомогою подій з автоматичним скиданням.

```
using System;
using System.Threading;

namespace l_3_3
{
    class Program
    {
        static AutoResetEvent newEvnt = new AutoResetEvent(false);
        static void Run()
        {
            Console.WriteLine("Дочірний потік з використанням подій з
автоматичним скиданням.");

            for (int i = 0; i <= 10; i++)
            {
                Console.Write(i + " ");
                if (i == 4)
                {
                    newEvnt.Set();
                    Thread.Sleep(100);
                    newEvnt.WaitOne();
                    Console.WriteLine();
                }
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Головний потік.");

            Thread newThrd = new Thread(Run);

            newThrd.Start();

            newEvnt.WaitOne();

            Console.WriteLine("\nПоловина роботі виконана.");
```

```
            Console.WriteLine("Натисніть будь-яку клавішу, щоб
продовжити.");
            Console.ReadKey();

            newEvnt.Set();

            newThrd.Join();

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу MyThread та об'єкт класу Thread, що відповідає за створення та управління потоками. У конструктор класу Thread передається метод Run(). Запуск нового потоку починається після виклику методу Start() з класу Thread. Створюється об’єкт події з автоматичним скиданням newEvnt.

У створеному потоці метод Run() виконує рахунок від 0 до 10 включно. Якщо змінна дорівнює чотирьом, то за умовою об’єкт події переключається. У головному потоці очікується поки користувач не натисне будь-яку клавішу. Далі об’єкт знову переключиться і рахунок продовжиться.

Метод Join() використовується для очікування завершення нового потіку.

Результат роботи програми наступний.

Головний потік.
Дочірний потік з використанням подій з автоматичним скиданням.
0 1 2 3 4
Половина роботі виконана.
Натисніть будь-яку клавішу, щоб продовжити.

5 6 7 8 9 10

Приклад 3.4.

Програма повинна реалізувати синхронізацію між головним потоком та створеним потоком за допомогою подій з ручним скиданням.

```
using System;
using System.Threading;

namespace l_3_4
{
    class Program
    {
        static ManualResetEvent newEvnt = new ManualResetEvent(false);
        static void Run()
        {
            Console.WriteLine("Дочірний потік з використанням подій з
ручним скиданням.");

            for (int i = 0; i <= 10; i++)
            {
                Console.Write(i + " ");
                if (i == 4)
                {
                    newEvnt.Set();
                    Thread.Sleep(100);
                    newEvnt.WaitOne();
                    Console.WriteLine();
                }
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Головний потік.");

            Thread newThrd = new Thread(Run);

            newThrd.Start();

            newEvnt.WaitOne();
            newEvnt.Reset();

            Console.WriteLine("\nПоловина роботі виконана.");
            Console.WriteLine("Натисніть будь-яку клавішу, щоб
продовжити.");
            Console.ReadKey();

            newEvnt.Set();
```

```
            newThrd.Join();
            Console.ReadKey();
        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу MyThread та об'єкт класу Thread, що відповідає за створення та управління потоками. У конструктор класу Thread передається метод Run(). Запуск нового потоку починається після виклику методу Start() з класу Thread. Також створюється об’єкт події з ручним скиданням newEvnt.

У створеному потоці метод Run() виконує рахунок від 0 до 9 включно. Якщо змінна дорівнює чотирьом, то за умовою об’єкт події переключається. У головному потоці очікується поки користувач не натисне будь-яку клавішу. Далі об’єкт знову переключиться і рахунок продовжиться.

Метод Join() у основному потоці виконувала роль очікування, поки новий потік не завершить роботу.

Результат роботи програми наступний.

Головний потік.

Дочірний потік з використанням подій з ручним скиданням.

0 1 2 3 4

Половина роботі виконана.

Натисніть будь-яку клавішу, щоб продовжити.

5 6 7 8 9 10

2.3.4. Семафори

Приклад 3.5.

Несинхронізовані потоки. Програма створює новий потік та обидва потоки виконуються одночасно без синхронізації.

```
using System;
using System.Threading;

namespace l_3_5
{
    class Program
    {
        static int[] arr = new int[10];

        static void Run()
        {
            for (int i = 0; i < 10; i++)
            {
                arr[i] = i + 1;
                Thread.Sleep(70);
            }
        }

        static void Main(string[] args)
        {
            Console.Write("An initial state of the array: ");
            for (int i = 0; i < 10; i++)
                Console.Write(arr[i] + " ");
            Console.WriteLine();

            Thread newThrd = new Thread(Run);
            newThrd.Start();

            Console.Write("A modified state of the array: ");
            for (int i = 0; i < 10; i++)
            {
                Console.Write(arr[i] + " ");
                Thread.Sleep(70);
            }
            Console.WriteLine();

            newThrd.Join();

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу Thread, що відповідає за створення та управління потоками. У конструктор класу Thread передається метод Run(). Запуск нового потоку починається після виклику методу Start() з класу Thread. Виконується цикл виводу масиву arr у початковому стані (всі елементи дорівнюють нулю), а потім у стані під час роботи нового потоку (результат неможливо передбачити).

У створеному потоці метод Run() виконує рахунок від 0 до 10 включно. Рахунок чисел проводиться з частотою 0.07 сек. Така частота реалізована за допомогою методу Sleep().

Метод Join() у основному потоці очікує поки новий потік не завершить роботу.

Один з варіантів результату роботи програми:

An initial state of the array: 0 0 0 0 0 0 0 0 0 0
A modified state of the array: 0 2 3 4 0 6 0 8 9 10

Не усі значення масиву arr встигають прийняти нові значення.

Приклад 3.5-1.

Синхронізація роботи потоків з використанням семафора. Програма реалізує синхронізацію між головним потоком та створеним потоком за допомогою семафора.

```
using System;
using System.Threading;

namespace l_3_5_1
{
    class Program
    {
        static Semaphore newSem = new Semaphore(0, 1);
        static int[] arr = new int[10];

        static void Run()
        {
```

```
            for (int i = 0; i < 10; i++)
            {
                arr[i] = i + 1;
                newSem.Release();
                Thread.Sleep(500);
            }
        }

        static void Main(string[] args)
        {
            Console.Write("An initial state of the array: ");
            for (int i = 0; i < 10; i++)
                Console.Write(arr[i] + " ");
            Console.WriteLine();

            Thread newThrd = new Thread(Run);
            newThrd.Start();

            Console.Write("A modified state of the array: ");
            for (int i = 0; i < 10; i++)
            {
                newSem.WaitOne();
                Console.Write(arr[i] + " ");
            }
            Console.WriteLine();

            newThrd.Join();

            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

У головному потоці методу Main() створюється об'єкт класу Thread, що відповідає за створення та управління потоками. У конструктор класу Thread передається метод Run(). Запуск нового потоку починається після виклику методу Start() з класу Thread. Виконується цикл виводу масиву arr у початковому стані (всі елементи дорівнюють нулю), а потім у стані під час роботи нового потоку з синхронізацією.

У створеному потоці метод Run() виконує рахунок від 0 до 10 включно. Рахунок чисел проводиться з частотою 0.5 сек. Така частота реалізована за допомогою методу Sleep().

Метод Join() у основному потоці очікує поки новий потік не завершить роботу.

Для синхронізації потоків використується семафор. Метод WaitOne() у головному потоці очікує до того часу, поки не буде отримано семафор newSem. Таким чином, він блокує виконання головного дочірнього потоку поки вказаний семафор не надасть дозвіл на доступ до ресурсу. Після присвоєння елементу масиву arr нового значення у дочірньому потоці, семафор звільняється викликом метода Release(). Таким чином, вивід елемента масиву виконується тільки після присвоєння йому нового значення.

Результат роботи програми наступний:

An initial state of the array: 0 0 0 0 0 0 0 0 0 0
A modified state of the array: 1 2 3 4 5 6 7 8 9 10

ЗАВДАННЯ

1. Виконати усі приклади розглянутих програм.
2. Виконати усі приклади програм для 2, 3 и N процесів і/або потоків. Наглядно показати паралельність виконання і синхронізацію процесів і/або потоків.
3. Виконати усі приклади програм для 2, 3 и N процесів і/або потоків в проектах C# Windows Forms.
4. Виконати свій варіант по списку підгрупи в проекті C# Windows Forms:

Варіант 1

У програмі створити два потоки. Один з них - періодично читає системний час і заповнення глобальної структури (години, хвилини, секунди), другий – виводить цю структуру на екран. За допомогою м’ютекса організувати роздільний доступ потоків до структури даних.

Варіант 2

Написати програму, яка містить два потоки, кожен з яких керує рухом однієї з двох куль. Перша куля рухається горизонтально, друга - вертикально. Швидкість куль різна. При досягненні межі клієнтської області вікна, куля змінює напрямок руху на протилежний. За допомогою об'єктів синхронізації (семафорів або подій) реалізувати алгоритм руху куль без зіткнень.

Варіант 3

Створити багатопотокову програму, яка формує потоки трьох типів. Кожен з потоків запускається відповідним пунктом меню і захоплює відповідно 1, 2, 3 ресурси (максимальне число ресурсів може змінюватися користувачем). Кількість, вид потоків, а також їх стан виводиться на екран. Якщо число ресурсів не дозволяє працювати потоку, він знаходиться в стані очікування. Видалення потоків здійснюється через меню в порядку запуску (першим видаляється потік, запущений першим).

Варіант 4

Написати програму, яка створює 2 потоки, кожному з яких виділяється половина вікна програми. Алгоритм роботи потоку:

- якщо користувач клацнув мишкою в частині вікна, що належить потоку, пофарбувати її у випадковий колір;
- якщо перший потік змінив колір своєї частини вікна, другий потік повинен забарвити свою в протилежний колір.

Варіант 5

Написати програму, яка створює 2 потоки. Перший потік читає поточні координати миші і зберігає їх у пам'ять. Другий потік малює на екрані коло і читає значення координат миші з пам'яті. При наближенні курсора миші до кола, пересунути коло на деяку відстань у протилежний бік.

Варіант 6

Написати програму, яка створює 2 потоки. Перший потік з інтервалом в 1 секунду генерує чергове число послідовності Фібоначчі. Другий потік виводить це число і суму всіх чисел на екран. Для синхронізації потоків використовувати семафор.

Варіант 7

Розробити програму, яка ділить робочу область вікна на 4 рівні частини. Перша частина вікна забарвлюється в довільний колір під час кожного кліка користувача; у відповідь на зміну кольору в першій частині, друга, третя і четверта частини змінюють свій колір відповідно до числових значень складових кольору в 1 частині вікна. Приклад: в першій частині згенерувати колір RGB (126, 78, 221), отже, друга частина забарвиться в RGB (126, 0, 0), третя в RGB (0, 78, 0), четверта в RGB (0, 0, 221). Використовувати різні потоки і синхронізацію.

Варіант 8

Розробити програму, яка кожні n секунд генерує два випадкових числа, рахує їх суму, різницю, добуток і частку. Кожна із зазначених операцій організована як окремий потік. Перший потік генерує числа. Всі отримані значення виводяться на форму.

Варіант 9

Розробити програму, яка в довільному місці робочої області генерує прямокутник з випадковим кольором, у відповідь на цю подію, симетрично щодо центру робочої області малюється прямокутник з протилежним кольором. Генерація проводиться постійно окремими потоками, синхронізація за допомогою подій.

Варіант 10

Розробити програму, яка ділить робочу область вікна на 6 частин і кожні *t* секунд змінює колір першої частини. У відповідь на зміну кольору в першій частині, друга частина з затримкою в *x* секунд змінює свій колір на ідентичний кольору в першій частині. Кожна наступна частина залежить від попередньої (3 від 2, 4 від 3 і тощо), алгоритм дій для частин ідентичний алгоритму для 2 частини. Використовувати потоки і синхронізацію.

Варіант 11

Розробити програму, яка запускає новий потік при натисканні лівої кнопки миші. Потік починає виводити зростаючу числову послідовність в поточну позицію курсору миші. Кожен новий потік повинен дочекатися завершення підрахунку раніше організованого потоку. Кількість потоків обмежена (задається користувачем). Рахунок обмежений (задається користувачем).

Варіант 12

Написати програму, яка створює N-пар потоків. Перший потік кожної пари, з деякою своєю затримкою генерує число, а другий – рахує квадрат цього числа і виводить його на екран. Синхронізувати ці два потоки.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке синхронізація?
2. Навіщо і коли потрібна синхронізація? Наведіть приклади.
3. Які існують об’єкти синхронізації?
4. Що таке м’ютекс?
5. Що таке семафор?
6. Що таке події?
7. Методи класу Monitor, що використовуються для синхронізації.

8. Чим семафор відрізняється від події?

Лабораторна робота № 4. Відображення файлу у пам’яті

Мета роботи: ознайомитися з методами для роботи з постійно та непостійно зіставленими у памяті файлами, розглянути різницю у їх застосуванні.

2.4.1. Постійний зіставлений у пам’яті файл

Приклад 4.1. Постійний зіставлений у пам’яті файл з файлу на диску.

Програма створює зіставлений у пам’яті файл з існуючого файлу на диску, записує в нього масив чисел та зчитує дані з нього.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.MemoryMappedFiles;

namespace l_4_1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] mas = new int[] { 1, 32, 45, 11, -5, 0, 78, 52 };
            List<int> inputMas = new List<int>();
            Console.WriteLine("Масив.");
            for (int i = 0; i < mas.Length; i++)
                Console.Write(mas[i] + " ");

            Console.WriteLine("\n Зіставлений у пам’яті файл з файлу
на диску.");
            using (MemoryMappedFile mnf =
MemoryMappedFile.CreateFromFile("a1.dta", FileMode.OpenOrCreate,
"file",1))
            {
                using (MemoryMappedViewStream stream =
mnf.CreateViewStream())
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    foreach (int i in mas)
                        writer.Write(i);
```

```
                    writer.Close();
                    Console.WriteLine("\n Файл на диску створений та
закритий.");
                }
                using (MemoryMappedViewStream stream =
mnf.CreateViewStream())
                {
                    BinaryReader reader = new BinaryReader(stream);
                    for (int i = 0; i < mas.Length; i++)
                        inputMas.Add(reader.ReadInt32());
                }

                foreach (int i in inputMas)
                    Console.Write(i + " ");

                Console.ReadKey();

            }
        }
    }
}
```

Опис роботи програми.

У програмі визначається масив цілих чисел mas та його елементи виводяться у консоль. За допомогою методу CreateFromFile() створюється відображений у пам’яті файл MemoryMappedFile з існуючого файлу. В параметрах цього методу вказується шлях до існуючого файлу, режим доступу, ім’я зіставленого файлу та розмір. Далі створюється файловий потік, за допомогою методу CreateViewStream() класу MemoryMappedViewStream, на основі якого створюється об’єкт класу BinaryWriter. За допомогою об’єкту класу BinaryWriter у файл записується масив mas, після чого потік запису закривається. Потім відкривається новій файловий потік, на основі якого створюється об’єкт двійкового зчитувача BinaryReader. За допомогою методу ReadInt32() цього об’єкту послідовно зчитуються елементи масиву і виводяться у консоль.

Результат роботи програми наступний:

Масив.

1 32 45 11 -5 0 78 52

Зіставлений у пам'яті файл з файлу на диску.

Файл на диску створений та закритий.

1 32 45 11 -5 0 78 52

2.4.2. Непостійний зіставлений у пам’яті файл

Приклад 4.2.

Програма створює непостійний зіставлений у пам’яті файл, записує в нього масив чисел та зчитує його.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.MemoryMappedFiles;

namespace l_4_2
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] mas = new int[] { 1, 32, 45, 11, 0, 78, 52 };
            List<int> inputMas = new List<int>();
            Console.WriteLine("Масив.");
            for (int i = 0; i < mas.Length; i++)
                Console.Write(mas[i] + " ");

            Console.WriteLine("\n Непостійний зіставлений у пам’яті
файл.");

            using (MemoryMappedFile mnf =
MemoryMappedFile.CreateNew("n_file", 4096))
            {
                using (MemoryMappedViewStream stream =
mnf.CreateViewStream())
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    foreach (int i in mas)
                        writer.Write(i);
                    writer.Close();

                }
```

```
                using (MemoryMappedViewStream stream =
mnf.CreateViewStream())
                {
                    BinaryReader reader = new BinaryReader(stream);
                    for (int i = 0; i < mas.Length; i++)
                        inputMas.Add(reader.ReadInt32());
                }

                foreach (int i in inputMas)
                    Console.Write(i + " ");

                Console.ReadKey();

            }
        }
    }
}
```

Опис роботи програми.

У програмі визначається масив цілих чисел mas та його елементи виводяться у консоль. Метод CreateNew() створює зіставлений у пам'яті файл MemoryMappedFile, не зіставлений з існуючим файлом на диску. Далі формується файловий потік, за допомогою методу CreateViewStream() класу MemoryMappedViewStream, на основі якого створюється об'єкт класу BinaryWriter. За допомогою об'єкту класу BinaryWriter у файл записується масив цілих чисел, після чого потік запису закривається. Відкривається новій файловий потік, на основі якого створюється двійковий зчитувач BinaryReader. Метод ReadInt32() цього об'єкту послідовно зчитує елементи масиву і виводить їх у консоль.

Результат роботи програми наступний:

Масив.

1 32 45 11 0 78 52

Непостійний зіставлений у пам'яті файл.

1 32 45 11 0 78 52

Приклад 4.3.

Передача даних між двома процесами за допомогою непостійного зіставленого у пам’яті файлу.

Опис роботи програми.

У програмі головний процес А створює дочірній процес В. Батьківський процес створює непостійний зіставлений у пам’яті файл та записує в нього дані. Дочірній процес відкриває вже існуючий зіставлений у пам’яті файл та дописує в нього.

Процес А. Процес А створює м'ютекс і новий зіставлений файл у пам’яті MemoryMappedFile за допомогою методу CreateNew(). Після чого метод CreateViewStream() класу MemoryMappedFile створює потік відображення MemoryMappedViewStream. Далі формується об’єкт класу BinaryWriter, який здійснює запис булевої змінної «true» у зіставлений у пам’яті файл. Створюється об’єкт процесу procB і процес В запускається.

Процес А чекає, поки виконається процес В, після чого зчитує значення типу bool з зіставленого у пам’яті файлу за допомогою методу ReadBoolean() класу BinaryReader і виводить їх у консоль.

Процес В. В процесі В відкривається вже існуючий зіставлений в пам’яті файл за допомогою методу OpenExisting("procfile") класу MemoryMappedFile, параметром якого є назва цього файлу. Також методом OpenExisting("testmapmutex") відкривається вже існуючий м'ютекс. Далі створюється об’єкт класу MemoryMappedViewStream та об’єкт класу BinaryWriter, за допомогою якого в файл записується булева змінна «false». Процес В закінчує роботу.

Лістинг 4.3 Процес А

```
using System;
using System.Diagnostics;
using System.IO;
```

```
using System.IO.MemoryMappedFiles;
using System.Threading;

namespace l_4_3_a
{
    class Program
    {
        static void Main(string[] args)
        {

            using (MemoryMappedFile mmf =
MemoryMappedFile.CreateNew("procfile", 10000))
            {
                bool mutexCreated;
                Console.WriteLine("\t\tПроцес A.");
                Console.WriteLine("Створюється м'ютекс.");
                Mutex mutex = new Mutex(true, "testmapmutex", out
mutexCreated);

                Console.WriteLine("Непостійний зіставлений у пам’яті
файл.");

                using (MemoryMappedViewStream stream =
mmf.CreateViewStream())
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    Console.WriteLine("Запис даних.");
                    writer.Write(true);
                }
                mutex.ReleaseMutex();

                Process procB = new Process();
                procB.StartInfo.FileName =
@"D:\Igor\Univer\Projects\Metod_SPO\l_4_3_b\l_4_3_b\bin\Debug\netcorea
pp3.1\l_4_3_b.exe";
                procB.Start();

                Console.WriteLine("Запуск процеса B. Для продовження
натиснути ENTER.");
                Console.ReadLine();

                mutex.WaitOne();
                using (MemoryMappedViewStream stream =
mmf.CreateViewStream())
                {
                    BinaryReader reader = new BinaryReader(stream);
                    Console.WriteLine("Процес A передав: {0}",
reader.ReadBoolean());
                    Console.WriteLine("Процес B передав: {0}",
reader.ReadBoolean());
                }
```

```
                mutex.ReleaseMutex();

                Console.ReadKey();

            }

        }
    }
}
```

Лістинг 4.3.1 Процес B

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

namespace l_4_3_b
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                using (MemoryMappedFile mmf =
MemoryMappedFile.OpenExisting("procfile"))
                {
                    Console.WriteLine("\t\tПроцес B.");
                    Console.WriteLine("Відкривається м'ютекс.");
                    Mutex mutex = Mutex.OpenExisting("testmapmutex");
                    mutex.WaitOne();

                    Console.WriteLine("Непостійний зіставлений у
пам’яті файл.");
                    using (MemoryMappedViewStream stream =
mmf.CreateViewStream(1, 0))
                    {
                        BinaryWriter writer = new
BinaryWriter(stream);
                        Console.WriteLine("Запис даних.");
                        writer.Write(false);
                    }
                    mutex.ReleaseMutex();
                }
            }
            catch (FileNotFoundException)
            {
                Console.WriteLine("Непостійний зіставлений у пам’яті
файл не існує! Запустить процес A першим!");
```

```
                Console.ReadKey();

            }
        }
    }
}
```

Результат роботи програми наступний:

Процес A.

Створюється м'ютекс.

Непостійний зіставлений у пам'яті файл.

Запис даних.

Запуск процеса B. Для продовження натиснути ENTER.

Процес B.

Відкривається м'ютекс.

Непостійний зіставлений у пам'яті файл.

Запис даних.

Процес A передав: True

Процес B передав: False

ЗАВДАННЯ

1. Виконати усі приклади розглянутих програм.
2. Переробити приклади 4.1 і 4.2 так, щоб записувати і зчитувати рядок.
3. Переробити приклад 4.3 так, щоб записувати у файл фразу “Hello, World!” (кожен процес записує по слову).
4. Виконати свій варіант по списку підгрупи в проекті C# Windows Forms.

Варіант 1

Написати програму, яка запускає дочірній процес. При натисканні миші в області батьківського процесу, координати цього місця натискання передаються у дочірній процес і в цих же координатах у дочірньому процесі малюється прямокутник випадкового розміру і кольору.

Варіант 2

Написати програму з дочірнім процесом. У батьківському процесі заповнюється масив чисел. У дочірньому – робиться запит n-ого елементу цього масиву і виводиться на екран.

Варіант 3

Створити «Чат» з двох програм. В елемент керування TextBox вводиться речення і після натискання кнопки «надіслати», воно передається у другу програму, де виводиться на екран.

Варіант 4

Написати дві програми. У першій задаються два числа. У другій програмі вибирається арифметична операція, яку необхідно виконати над цими числами. Обчислений результат виводиться в першій програмі.

Варіант 5

Написати програму, в якій задається RGB-код кольору. У дочірньому процесі колір фону вікна змінюється на заданий колір.

Варіант 6

Написати дві програми. В одній програмі, після натискання кнопки, фону вікна задається випадковий колір. У другу програму виводиться RGB-код цього кольору.

Варіант 7

Написати дві програми. У першій програмі горизонтально зліва направо рухається коло. По досягненню колом правого краю вікна, в другій програмі починає рухатися зліва направо коло того ж розміру і на тієї же «висоті» вікна.

Варіант 8

Написати програму. За допомогою кнопок задається напрямок пересування квадрата в дочірньому процесі (вліво, вправо, вгору, вниз). У TextBox задається відстань, на яку необхідно пересунути квадрат.

Варіант 9

Написати програму. У батьківському процесі кнопкою задається фон вікна випадкового коліру. Після чого в дочірньому процесі фону задається колір, протилежний кольору фону вікна батьківського процесу.

Варіант 10

Написати програму: на формі дочірнього процесу є 5 кнопок, після натискання на які в файл записується рядок «Була натиснута N-а кнопка» (файл не переписується, а дописується); в батьківському процесі виводиться зміст файлу.

Варіант 11

Написати дві програми: в першій - вводиться рядок, друга рахує кількість букв, чисел і пробілів в цьому рядку.

Варіант 12

Написати програму, в якій задаються координати і розмір кола, а в дочірньому процесі в заданому місці малюється це коло.

КОНТРОЛЬНІ ПИТАННЯ

1. Коротко описати концепцію механізму відображення файлів у пам’яті.
2. Назвати два типи зіставлених у пам’яті файлів і їх відмінність.
3. Який метод використовується для отримання об’єкта, який представляє постійний зіставлений у пам’яті файл?
4. Який метод використовується для отримання об’єкта, який представляє непостійний зіставлений у пам’яті файл?
5. Який метод використовується для отримання об’єкта представлення для послідовного і довільного доступу до зіставленого в пам’яті файлу?

Лабораторна робота № 5. Канали передачі даних

Мета роботи: розглянути застосування анонімного та іменованого каналів, розглянути різницю при їх використанні.

2.5.1. Анонімний канал

Приклад 5.1.

Між двома процесами створюється анонімний канал. Клієнтський процес передає серверному процесу випадкові числа.

Опис роботи програми.

Сервер. Створюється об’єкт серверної частини анонімного каналу AnonymousPipeServerStream. Конструктор AnonymousPipeServerStream отримує параметри: напрямок передачі (In, Out, InOut) та HandleInheritability.Inheritable, що робить можливим успадковування дескриптора каналу.

Метод StartClient запускає клієнтський процес. При цьому в інформації запуску ProcessStartInfo властивість UseShellExecute = false, що означає, що оболонка ОС не буде використовуватися для запуску процесу, він буде запущений в рамках батьківського. Інакше анонімний канал не буде працювати. Крім того, метод StartClient приймає параметром рядок – дескриптор клієнта. Він отримується з методу AnonymousPipeServerStream. GetClientHandleAsString(). Дескриптор передається в якості аргументу в клієнтський процес.

Спочатку сервер очікує на синхронізуюче повідомлення, яке містить рядок «SYNC». Далі починається зчитування переданої клієнтом інформації до моменту, поки клієнтський процес не буде припинено.

Клієнт. Створює об’єкт AnonymousPipeClientStream. В конструктор цього класу передається отриманий з батьківського процесу дескриптор дочірнього процесу. На його основі створюється клієнтська частина каналу.

Клієнт надсилає повідомлення “SYNC” до сервера та очікує, поки сервер прочитає дане повідомлення (метод WaitForPipeDrain() блокує потік, поки надіслане повідомлення не буде повністю зчитане іншою стороною каналу). Далі генеруються випадкові числа та надсилаються до серверу, поки не буде введено літеру ‘q’.

Лістинг 5.1. Серверний процес

```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;

namespace l_5_1_s
{
    class Program
    {
        static Process client;
        const string EXE_PATH =
@"D:\Igor\Univer\Projects\Metod_SPO\l_5_1_c\l_5_1_c\bin\Debug\netcorea
pp3.1\l_5_1_c.exe";
        static void Main(string[] args)
        {
            Console.WriteLine("\tPROCESS SERVER.");


            using (AnonymousPipeServerStream pipeServer = new
AnonymousPipeServerStream(PipeDirection.In,
                    HandleInheritability.Inheritable))
                {
                    StartClient(pipeServer.GetClientHandleAsString());
                    pipeServer.DisposeLocalCopyOfClientHandle();

                using (StreamReader sr = new StreamReader(pipeServer))
                {
                    string temp;

                     // Wait for 'sync message' from the client.
```

```
                        do
                        {
                            Console.WriteLine("\t[SERVER] Wait for
sync...");
                            temp = sr.ReadLine();
                            if (temp == null)
                                break;
                        }
                        while (!temp.StartsWith("\tSYNC"));

                    Console.WriteLine("\t[SERVER] Client has
connected");

                        do
                        {
                            temp = sr.ReadLine();

Console.WriteLine(string.IsNullOrEmpty(temp) ? "\t[SERVER] No response
from client" :
                                    "\t[SERVER] Received: " + temp);
                        }
                        while (!client.HasExited);

                }
                    client.Close();
                    Console.WriteLine("\t[SERVER] Client quit.
Server terminating.");
                    Console.ReadKey();
                }
            }

            static void StartClient(string clientHandle)
            {
                ProcessStartInfo info = new
ProcessStartInfo(EXE_PATH);
                info.Arguments = clientHandle;
                info.UseShellExecute = false;
                client = Process.Start(info);
            }


    }
}
```

Лістинг 5.1.1. Клієнтський процес

```
using System;
using System.IO;
using System.IO.Pipes;
```

```
namespace l_5_1_c
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("PROCESS CLIENT.");
            Random rand = new Random();
            if (args.Length > 0)
            {
                using (PipeStream pipeClient = new
AnonymousPipeClientStream(PipeDirection.Out, args[0]))
                {
                    using (StreamWriter sw = new
StreamWriter(pipeClient))
                    {
                        sw.AutoFlush = true;

          // Send a 'sync message' and wait for client to receive it.

                        sw.WriteLine("SYNC");
                        pipeClient.WaitForPipeDrain();

                        do
                        {
                            int num = rand.Next(0, 100);
                            sw.WriteLine(num);
                            Console.WriteLine("[CLIENT] Sended: " +
num);
                            Console.Write("[CLIENT] Enter 'q' to exit
or any key to continue: ");

                        }
                        while (Console.ReadLine() != "q");
                    }
                }
            }

    }
}
```

Результат роботи програми наступний:

```
        PROCESS SERVER.
        [SERVER] Wait for sync...
Процес клієнт.
[CLIENT] Sended: 2
```

```
[CLIENT] Enter 'q' to exit or any key to continue:      [SERVER] Wait
for sync...
        [SERVER] Wait for sync...

[CLIENT] Sended: 56
[CLIENT] Enter 'q' to exit or any key to continue:      [SERVER] Wait
for sync...

[CLIENT] Sended: 43
        [SERVER] Wait for sync...
[CLIENT] Enter 'q' to exit or any key to continue:
[CLIENT] Sended: 39
[CLIENT] Enter 'q' to exit or any key to continue:      [SERVER] Wait
for sync...

[CLIENT] Sended: 10
[CLIENT] Enter 'q' to exit or any key to continue:      [SERVER] Wait
for sync...
q
        [SERVER] Client has connected
        [SERVER] No response from client
        [SERVER] Client quit.       Server terminating.
```

2.5.2. Іменований канал

Приклад 5.2.

Між двома процесами створюється іменований канал. Процеси обмінюються числами.

Опис роботи програми.

Сервер. Створюється об’єкт NamedPipeServerStream, який відповідає за серверну частину іменованого потоку. В конструкторі задається ім’я, напрямок передачі даних (InOut) та кількість можливих підключених клієнтів (1). Метод WaitForConnection() блокує виконання потоку до того моменту, поки не під’єднається клієнт. Після того, як клієнт під’єднається до сервера, починається передача даних методом WriteByte() – передається один байт. Потім очікує на відповідь від клієнта (метод WaitForPipeDrain() блокує потік,

поки надіслане повідомлення не буде повністю зчитане іншою стороною каналу) та зчитує її ReadByte().

Клієнт. Створюється об'єкт NamedPipeClientStream. Конструктор приймає адресу сервера (в даному випадку «.» означає локальний хост), ім'я (має співпадати з ім'ям серверної частини каналу), напрямок передачі даних. Аналогічно з сервером, клієнт спочатку приймає число, потім надсилає своє число серверу.

Лістинг 5.2 Серверний процес

```
using System;
using System.IO;
using System.IO.Pipes;

namespace l_5_2_s
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Process Server.");
            NamedPipeServerStream serverPipe = new
NamedPipeServerStream("myPipe", PipeDirection.InOut, 1);
            Console.WriteLine("Waiting for client...");
            serverPipe.WaitForConnection();
            Console.WriteLine("Client has connected...");
            Console.WriteLine("Press any key to continue");
            Console.ReadKey();

            try
            {
                //send

                byte sendData = 48;
                serverPipe.WriteByte(sendData);
                Console.WriteLine("Sended: " + sendData);

                //receive

                int dataReceive = serverPipe.ReadByte();
                Console.WriteLine("Received: " + dataReceive);
                serverPipe.Close();
            }
            catch (IOException ex)
```

```
            {
                Console.WriteLine("Error: " + ex.Message);
            }

            Console.ReadKey();

        }
    }
}
```

Результат роботи серверного процесу наступний:

Process Server.

Waiting for client...

Client has connected...

Press any key to continue

Sended: 48

Received: 24

Лістинг 5.2.1. Клієнтський процес

```
using System;
using System.IO.Pipes;
using System.Security.Principal;

namespace l_5_2_c
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Process client.");

           //connect

            NamedPipeClientStream pipe = new
NamedPipeClientStream(".", "myPipe",
                PipeDirection.InOut, PipeOptions.None,
TokenImpersonationLevel.None);
            pipe.Connect();
            Console.WriteLine("Connected with server");

            //read data
```

```
            int dataReceive = pipe.ReadByte();
            Console.WriteLine("Client receive: " +
dataReceive.ToString());

            //write data

            byte dataSend = 24;
            pipe.WriteByte(dataSend);
            Console.WriteLine("Client send: " + dataSend.ToString());

            //close pipe

            pipe.Close();

            Console.ReadKey();

        }
    }
}
```

Результат роботи клієнтського процесу наступний:

Process client.

Connected with server

Client receive: 48

Client send: 24

Приклад 5.3.

Процес-сервер запускає кілька потоків-клієнтів, створює іменовані канали, через які сервер та клієнти обмінюються повідомленнями.

Лістинг 5.3 Серверний процес

```
using System;
using System.Diagnostics;
using System.IO.Pipes;
using System.Security.Principal;
using System.Text;
using System.Threading;
```

```
namespace l_5_3_s
{
    public class Server
    {
        const string EXE_PATH =
@"C:\Users\ponel\source\repos\l_5_3_c\l_5_3_c\bin\Debug\netcoreapp3.1\
l_5_3_c.exe";
        Thread[] serverThreads;
        int numClients;
        byte[] byteMessage;

        public Server(int numClients)
        {
            this.numClients = numClients;
            serverThreads = new Thread[numClients];
        }

        public void Start(string message)
        {
            byteMessage = Encoding.ASCII.GetBytes(message);

            for (int i = 0; i < numClients; i++)
            {
                serverThreads[i] = new Thread(ServerThread);
                serverThreads[i].Start();
            }
        }

        private void ServerThread()
        {
            try
            {
                int threadID = Thread.CurrentThread.ManagedThreadId;
                NamedPipeServerStream serverPipe = new
NamedPipeServerStream("pipe" + threadID, PipeDirection.InOut, 1);

                //client creating

                ProcessStartInfo info = new ProcessStartInfo
                {
                    FileName = EXE_PATH,
                    Arguments = threadID.ToString()
                };
                Process client = Process.Start(info);
                Console.WriteLine("[{0}] Waiting for client...",
threadID);


                serverPipe.WaitForConnection();

                //sending
```

```
                serverPipe.Write(byteMessage, 0, byteMessage.Length);

                // чекаємо поки клієнт отримає повідомлення

                serverPipe.WaitForPipeDrain();
                Console.WriteLine("[{0}] Client has received the
message", threadID);

                //receive

                byte[] reqBytes = new byte[100];
                serverPipe.Read(reqBytes, 0, 100);
                Console.WriteLine("[{0}] Got request: {1}", threadID,
Encoding.ASCII.GetString(reqBytes));

                client.WaitForExit();
                Console.WriteLine("[{0}] Client has exited",
threadID);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error: " + ex.Message);
                return;
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Process Server.");
            Console.Write("Count of processes: ");
            int count = Convert.ToInt32(Console.ReadLine());

           //create server

            Server server = new Server(count);

            Console.Write("Enter message: ");

            //starting server

            server.Start(Console.ReadLine());
            Console.ReadKey();

        }
    }
}
```

Лістинг 5.3.1 Клієнтський процес

```
using System;
using System.Diagnostics;
using System.IO.Pipes;
using System.Security.Principal;
using System.Text;

namespace l_5_3_c
{
    public class Client
    {
        NamedPipeClientStream clientPipe;
        int clientID;

        public Client(string serverID)
        {
            clientID = Process.GetCurrentProcess().Id;
            clientPipe = new NamedPipeClientStream(".", "pipe" +
serverID,
                PipeDirection.InOut, PipeOptions.None,
TokenImpersonationLevel.None);
        }

        public void Start()
        {
            try
            {
                //receive

                clientPipe.Connect();
                byte[] inBytes = new byte[100];
                clientPipe.Read(inBytes, 0, 100);
                string inStr = Encoding.ASCII.GetString(inBytes);
                Console.WriteLine("\t Received from server: {0}",
inStr);

                //send

                Console.WriteLine("\t Sending message to server...");
                byte[] outBytes = new byte[100];
                outBytes = Encoding.ASCII.GetBytes("\t Hello, server!
I'm your client #" + clientID);
                clientPipe.Write(outBytes, 0, outBytes.Length);

                //чекаємо поки сервер отримає повідомлення

                clientPipe.WaitForPipeDrain();
                Console.WriteLine("\t The message has been received by
server");
```

```
            }
            catch (Exception ex)
            {
                Console.WriteLine("\t Error: " + ex.Message);
                return;
            }
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\t Process client.");
            Client client = new Client(args[0]);
            client.Start();
            Console.ReadKey();

        }
    }
}
```

Опис роботи програми.

Серверний процес.

Клас Server відповідає за серверну частину каналу. При створенні об’єкту сервера в конструкторі вказується кількість клієнтів. Створюється окремий потік для взаємодії з клієнтом. Робота починається з методу Start, який отримує параметром рядок, що буде надіслано до всіх клієнтів. В цьому методі відбувається переведення рядку в масив байтів та запуск потоків, які виконують метод ServerThread.

В кожному потоці ServerThread створюється об’єкт серверної частини каналу NamedPipeServerStream. Ім’я задається на основі номеру потоку («pipe» + ID).

Формується об’єкт ProcessStartInfo, за допомогою якого будуть запущені клієнтські процеси: вказується шлях до .exe файлу та в аргументах передається ID процесу.

Далі потік надсилає клієнту повідомлення, очікує на відповідь від клієнта. Після отримання відповіді виводить її на екран та чекає, поки клієнтський процес буде завершено, про що теж буде виведено повідомлення.

Клієнтський процес.

В конструкторі класу Client отримується рядок з ідентифікатором потоку в сервері, який здійснює роботу з клієнтом. Створюється об’єкт клієнтської частини каналу (ім’я задається на основі отриманого ID потоку). Таким чином кожен клієнт та серверний потік мають між собою окремий двосторонній канал. Робота клієнта починається з методу Start. Клієнт під’єднується до сервера, отримує повідомлення та надсилає відповідь, яка містить ідентифікатор процесу клієнта.

Результат роботи програми наступний:

```
Process Server.
Count of processes: 3
Enter message: PROCESS
[5] Waiting for client...
[6] Waiting for client...
[7] Waiting for client...
        Process client.
        Process client.
        Process client.
[6] Client has received the message
[5] Client has received the message
[7] Client has received the message
        Received from server: PROCESS
        Sending message to server...
        Received from server: PROCESS
        Sending message to server...
        Received from server: PROCESS
        Sending message to server...
```

```
        The message has been received by server
        The message has been received by server
        The message has been received by server
[5] Got request:    Hello, server! I'm your client #21392
[7] Got request:    Hello, server! I'm your client #13536
[6] Got request:    Hello, server! I'm your client #21320
```

ЗАВДАННЯ

1. Виконати усі приклади програм.
2. Виконати усі приклади програм в проектах C# Windows Forms.
3. Написати програму. Використовуючи анонімний канал, сервер надсилає клієнту число і рядок. Клієнт повинен вивести рядок, який він отримує, задану кількість разів.
4. Виконати свій варіант по списку підгрупи в проекті C# Windows Forms, використовуючи іменовані канали:

Варіант 1

Написати програму: у вікні серверного процесу після натискання кнопки фону задається випадковий колір. Після цього фон вікна клієнтського процесу повинен мати протилежний колір.

Варіант 2

Написати програму: сервер відправляє двом клієнтам рядок. Перший клієнт рахує кількість букв і цифр. Другий клієнт рахує кількість слів. Обидва клієнта повертають результат серверу, який їх виводить.

Варіант 3

Написати програму. У вікні серверного процесу задаються два числа. У клієнтській програмі вибирається арифметична операція, яку необхідно виконати над цими числами. Обчислений результат виводиться на сервері.

Варіант 4

Написати програму. За допомогою кнопок у вікні серверного процесу задається напрямок пересування квадрата у вікні клієнтського процесу. У TextBox задається відстань, на яку необхідно пересунути квадрат.

Варіант 5

Написати програму. У вікні серверного процесу по таймеру фону задається випадковий колір. У вікні клієнтського процесу виводиться RGB-код цього кольору.

Варіант 6

Написати програму. У вікні серверного процесу задаються координати і розмір кола, а в клієнтському вікні в заданому місці малюється дане коло.

Варіант 7

Написати програму. На формі клієнтського процесу є 5 кнопок. При натисканні на якусь із цих кнопок, в серверному процесі виводиться рядок «Була натиснута n-а кнопка».

Варіант 8

Написати програму. Серверний процес заповнює масив чисел. Клієнтський процес робить запит n-го елементу цього масиву і виводить його на екран.

Варіант 9

Написати програму. У вікні серверного та клієнтського процесів є поле для введення тексту. Коли клієнт вводить текст, він автоматично з'являється в серверному вікні.

Варіант 10

Написати програму. При натисканні миші в області вікна серверного процесу, у вікні клієнтського на цих же координатах малюється прямокутник випадкового розміру і кольору.

Варіант 11

Написати програму. По вікну серверного процесу горизонтально зліва направо рухається коло. По досягненню колом краю вікна, воно починає рухатися у вікні клієнтського процесу в тому ж напрямку і на тій же «висоті».

Варіант 12

Написати програму. У серверному процесі задається RGB-код кольору. У клієнтському процесі колір фону вікна змінюється на заданий колір.

КОНТРОЛЬНІ ПИТАННЯ

1. В чому полягає різниця між анонімними та іменованими каналами?
2. Об’єкти яких класів використовуються для організації анонімних каналів?
3. Яким чином організовується запис і читання інформації в анонімному каналі?
4. Де і для чого використовуються іменовані канали?
5. Яку адресу необхідно вказати при створенні з’єднання за допомогою іменованих каналів?

Лабораторна робота № 6. Бібліотека розпаралелювання задач TPL

Мета роботи: ознайомитися з особливостями паралельного програмування та бібліотекою TPL.

2.6.1 Клас Parallel

Приклад 6.1. В програмі два методи Meth() и Meth2() виконуються паралельно за допомогою виклику методу Invoke().

```
using System;
using System.Threading.Tasks;
using System.Threading;

namespace paral_invoke
{
    class Program
    {

        // Метод виконується як задача.

        static void Meth()
        {
            Console.WriteLine("Meth запущений");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(500);
     Console.WriteLine("В методі Meth підрахунок = " + count);
            }
            Console.WriteLine("Meth завершений");
        }

        // Метод виконується як задача.

        static void Meth2()
        {
            Console.WriteLine("\tMeth2 запущений");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(500);
  Console.WriteLine("\tВ методі Meth2, підрахунок = " + count);
            }
            Console.WriteLine("\tMeth2 завершений");
        }
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("\t\tОсновний потік запущений.");

                    // Виконати паралельно два іменованих методи.

            Parallel.Invoke(Meth, Meth2);

            Console.WriteLine("\t\tОсновний потік завершено.");
            Console.ReadLine();
        }
    }
}
```

Опис роботи програми.

Виконання методу Main() припиняється до повернення з методу Invoke(). Отже, метод Main(), на відміну від методів Meth() і Meth2(), не виконується паралельно. Тому, якщо потрібно, щоб виконання викликаючого потоку тривало не можна застосовувати метод Invoke().

У цьому прикладі використовуються іменовані методи, але для виклику методу Invoke() ця умова не є обов'язковою.

Виконання цієї програми може привести до наступного результату.

```
                Основний потік запущений.
Meth() запущений
        Meth2() запущений
В методі Meth() підрахунок = 0
        В методі Meth2() підрахунок = 0
В методі Meth() підрахунок = 1
        В методі Meth2() підрахунок = 1
В методі Meth() підрахунок = 2
        В методі Meth2() підрахунок = 2
В методі Meth() підрахунок = 3
        У методі Meth2() підрахунок = 3
```

В методі Meth() підрахунок = 4

Meth() завершено

У методі Meth2() підрахунок = 4

Meth2() завершено

Основний потік завершено.

Приклад 6.2. Використовується попередня програма, де в якості аргументів у виклиці метода Invoke() застосовується **лямбда-вираз**.

```
using System;
using System.Threading.Tasks;
using System.Threading;

namespace paral_invoke2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\t\tОсновний потік запущений.");

 // Виконання двох анонімних методів, які вказуються в лямбда-виразах

            Parallel.Invoke(() => {
                Console.WriteLine("Вираз #1 запущений");
                for (int count = 0; count < 5; count++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine("У виразі #1 підрахунок = " +
count);
                }
                Console.WriteLine("Вираз  #1 завершений");
            },
            () => {
                Console.WriteLine("\tВираз  #2 запущений");
                for (int count = 0; count < 5; count++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine("\tУ виразі #2 підрахунок = " +
count);
                }
                Console.WriteLine("\tВираз  #2 завершений");
            });
            Console.WriteLine("\t\tОсновний потік завершений.");
            Console.ReadLine();
```

```
        }
    }
}
```

Ця програма отримує результат, який схожий на результат виконання прикладу 6.1.

2.6.2 Застосування методу For()

Приклад 6.3. Розпаралелювання циклу методом For().

Створюється дуже великий масив цілих arr. Викликається метод For(), якому в якості тіла циклу передається метод Calculations(), який виконує деяку обробку цього масиву. Виконувана операція повинна бути нетривіальною, щоб паралелізм даних приніс якийсь позитивний ефект. В іншому випадку послідовне виконання циклу може завершитися швидше.

```
using System;
using System.Threading.Tasks;

namespace paral_for
{
    class Program
    {
        static int[] arr;

    // Метод, використовуєься в якості тіла паралельно виконуваного
    //циклу.  Оператори цього циклу просто використовують час ЦП
    //для  демонстрації.

        static void Calculations(int i)
        {
            if ((0<arr[i]) && (arr[i] < 999999))
                    arr[i] = 1;
            if ((999999<arr[i]) && (arr[i] < 1999999))
                    arr[i] = 10;
            if ((1999999<arr[i]) && (arr[i] < 123999999))
                    arr[i] = 100;
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Основний поток запущений.");
            arr = new int[100000000];
```

```
                // Ініціалізація даних у звичайному циклі for.

            for (int i = 0; i < arr.Length; i++)
                      arr[i] = i;

                      // Розпаралелювання циклу методом For().

            Parallel.For(0, arr.Length, Calculations);

            Console.WriteLine("Основний потік  завершений.");
            Console.ReadLine();
        }
    }
}
```

Опис роботи програми.

Ця програма складається з двох циклів. У першому, стандартному, циклі for ініціалізується масив arr. А в другому циклі, що виконується паралельно методом For(), над кожним елементом масиву arr виконується перетворення. Це перетворення носить довільний характер і вибрано лише для цілей демонстрації. Метод For() автоматично розбиває виклики методу Calculations() на частини для паралельної обробки окремих порцій даних, які зберігаються в масиві. Отже, якщо запустити дану програму на комп'ютері з двома або більше доступними процесорами, то цикл перетворення даних в масиві може бути виконано методом For() паралельно.

Слід мати на увазі, що далеко не усі цикли можуть виконуватися ефективно, коли вони розпаралелені. Як правило, дрібні цикли, а також цикли, що складаються з дуже простих операцій, виконуються швидше послідовним способом, ніж паралельним. Саме тому цикл for ініціалізації масиву даних не розпаралелюється методом For() в розглянутій програмі.

Приклад 6.4. Демонстрація відмінності в часі для послідовного та паралельного виконання циклу for.

Розпаралелювання дрібних і дуже простих циклів може виявитися неефективним тому, що час, потрібний для організації паралельних задач, а

також час, що витрачається на перемикання контексту, перевищує час, який економиться завдяки паралелізму. На підтвердження цього факту в наведеному нижче прикладі програми створюються **послідовний і паралельний варіанти циклу for**, а для порівняння на екран виводиться **час виконання кожного з них.**

```
using System;
using System.Threading.Tasks;
using System.Diagnostics;

namespace paral_for2
{
      // Продемонструвати відмінності в часі для послідовного та
      // паралельного виконання циклу for.

    class Program
    {
        static int[] arr;

        // Метод – це тіло паралельно виконуємого циклу.
        // Оператори цього циклу просто використовують час ЦП
        //для  демонстрації.

        static void Calculations(int i)
        {
            if ((0<arr[i]) && (arr[i] < 999999))
                    arr[i] = 1;
            if ((999999<arr[i]) && (arr[i] < 1999999))
                    arr[i] = 10;
            if ((1999999<arr[i]) && (arr[i] < 123999999))
                    arr[i] = 100;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Основний потік запущено.");

            // Створити екземпляр обєкта типу Stopwatch
            // для зберігання часу виконання циклу.

            Stopwatch sw = new Stopwatch();
            arr = new int[100000000];

                // Ініціалізація даних

            sw.Start();
```

```
            // Паралельний варіант ініціалізації массиву в циклі.

            Parallel.For(0, arr.Length, (i) => arr[i] = i);
            sw.Stop();
            Console.WriteLine("Паралельно виконуємий цикл
ініціалізації: " + "{0} секунд", sw.Elapsed.TotalSeconds);
            sw.Reset();
            sw.Start();

            // Послідовний варіант ініціалізації массиву в циклі.

            for (int i = 0; i < arr.Length; i++)
                  arr[i] = i;
            sw.Stop();
            Console.WriteLine("Послідовно виконуємий цикл
ініціалізації: " + "{0} секунд", sw.Elapsed.TotalSeconds);
            Console.WriteLine();

            // Виконати перетворення.

            sw.Start();

            // Паралельний варіант перетворення даних у циклі.

            Parallel.For(0, arr.Length, Calculations);
            sw.Stop();
            Console.WriteLine("Паралельно виконуємий цикл перетворення:
" + "{0} секунд", sw.Elapsed.TotalSeconds);
            sw.Reset();
            sw.Start();

            // Послідовний варіант перетворення даних у циклі.

            for (int i = 0; i < arr.Length; i++)
                  Calculations(i);
            sw.Stop();
            Console.WriteLine("Послідовно виконуємий цикл перетворення:
" + "{0} секунд", sw.Elapsed.TotalSeconds);
            Console.WriteLine("Основний потік завершено.");
            Console.ReadLine();
        }
    }
}
```

При виконанні цієї програми на двоядерному комп'ютері отримаємо наступний результат.

Основний потік запущений.

Паралельно виконується цикл ініціалізації: 1.0537757 секунд
Послідовно виконується цикл ініціалізації: 0.3457628 секунд
Паралельно виконується цикл перетворення: 4.2246675 секунд.
Послідовно виконується цикл перетворення: 5.3849959 секунд.
Основний потік завершено.

Паралельний варіант циклу ініціалізації масиву даних виконується приблизно в три рази повільніше, ніж послідовний. Справа у тому, що в даному випадку на операцію присвоювання витрачається так мало часу, що витрати на додаткове розпаралелювання перевищують економію, яку воно дає.

Паралельний варіант циклу перетворення даних виконується швидше, ніж послідовний. В даному випадку економія від розпаралелювання з лишком відшкодовує витрати на його додаткову організацію.

Для обчислення часу виконання циклу застосовується клас Stopwatch. Створюється екземпляр його об'єкта, викликається метод Start(), який починає відлік часу, а потім - метод Stop(), який завершує відлік часу. А за допомогою методу Reset() відлік часу скидається в початковий стан. Тривалість виконання отримується за допомогою властивості Elapsed, яка повертає об'єкт типу TimeSpan. За допомогою цього об'єкта і властивості TotalSeconds час відображається в секундах, включаючи і частки секунди.

Приклад 6.5.

Демонстрація застосування **методу Break() для переривання циклу**, який паралельно виконується методом For().

Опис роботи програми.

Це варіант прикладу 6.3, перероблений таким чином, щоб метод MyTransform() використовував тепер об'єкт типу ParallelLoopState в якості свого параметра, а метод Break() викликався при виявленні від'ємного

значення в масиві даних. Від'ємне значення, за яким переривається виконання циклу, вводиться в масив arr всередині методу Main(). Далі перевіряється стан завершення циклу перетворення даних. Властивість IsCompleted буде містити логічне значення false, оскільки в масиві arr виявляється від'ємного значення. При цьому на екран виводиться номер кроку, на якому цикл був перерваний. В програмі виключені всі надлишкові цикли, що застосовувалися в її попередній версії, а залишені тільки найефективніші з них:

- послідовно виконується цикл ініціалізації і
- паралельно виконується цикл перетворення.

```
// Використовувати об'єкти типу ParallelLoopResult і ParallelLoopState,
// а також метод Break() разом з методом For() для паралельного
// виконання циклу.

using System;
using System.Threading.Tasks;

namespace paral_break
{
    class Program
    {
        static int[] arr;

        // Метод слугує у якості тіла паралельно виконуємого циклу.
        // Оператори цього циклу просто використовують час ЦП для
        // демонстрації.

        static void Calculations(int I, ParallelLoopState pls)
        {
            if (arr[i] < 0) pls.Break();
            if ((0<arr[i]) && (arr[i] < 999999))
                    arr[i] = 1;
            if ((999999<arr[i]) && (arr[i] < 1999999))
                    arr[i] = 10;
            if ((1999999<arr[i]) && (arr[i] < 123999999))
                    arr[i] = 100;
        }

        static void Main(string[] args)
        {
            arr = new int[100000000];

            // Ініціалізувати дані
```

```
            for (int i = 0; i < arr.Length; i++)
                      arr[i] = i;

            // Помістити від'ємне значення в масив arr.

            arr[1000] = -10;

           // Паралельний варіант ініціалізації массиву в циклі

            ParallelLoopResult loopResult = Parallel.For(0, arr.Length,
Calculations);

           // Перевірка, чи завершився цикл.

          if (!loopResult.IsCompleted)
            Console.WriteLine("\nЦикл завершився передчасно через те, "
          + "що знайдене від'ємне значення\n" + "на кроці цикла номер "
                + loopResult.LowestBreakIteration + "\n");
            Console.WriteLine("Основний потік завершено.");
            Console.ReadLine();
        }
    }
}
```

Виконання цієї програми може привести, наприклад, до наступного результату.

Основний потік запущений.

Цикл завершився передчасно через те, що виявлено від'ємне значення на кроці циклу номер 1000

Основний потік завершено.

З наведеного вище результату видно, що цикл перетворення даних передчасно завершується після 1000 кроків. Справа у тому, що метод Break() викликається всередині методу MyTransform() при виявленні в масиві даних від'ємного значення.

2.6.3. Застосування методу ForEach()

Приклад 6.6. Демонстрування використання методу ForEach().

Опис роботи програми.

Створюється великий масив цілих значень. А різниця даного прикладу від прикладу 6.5 в тому, що метод, який виконується на кожному кроці циклу, просто виводить на консоль значення з масиву. Як правило, метод WriteLine() у розпаралелювані циклів не застосовується, тому що введення-виведення на консоль здійснюється настільки повільно, що цикл виявляється повністю прив'язаним до вводу-виводу. Але в даному прикладі метод WriteLine() застосовується виключно з метою демонстрації можливостей методу ForEach(). При виявленні від'ємного значення виконання циклу переривається викликом методу Break(). Незважаючи на те, що метод Break() викликається в одній задачи, інша задача може як і раніше виконуватися протягом кількох кроків циклу, перш ніж цикл буде перервано, хоча це залежить від конкретних умов роботи середовища виконання.

```
// Використовувати об'єкти типу ParallelLoopResult і ParallelLoopState,
// а також метод Break() разом з методом ForEach () для паралельного
// виконання циклу.

using System;
using System.Threading.Tasks;

namespace paral_foreach_break
{
    class Program
    {
        static int[] arr;

        // Метод як тіло для паралельно виконуваного циклу.
        // змінній  v передається значення елемента масива даних.

        static void DisplayArr(int v, ParallelLoopState pls)
        {
            // Перервати цикл при виявленні від'ємного значення.

            if (v < 0) pls.Break();
            Console.WriteLine("Значення: " + v);
        }
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("Основний потік запущено.");
            arr = new int[100000000];

            // Ініціалізувати дані.

            for (int i = 0; i < arr.Length; i++) arr[i] = i;

            // Помістити від'ємне значення в масив arr

            arr[10000] = -10;

            // Використовувати цикл, який паралельно виконується методом
           //    ForEach(), для відображення даних на екрані.

            ParallelLoopResult loopResult = Parallel.ForEach (arr,
DisplayArr);

            // Перевіка, чи завершився цикл.

            if (!loopResult.IsCompleted) Console.WriteLine ("\nЦикл
завершився передчасно через те, " + "що знайдене від'ємне значення \n"
+ "на кроці циклу номер " + loopResult.LowestBreakIteration + ".\n");
            Console.WriteLine("Основний потік завершено.");
            Console.ReadLine();
        }
    }
}
```

ЗАВДАННЯ

1. Для всіх завдань реалізуйте послідовну і паралельну обробку (методи Parallel.For, Parallel.ForEach тощо). Виконайте аналіз ефективності паралельної обробки.
2. Виконайте свій варіант відповідно до списку у підгрупи в проекті C# Windows Forms.

Варіант 1

Маємо дуже великий двомірний масив (20 000 x 20 000 елементів) і у ньому потрібно знайти найбільше число.

Варіант 2

Обчислити загальний розмір файлів в каталозі. Вхідний параметр: шлях до каталогу з файлами.

Варіант 3

Маємо дві великі матриці (по 10 000 х 10 000 елементів). Знайти суму матриць.

Варіант 4

Обробка всіх зображень з каталогу. Вхідні параметри: шлях до каталогу з файлами зображень, шлях до нового створюваному каталогу. Кожне зображення з початкового каталогу обертається та зберігається у новий створений каталог.

Варіант 5

Обчислити суму значень в масиві, який містить 100 000 000 елементів. Значення кожного елемента дорівнює його індексу.

Варіант 6

Маємо дві великі матриці (по 1000 х 1000 елементів). Знайти добуток матриць.

Варіант 7

Знайти суму квадратних коренів чисел в діапазоні від 1 до 10 000 000.

Варіант 8

Реалізуйте зведення елементів великого масиву (100 000 000 елементів) в ступінь *x*, використовуючи розподіл масиву на рівне число елементів. Число потоків задається параметром N.

Варіант 9

Додавання зображень. Вхідні параметри: шлях до каталогу з файлами зображень, шлях до нового створюваному каталогу, файл зображень для складання. Кожне зображення з початкового каталогу складається з заданим зображенням і результат зберігається в новий створений каталог.

Варіант 10

Обчислити загальну кількість слів у всіх файлах-документах в каталозі. Вхідний параметр: шлях до каталогу з файлами документів.

Варіант 11

Розбити масив, який складається з 100 000 000 елементів, на два підмасиви. Задається значення порогу x. У перший масив заносяться елементи, котрі менші зазначеного порогу, а у другий – ті що більше.

Варіант 12

Маємо дві великі матриці (по 10 000 x 10 000 елементів). Знайти різницю матриць.

КОНТРОЛЬНІ ПИТАННЯ

1. Які є два способи розпаралелювання програм?
2. Як створити задачу?
3. Яким чином можна повернути значення з задачи?
4. Як скасувати задачу?
5. Які методи має клас Parallel?
6. Як виконується розпаралелювання задач методом Invoke()?
7. Яка головна особливість методу For()?
8. Як застосовується метод ForEach()?

2.7. Лабораторна робота № 7. Можливості PLINQ

Мета роботи: розглянути особливості використання паралельної мови інтегрованих запитів PLINQ для досягнення паралелізму даних усередині запиту.

2.7.1. Створення паралельного запиту методом AsParallel()

Приклад 7.1. Простий запит PLINQ.

```
using System;
using System.Linq;

namespace l_7_1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arr = new int[10000000];

            // ініціалізувати масив даних позитивними значеннями

            for (int i = 0; i < arr.Length; i++) arr[i] = i;

            // ввести у масив даних рядок негативних значень

            arr[1000] = -1;
            arr[14000] = -2;
            arr[15000] = -3;
            arr[676000] = -4;
            arr[8024540] = -5;
            arr[9908000] = -6;

           // Використувувати запит PLINQ для пошуку негативних значень

            var negatives = from val in arr.AsParallel()
                            where val < 0
                            select val;

            foreach (var v in negatives)
                Console.Write(v + " ");
            Console.WriteLine();
        }
    }
```

```
}
```

Опис роботи програми.

Програма починається зі створення великого масиву arr, котрий ініціалізується цілими позитивними значеннями. Потім до нього вводиться ряд негативних значень. А далі формується запит на повернення послідовності негативних значень.

Метод *AsParallel()* викликається для джерела даних, яким слугує масив arr. Завдяки цьому дозволяється паралельне виконання операцій над масивом arr: виконується пошук негативних значень паралельно у кількох потоках. У разі виявлення негативних значень вони додаються до послідовності виведення. Це означає, що порядок формування послідовності виведення може і не відображати порядок розташування негативних значень у масиві arr.

Як приклад, нижче наведено результат виконання вище наведеного коду для двоядерної системи.

-5 -6 -1 -2 -3 -4

З прикладу видно, що у потоці, де пошук виконувався у верхній частині масиву, негативні значення -5 і -6 знайдені раніше, ніж значення -1 у потоці, де пошук відбувався в нижній частині масиву. Слід пам'ятати, що через відмінності у завантаженні процесора задачами, кількості доступних процесорів та інших чинників системного характеру можуть бути отримані різні результати. А найголовніше, що результуюча послідовність зовсім не обов'язково відображатиме порядок формування початкової послідовності.

Приклад 7.2. Запит PLINQ, в якому порядок проходження елементів у результуючій послідовності відображає порядок їх розташування у початкової послідовності.

```
using System;
```

```
using System.Linq;

namespace l_7_1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arr = new int[10000000];

            // ініціалізувати масив даних позитивними значеннями

            for (int i = 0; i < arr.Length; i++) arr[i] = i;

            // ввести у масив даних рядок негативних значень

            arr[1000] = -1;
            arr[14000] = -2;
            arr[15000] = -3;
            arr[676000] = -4;
            arr[8024540] = -5;
            arr[9908000] = -6;

          // Використувувати запит PLINQ для пошуку негативних значень

            var negatives = from val in arr.AsParallel().AsOrdered()
                            where val < 0
                            select val;

            foreach (var v in negatives)
                Console.Write(v + " ");
            Console.WriteLine();

        }
    }
}
```

Опис роботи програми.

Цей приклад відрізняється від прикладу 7.1 тим, що у паралельному запиті додається виклик методу AsOrdered(). Після виконання програми порядок розташування елементів у результуючій послідовності відображатиме порядок їх розташування у початковій послідовності.

-1 -2 -3 -4 -5 -6

Приклад 7.3. Програма демонструє порядок **скасування паралельного запиту**, сформованого на прикладі 7.1.

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace l_7_3
{
    class Program
    {
        static void Main(string[] args)
        {
            CancellationTokenSource cancelTokSrc = new
CancellationTokenSource();
            int[] arr = new int[10000000];

            // ініціалізувати масив даних позитивними значеннями

            for (int i = 0; i < arr.Length; i++) arr[i] = i;

            // ввести у масив даних рядок негативних значень

            arr[1000] = -1;
            arr[14000] = -2;
            arr[15000] = -3;
            arr[676000] = -4;
            arr[8024540] = -5;
            arr[9908000] = -6;

    // Використувувати запит PLINQ для пошуку негативних значень

            var negatives = from val in arr.AsParallel().
            WithCancellation(cancelTokSrc.Token)
                            where val < 0
                            select val;

// Створити задачу для скасування запиту після закінчення 50 м.секунд

            Task cancelTsk = Task.Factory.StartNew(() => {
                Thread.Sleep(50);
                cancelTokSrc.Cancel();
            });
            try
            {
                foreach (var v in negatives)
                    Console.Write(v + " ");
            }
```

```
            catch (OperationCanceledException exc)
            {
                Console.WriteLine(exc.Message);
            }
            catch (AggregateException exc)
            {
                Console.WriteLine(exc);
            }
            finally
            {
                cancelTsk.Wait();
                cancelTokSrc.Dispose();
                cancelTsk.Dispose();
            }
            Console.WriteLine();

        }
    }
}
```

Опис роботи програми.

У програмі організується окрема задача, яка очікує протягом 50 мілісекунд, а потім скасовує запит. Окрема задача потрібна тому, що цикл foreach, в якому виконується запит, блокує виконання методу Main() до завершення циклу.

Нижче наведено результат виконання цієї програми. Якщо запит скасується до її завершення, на екран виводиться повідомлення про виняткову ситуацію:

-5 -1 -2 -3 -4 The query has been canceled via the token supplied to WithCancellation.

ЗАВДАННЯ

1. Для всіх завдань реалізуйте послідовну і паралельну обробку для різних розмірів масиву. Виконайте аналіз ефективності паралельної обробки залежно від розміру масиву.

2. Виконайте свій варіант відповідно до списку у підгрупи в проекті C# Console.

Варіант 1

Визначити кількість нулів у масиві розміром 100 000 000 елементів і вивести результат.

Варіант 2

У масиві розміром у 100 000 елементів вибрати парні числа, упорядкувати їх за зростанням, згрупувати повторювані числа та вивести список унікальних парних чисел і скільки разів, кожне з них зустрічається в масиві. Обчислити загальну кількість парних чисел.

Варіант 3

Для масиву розміром 100 000 000 елементів знайти суму всіх елементів та вивести її.

Варіант 4

Обчислити послідовність чисел Фібоначчі кожного елементу в масиві, який має розмір 10 000 000 елементів, упорядкувати їх за зростанням, згрупувати повторювані числа і вивести результат.

Варіант 5

Визначити кількість позитивних чисел у масиві розміром у 100 000 елементів, упорядкувати їх за зростанням, згрупувати повторювані числа та вивести.

Варіант 6

Знайти середнє значення чисел у масиві, розміром 100 000 000 елементів, і вивести результат.

Варіант 7

Маємо масив розміром 100 000 000 елементів. Вибрати числа в заданому діапазоні, упорядкувати їх за зростанням, згрупувати повторювані числа і вивести результат.

Варіант 8

Маємо клас User, з властивостями Login та Password. Маємо великий масив об'єктів класу User. Знайти користувачів, у яких пароль пароль має довжину більше заданої. Вивести інформацію про знайдених користувачів.

Варіант 9

Знайти факторіали для кожного елементу в масиві розміром 10 000 000 елементів, упорядкувати їх за зростанням, згрупувати повторювані значення та вивести результат.

Варіант 10

Визначити кількість негативних чисел у масиві розміром 100 000 елементів, упорядкувати їх за спаданням, згрупувати повторювані числа та вивести результат.

Варіант 11

Маємо клас User з властивостями Name та City. Маємо великий масив об'єктів класу User. Знайти користувачів, які мешкають у заданому місті. Вивести місто та імена знайдених користувачів.

Варіант 12

Визначити кількість непарних чисел у масиві розміром у 100 000 елементів, упорядкувати їх за спаданням, згрупувати повторювані числа та вивести результат.

КОНТРОЛЬНІ ПИТАННЯ

1. Як створити паралельний запит?
2. Для чого використовується клас ParallelEnumerable?
3. Для чого застосовується метод AsParallel()?
4. Що виконує метод AsOrdered()?
5. Як перервати паралельний запит до закінчення його виконання?
6. Як запросити послідовне виконання частини паралельного запиту?

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Побегайло А. «Системное программирование в Windows». – СПб.: БХВ-Петербург, 2006. – 1056 с.
2. Шилдт Г. Полное руководство по C# 4.0.: Пер. с англ. - М.: ООО «И. Д. Вильямс», 2012. - 1047 с.
3. Andrew Troelsen, Phillip Japikse. Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming. Tenth Edition - Apress, 2021. – 1353p.
4. Клири Стивен. Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование. 2-е межд. изд. — СПб.: Питер, 2020. — 272 с.
5. Rishabh Verma, Neha Shrivastava, Ravindra Akella. Parallel Programming with C# and .NET Core: Developing Multithreaded Applications Using C# and .NET Core 3.1 from Scratch (English Edition) – BPB PUBLICATIONS, 2020. – 394p.
6. What is .NET? Introduction and overview [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/core/introduction. – Last access: 2021. – Title from the screen.
7. Параллельное программирование в .NET [Электроний ресурс] – Режим доступу. – URL: https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming (Дата звертання: 20.05.2020).
8. Managed threading basics [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/uk-ua/dotnet/standard/threading/managed-threading-basics 15.09.2021. – Last access: 2021. – Title from the screen.
9. Thread Class [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread. – Last access: 2021. – Title from the screen.

10. Process Class [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process. – Last access: 2021. – Title from the screen.
11. Overview of synchronization primitives [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives. – Last access: 2021. – Title from the screen.
12. Memory-mapped files [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files. – Last access: 2021. – Title from the screen.
13. Pipe Operations in .NET [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/standard/io/pipe-operations. – Last access: 2021. – Title from the screen.
14. Task Parallel Library (TPL) [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/uk-ua/dotnet/standard/parallel-programming/task-parallel-library-tpl. – Last access: 2021. – Title from the screen.
15. System.Threading.Tasks Пространство имен [Электроний ресурс] – Режим доступу. – URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.tasks. – (Дата звертання: 20.05.2021).
16. Task Class [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task. – Last access: 2021. – Title from the screen.
17. Parallel Class [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel. – Last access: 2021. – Title from the screen.
18. Introduction to PLINQ [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/introduction-to-plinq. – Last access: 2021. – Title from the screen.