Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

Методичні вказівки

до виконання лабораторних робіт
з дисципліни «Паралельні та розподілені обчислення»
для студентів спеціальності
123 «Комп’ютерна інженерія»
усіх форм навчання

Кривий Ріг
2021

Укладачі: Музика І. О., канд. техн. наук, доцент
Кузнєцов Д. І., канд. техн. наук, доцент
Рецензент: Жосан А. А., канд. техн. наук, доцент

Дані методичні вказівки містять завдання та теоретичні відомості для виконання лабораторних робіт з дисципліни «Паралельні та розподілені обчислення» за спеціальністю 123 «Комп'ютерна інженерія». Висвітлені питання розробки програмних додатків із застосуванням багатопоточності та засобів синхронізації доступу до розподілених даних на базі платформи .NET Framework та мови програмування C#.

Розглянуто
на засіданні кафедри
комп'ютерних систем та мереж

Протокол № 1
від 27.08.2021 р.

Схвалено
на вченій раді факультету
інформаційних технологій

Протокол № 1
від 30.08.2021 р.

ВСТУП

Паралельні та розподілені обчислення – спосіб організації роботи програми, що ґрунтується на використанні декількох процесорів чи обчислювальних ядер комп'ютерної системи. Сьогодні паралельні обчислення використовуються навіть при розробці програмного забезпечення для мобільних пристроїв. Технології побудови та відлагодження паралельних програм досить складні і, тому виходять за межі курсу звичайного програмування. Організацію програми з використанням декількох потоків підтримують майже всі сучасні мови програмування, зокрема: Java, C#, C++, Python, Object Pascal, Objective-C та ін.

Дисципліна «Паралельні та розподілені обчислення» – одна із фундаментальних дисциплін професійного спрямування у загальній системі підготовки фахівців за спеціальністю 123 «Комп'ютерна інженерія». Передумовою успішного вивчення даної дисципліни є володіння навичками програмування та алгоритмізації, які отримані під час вивчення дисциплін попередніх курсів навчання: «Основи інформаційних технологій», «Вища математика», «Програмування», «Об'єктно-орієнтоване програмування».

Метою викладання даної навчальної дисципліни є вивчення принципів побудови паралельних алгоритмів, застосування спеціальних засобів синхронізації (семафорів, м'ютексів, моніторів), керування процесами та потоками, використання сучасних програмних бібліотек на мовах C# та C++.

Запропонований матеріал буде корисним як студентам, що вивчають програмування, так і професійним програмістам, які займаються розробкою паралельних комп'ютерних додатків, систем розподіленої обробки інформації.

ЛАБОРАТОРНА РОБОТА № 1

Тема: знайомство з потоками в C#.

Мета: створити потоки, використовуючи клас `Thread`, виявити проблеми потокобезпечності, проаналізувати паралельну програму за допомогою візуалізатора паралелізму Visual Studio.

Теоретичні відомості

C# підтримує паралельне виконання коду через багатопоточність. Потік – це незалежний шлях виконання, здатний виконуватися одночасно з іншими потоками.

Програма на C # запускається як єдиний потік, автоматично створюваний CLR і операційною системою (головний потік), і стає багатопотоковою за допомогою створення додаткових потоків. Розглянемо простий приклад.

Зауваження: всі приклади вимагають імпорту наступних просторів імен: `System` та `System.Threading`.

```
using System;
using System.Threading;
namespace ThreadTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t = new Thread(WriteY);
            t.Start();      // Виконати WriteY у новому потоці
            while (true)
                Console.Write("x"); // Весь час друкувати 'x'
        }

        static void WriteY()
        {
            while (true)
                Console.Write("y"); // Весь час друкувати 'y'
```

```
        }
    }
}
```

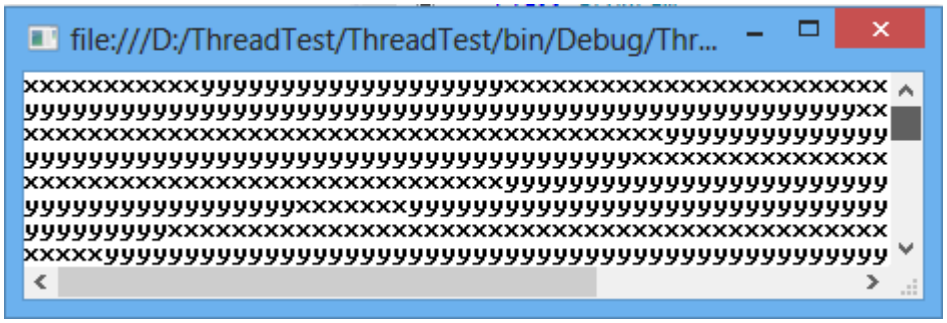


Рисунок 1.1 – Результат роботи програми

У головному потоці створюється новий потік *t*, метод якого безперервно друкує символ «*y*». Одночасно головний потік безперервно друкує символ «*x*». CLR призначає кожному потоку свій стек, тому локальні змінні зберігаються окремо.

У наступному прикладі визначаємо метод з локальною змінною, а потім виконуємо його одночасно в головному і в новоствореному потоці.

```
static void Main()
{
  new Thread(Go).Start();    // Выполнить Go() в новом потоке
  Go();                      // Выполнить Go() в главном потоке
}

static void Go()
{
  // Определяем и используем локальную переменную 'cycles'
  for (int cycles = 0; cycles < 5; cycles++)
    Console.Write('#');
}
```

Консольне виведення: ##########. Окремий екземпляр `cycles` створюється в стеку кожного потоку, тому виводиться, як і очікувалося, десять знаків «#».

Разом з тим потоки розділяють дані, які належать до того ж екземпляра об’єкта, що й самі потоки.

```
public class Common
{
    bool done;
    public void Go()   // Go – экземплярный метод
    {
        if (!done)
        {
            done = true;
            Console.WriteLine("Done");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Common common = new Common();   // Создаем общий объект
        new Thread(common.Go).Start();
        common.Go();
        Console.ReadKey();
    }
}
```

Оскільки обидва потоки викликають метод `Go()` одного і того ж об’єкта `Common`, вони поділяють поле `done`. Результат – «*Done*», надруковане один раз замість двох.

Для статичних полів працює інший спосіб розділення даних між потоками. Ось той же самий приклад, але зі статичним полем `done`.

```
public class Common
{
    static bool done; // Статическое поле, разделяемое потоками
    public void Go()  // Go – экземплярный метод
    {
        if (!done)
        {
            done = true;
```

```
            Console.WriteLine("Done");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Common common = new Common();    // Создаем общий объект
        new Thread(common.Go).Start();
        common.Go();
        Console.ReadKey();
    }
}
```

Обидва приклади демонструють інше ключове поняття – потокову безпеку (або скоріше її відсутність). Фактично результат виконання програми не визначений: можливо (хоча і малоймовірно), «*Done*» буде надруковано двічі. Однак якщо ми поміняємо порядок викликів в методі `Go()`, шанси побачити «*Done*» надрукованим два рази підвищуються радикально.

```
public void Go()
{
    if (!done)
    {
        Console.WriteLine("Done");
        done = true;
    }
}
```

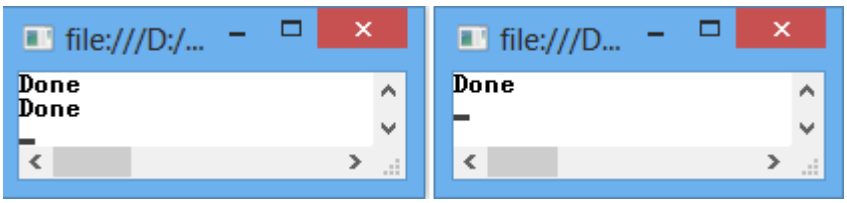


Рисунок 1.2 – Випадковий характер роботи програми

Проблема полягає в тому, що один потік може виконати оператор `if`, поки інший потік виконує `WriteLine`, тобто до того як `done` буде

встановлено в true. Однин із підходів до вирішення даної задачі полягає в отриманні ексклюзивного блокування на час читання і запису загальних змінних (полів об'єкта). C # забезпечує це за допомогою оператора lock.

```
public class Common
{
    static bool done;
    static object locker = new object();
    public void Go()
    {
        lock (locker)
        {
            if (!done)
            {
                Console.WriteLine("Done");
                done = true;
            }
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Common common = new Common();   // Создаем общий объект
        new Thread(common.Go).Start();
        common.Go();
        Console.ReadKey();
    }
}
```

Коли два потоки одночасно борються за блокування (у нашому випадку об'єкта locker), один потік переходить у режим очікування (блокується), поки блокування не звільняється. У даному випадку це гарантує, що тільки один потік може одночасно виконувати критичну секцію коду, і «Done» буде надруковано лише один раз. Код, захищений таким чином, називається потокобезпечним.

Тимчасове призупинення (блокування) – основний спосіб координації, або синхронізації дій потоків. Очікування ексклюзивного

блокування – це одна з причин, з якої потік може блокуватися. Інша причина – якщо потік призупиняється (Sleep) на заданий проміжок часу.

```
Thread.Sleep(TimeSpan.FromSeconds(30)); // Блокування на 30 с
```

Також потік може очікувати завершення іншого потоку, викликаючи його метод Join.

```
Thread t = new Thread(Go);     // Go – статический метод
t.Start();
t.Join();                      // Ожидаем завершения потока
```

Коли потік блокований, він не споживає ресурсів CPU.

Як працює багатопоточність

Управління багатопоточністю здійснює планувальник потоків. Цю функцію CLR зазвичай делегує операційній системі. Планувальник потоків гарантує, що активним потокам виділяється відповідний час на виконання, а потоки, які очікують або блоковані – не споживають ресурси CPU.

На однопроцесорних комп'ютерах планувальник потоків використовує квантування часу – швидке перемикання між активними потоками. Це призводить до непередбачуваної поведінки, як у самому першому прикладі, де кожна послідовність символів «X» і «Y» відповідає кванту часу, виділеного потоку. У Windows XP типове значення кванта часу (десятки мілісекунд) набагато більше, ніж витрати CPU на перемикання контексту між потоками (декілька мікросекунд).

На багатопроцесорних комп'ютерах багатопоточність реалізована як суміш квантування часу і справжнього паралелізму, коли різні потоки виконують код на різних CPU. Необхідність квантування часу все одно залишається, так як операційна система повинна обслуговувати як свої власні потоки, так і потоки інших додатків.

Кажуть, що потік витісняється, коли його виконання призупиняється через зовнішній фактор типу квантування часу. У

більшості випадків потік не може контролювати, коли і де він буде витіснений.

Потоки чи процеси

Всі потоки однієї програми логічно містяться в межах процесу – модуля операційної системи, в якому виконується додаток.

У деяких аспектах потоки і процеси схожі, наприклад, час поділяється між процесами, що виконуються на одному комп'ютері так само, як між потоками одного C#-додатка. Ключова відмінність полягає в тому, що процеси повністю ізольовані один від одного. Потоки поділяють пам'ять з іншими потоками цього ж додатка. Завдяки цьому один потік може постачати дані у фоновому режимі, а інший – показувати ці дані протягом їх надходження.

Коли використовувати потоки

Типовий додаток з багатопоточністю виконує тривалі обчислення у фоновому режимі. Головний потік продовжує виконання, в той час як робочий потік виконує фонову задачу. У додатках *Windows Forms*, коли головний потік зайнятий тривалими обчисленнями, він не може обробляти повідомлення клавіатури і миші, і додаток перестає відповідати. З цієї причини слід запускати завдання, які віднімають багато часу, у робочому потоці, навіть якщо головний потік в цей час демонструє користувачеві модальний діалог з написом «Працюю... Будь ласка, чекайте», оскільки програма не може перейти до наступної операції, поки не закінчена поточна. Таке рішення гарантує, що програма не буде позначено операційною системою як «Не відповідає», спокушаючи користувача з горя закінчити процес. У цьому випадку модальний діалог може надати кнопку «Скасувати», так як форма продовжує отримувати повідомлення, поки завдання виконується у фоновому потоці. Клас `BackgroundWorker` напевно стане в нагоді при реалізації такої моделі.

У разі додатків без *UI*, наприклад, служб *Windows*, багатопоточність має сенс, якщо завдання може зайняти багато часу, оскільки потрібно очікувати відповіді від іншого комп'ютера (сервера додатків, сервера баз даних або клієнта). Запуск такого завдання в окремому робочому потоці означає, що головний потік негайно звільняється для інших завдань.

Інше застосування багатопоточності знаходить у методах, що виконують інтенсивні обчислення. Такі методи можуть виконуватися швидше на багатопроцесорних комп'ютерах, якщо робоче навантаження розподіллено по декількох потокам (кількість процесорів можна отримати через властивість `Environment.ProcessorCount`).

C#-додаток можна зробити багатопоточним двома способами: або явно створюючи додаткові потоки і керуючи ними, або використовуючи можливості неявного створення потоків .NET Framework – `BackgroundWorker`, пул потоків , потоковий таймер, Remoting-сервер , Web-служби або додаток ASP.NET. У двох останніх випадках альтернативи багатопоточності не існує. Однопотоковий web-сервер не просто поганий, він просто неможливий! На щастя, у випадку серверних додатків, що не зберігають стан (*stateless*), багатопоточність реалізується зазвичай досить просто, складності можливі хіба що в синхронізації доступу до даних в статичних змінних.

Коли потоки не потрібні

Потік разом з перевагами має і свої недоліки. Самий головний з них – значне збільшення складності програм. Складність збільшують не додаткові потоки самі по собі, а необхідність організації їх взаємодії. Від того, наскільки ця взаємодія є складною, залежить тривалість циклу розробки. Таким чином, потрібно або підтримувати дизайн взаємодії потоків простим, або не використовувати багатопоточність взагалі, якщо тільки ви не маєте протиприродної схильності до переписування та відлагодження коду.

Крім того, надмірне використання багатопоточності забирає ресурси і час CPU на створення потоків і перемикання між потоками. Зокрема, коли використовуються операції читання/запису на диск, більш швидким може виявитися послідовне виконання завдань в одному або двох потоках, ніж одночасне їх виконання в декількох потоках.

Створення та запуск потоків

Для створення потоків використовується конструктор класу `Thread`, який приймає як параметр делегат типу `ThreadStart`, що вказує метод, який потрібно виконати. Делегат `ThreadStart` визначається так:

```
public delegate void ThreadStart();
```

Виклик методу `Start` починає виконання потоку. Потік триває до виходу з виконуваного методу. Ось приклад, що використовує повний синтаксис C# для створення делегата `ThreadStart`:

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread(new ThreadStart(Go));
    t.Start();   // Выполнить Go() в новом потоке.
    Go();  // Одновременно запустить Go() в главном потоке.
  }
  static void Go()
  {
    Console.WriteLine("hello!");
  }
}
```

У цьому прикладі потік виконує метод `Go()` одночасно з головним потоком. Результат – два майже одночасних «hello»:

```
hello!
hello!
```

Потік можна створити, використовуючи більш зручний скорочений синтаксис C#:

```
static void Main()
{
  Thread t = new Thread(Go); //Без явного использования
ThreadStart
  t.Start();
  ...
}
static void Go() { ... }
```

У цьому випадку делегат `ThreadStart` виводиться компілятором автоматично. Інший варіант скороченого синтаксису використовує анонімний метод для створення потоку:

```
static void Main()
{
  Thread t = new Thread( delegate()
    {Console.WriteLine("Hello!");});
  t.Start();
}
```

Потік має властивість `IsAlive`, яка повертає `true` після виклику `Start()` і до завершення потоку. Потік, який закінчив виконання, не може бути початий заново.

Передача даних в ThreadStart

Припустимо, що в розглянутому вище прикладі ми хочемо більш явно розрізняти виведення результату на консоль кожного з потоків, наприклад, по регістру символів. Можна досягти цього, передаючи відповідний прапор в метод `Go()`, але в цьому випадку не можна використовувати делегат `ThreadStart`, оскільки він не приймає аргументів. На щастя,. *NET Framework* визначає іншу версію делегата – `ParameterizedThreadStart`, який може приймати один аргумент:

```
public delegate void ParameterizedThreadStart(object obj);
```

Попередній приклад можна переписати так:

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread(Go);
    t.Start(true);             // == Go(true)
    Go(false);
  }

  static void Go(object upperCase)
  {
    bool upper = (bool)upperCase;
    Console.WriteLine(upper ? "HELLO!" : "hello!");
  }
}
```

Консольний результат:

```
hello!
HELLO!
```

У цьому прикладі компілятор автоматично виводить делегат `ParameterizedThreadStart`, так як метод `Go()` приймає як параметр один `object`. З тим же успіхом можна було написати:

```
Thread t = new Thread(new ParameterizedThreadStart(Go));
t.Start(true);
```

Особливість використання `ParameterizedThreadStart` полягає в тому, що перед використанням потрібно привести аргумент з типу `object` до потрібного типу (в даному випадку `bool`).

В якості альтернативи можна використовувати анонімний метод:

```
static void Main()
{
  Thread t = new Thread(delegate(){ WriteText("Hello"); });
  t.Start();
}
static void WriteText(string text)
{
  Console.WriteLine(text);
}
```

Зручність полягає в тому, що потрібний метод (в даному випадку `WriteText`) можна викликати з будь-якою кількістю аргументів і без жодного приведення типів. Однак потрібно взяти до уваги особливість семантики анонімних методів, пов’язану із зовнішньої змінною, яка стає очевидною в наступному прикладі:

```
static void Main()
{
  string text = "Before";
  Thread t = new Thread(delegate() { WriteText(text); });
  text = "After";
  t.Start();
}

static void WriteText(string text)
{
  Console.WriteLine(text);
}
```

Консольний результат:

```
After
```

Анонімні методи відкривають химерні можливості ненавмисної взаємодії через зовнішні змінні, якщо вони змінюються кимось після старту потоку. Взаємодії через поля класу, як правило, більш ніж достатньо. Найкраще, як тільки почалося виконання потоку, розглядати зовнішні змінні як змінні тільки для читання, за винятком реалізацій з відповідними блокуваннями на обох сторонах.

Інший спосіб передачі даних у потік полягає у запуску в потоці методу певного екземпляра об’єкту, а не статичного методу. Тоді властивості обраного об’єкта будуть визначати поведінку потоку, як у наступному варіанті оригінального прикладу.

```
class ThreadTest
{
  bool upper;

  static void Main()
  {
```

```
    ThreadTest instance1 = new ThreadTest();
    instance1.upper = true;
    Thread t = new Thread(instance1.Go);
    t.Start();
    ThreadTest instance2 = new ThreadTest();

  instance2.Go(); //Запуск в главном потоке - с upper=false
  }

  void Go()
  {
    Console.WriteLine(upper ? "HELLO!" : "hello!");
  }
}
```

Іменування потоків

Потоку можна дати ім’я, використовуючи властивість `Name`. Це надає велику зручність при налагодженні: імена потоків можна вивести в `Console.WriteLine` і побачити у вікні *Debug-Threads* в Microsoft Visual Studio. Ім’я потоку може бути призначено у будь-який момент, але тільки один раз – при спробі змінити його буде згенеровано виняток.

Головному потоку додатку також можна призначити ім’я – в наступному прикладі доступ до головного потоку здійснюється через статичну властивість `CurrentThread` класу `Thread`:

```
class ThreadNaming
{
  static void Main()
  {
    Thread.CurrentThread.Name = "main";
    Thread worker = new Thread(Go);
    worker.Name = "worker";
    worker.Start();
    Go();
  }
```

```
  static void Go()
  {
    Console.WriteLine("Hello from " +
      Thread.CurrentThread.Name);
  }
}
```

Консольний результат:

```
Hello from main
Hello from worker
```

Основні та фонові потоки

За умовчанням потоки створюються як основні, що означає, що програма не буде завершена, поки один з таких потоків буде виконуватися. C# також підтримує фонові потоки, вони не продовжують життя додатку, а завершуються відразу ж, як тільки всі основні потоки будуть завершені.

Зміна статусу потоку з основного на фоновий не змінює його пріоритет або статус в планувальнику потоків. Статус потоку перемикається з основного на фоновий за допомогою властивості `IsBackground`, як показано в наступному прикладі:

```
class PriorityTest
{
  static void Main(string[] args)
  {
    Thread worker = new Thread(delegate()
      {Console.ReadLine();});
    if (args.Length > 0)
      worker.IsBackground = true;
    worker.Start();
  }
}
```

Якщо програма викликається без аргументів, робочий потік виконується за умовчуванням як основний потік і чекає на `ReadLine`, поки користувач не натисне *Enter*. Тим часом головний потік

завершується, але додаток продовжує виконуватися, оскільки робочий потік ще живий.

Якщо ж програму запустити з аргументами командного рядка, робочий потік отримає статус фонового і програма завершиться практично відразу після завершення головного потоку, із знищенням потоку, який очікує введення користувача за допомогою методу `ReadLine`.

Коли фоновий потік завершується таким способом, всі блоки `finally` всередині потоку ігноруються. Оскільки невиконання коду в `finally` зазвичай небажано, буде правильно очікувати завершення всіх фонових потоків перед виходом з програми, призначивши потрібний таймаут (за допомогою методу `Thread.Join`). Якщо з якихось причин робочий потік не завершується за виділений час, можна спробувати аварійно завершити його (`Thread.Abort`), а якщо і це не вдається зробити, дозволити йому бути знищеним разом з процесом.

Перетворення робочого потоку у фоновий може бути останнім шансом завершити додаток, оскільки не вмираючий основний потік не дасть додатку завершитися. Завислий основний потік особливо підступний в додатках *Windows Forms*, оскільки додаток завершується, коли завершується його головний потік, але його процес продовжує виконуватися. У диспетчері завдань він зникне із списку додатків, хоча ім'я його виконуваного файлу залишиться в списку процесів, що виконуються. Поки користувач не знайде і не видалить його, процес продовжить споживати ресурси і, можливо, буде перешкоджати запуску або нормальному функціонуванню знову запущеного екземпляра додатку.

Пріоритети потоків

Властивість `Priority` визначає, скільки часу на виконання буде виділено потоку у порівнянні з іншими потоками того ж процесу. Існує 5 градацій пріоритету потоку.

```
enum ThreadPriority {Lowest, BelowNormal, Normal, AboveNormal,
Highest}
```

Значення пріоритету стає важливим, коли одночасно виконуються декілька потоків. Установка пріоритету потоку на максимум ще не означає роботу в реальному часі (*real-time*), так як існує ще пріоритет процесу всього додатку. Щоб працювати в реальному часі, потрібно використовувати клас `Process` з простору імен `System.Diagnostics` для підняття пріоритету процесу, наприклад:

```
Process.GetCurrentProcess().PriorityClass =
ProcessPriorityClass.High;
```

Від `ProcessPriorityClass.High` один крок до найвищого пріоритету процесу – `Realtime`. Встановлюючи пріоритет процесу в `Realtime`, ви говорите операційній системі, що хочете, щоб ваш процес ніколи не витіснявся. Якщо ваша програма випадково потрапить в нескінченний цикл, операційна система може бути повністю заблокована. Врятувати вас в цьому випадку зможе тільки кнопка вимкнення живлення. З цієї причини `ProcessPriorityClass.High` вважається максимальним пріоритетом процесу, придатним до вживання.

Якщо *real-time* програма має користувальницький інтерфейс, може бути не бажано піднімати пріоритет його процесу, так як оновлення екрану буде споживати занадто багато часу CPU, гальмуючи весь комп’ютер, особливо якщо UI досить складний. Зменшення пріоритету головного потоку в поєднанні з підвищенням пріоритету процесу гарантує, що *real-time* потік не буде витіснятися перемальовуванням екрану, але не рятує від гальм весь комп’ютер , так як операційна система все ще буде виділяти багато часу CPU всьому процесу в цілому. Ідеальне рішення полягає в тому, щоб тримати роботу в реальному часі і користувальницький інтерфейс в різних процесах (з різними пріоритетами), що підтримують зв’язок через *Remoting* або *shared memory*. Загальна пам’ять вимагає звернення до Win32 API.

Обробка виключень

Обрамлення коду створення і запуску потоку блоками try/catch/finally має мало сенсу. Подивіться наступний приклад:

```
public static void Main()
{
  try
  {
    new Thread(Go).Start();
  }
  catch(Exception ex)
  {
    // Сюда мы никогда не попадем!
    Console.WriteLine("Исключение!");
  }

  static void Go() { throw null; }
}
```

Try/catch тут фактично зовсім марний, і NullReferenceException в новоствореному потоці оброблено не буде, оскільки кожен потік має свій незалежний шлях виконання. Рішення полягає в додаванні обробки виключень безпосередньо в метод потоку.

```
public static void Main()
{
  new Thread(Go).Start();
}

static void Go()
{
  try
  {
    ...
    throw null;      // это исключение будет поймано ниже
    ...
  }
  catch(Exception ex)
  {
```

```
    Логирование исключения и/или сигнал другим потокам
    ...
  }
}
```

Починаючи з. NET 2.0, необроблений виняток у будь-якому потоці призводить до закриття всієї програми, а значить ігнорування винятків – це поганий метод програмування. Отже, блок `try/catch` необхідний у кожному методі потоку, принаймні, в додатках не для власного вживання, щоб уникнути закриття програми через необроблене виключення. Це може бути досить обтяжливо, особливо для програмістів Windows Forms, які використовують глобальний перехоплювач винятків, як показано нижче:

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

static class Program
{
  static void Main()
  {
    Application.ThreadException += HandleError;
    Application.Run(new MainForm());
  }

  static void HandleError(object sender,
ThreadExceptionEventArgs e)
  {
    Логирование исключения, завершение или продолжение работы
  }
}
```

Подія Application.ThreadException виникає, коли виняток генерується в коді, який був викликаний з обробника повідомлення Windows (наприклад, від клавіатури, миші і т. д.) – коротше кажучи , практично з будь-якого коду програми Windows Forms. Оскільки це чудово працює, з’являється почуття удаваної безпеки, що всі виключення будуть оброблені цим центральним обробником. Винятки,

що виникають у робочих потоках – гарний приклад винятків, які не ловляться в Application.ThreadException.

.NET Framework надає низькорівневу подію для глобальної обробки виключень – `AppDomain.UnhandledException`. Ця подія відбувається, коли є необроблене виняток у будь-якому потоці і для будь-яких типів додатків (з призначеним для користувача інтерфейсом або без нього). Однак, хоча це і хороший спосіб реєстрації необроблених винятків, він не надає ніякого способу запобігти закриттю програми.

Отже, явна обробка виключень потрібна у всіх потокових методах. Спростити роботу можна, використовуючи класи-обгортки, наприклад, `BackgroundWorker`.

Аудиторні завдання

Завдання № 1

Написати програму, в якій два потоки T0 і T1 виводять на екран символи * та #, потік T0 виводить десять символів «*», а T1 – вісім символів «#». Виконати аналіз за допомогою «Візуалізатора паралелізму» Microsoft Visual Studio.

Розв’язання

У самому спрощеному варіанті програму можна представити у такому вигляді:

```
using System;
using System.Threading;

namespace Lab3
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t0 = new Thread(Print0);
            Thread t1 = new Thread(Print1);
            t0.Start();
            t1.Start();
```

```
            Console.ReadKey();
        }

        static void Print0()
        {
            for (int i = 0; i < 10; i++)
                Console.Write('*');
        }

        static void Print1()
        {
            for (int i = 0; i < 8; i++)
                Console.Write('#');
        }
    }
}
```

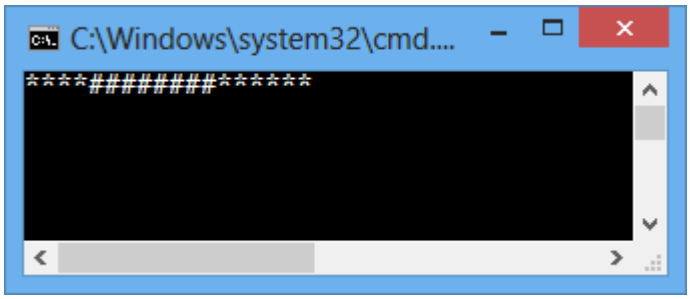


Рисунок 1.3 – Один з можливих результатів виконання програми

Однак для більш детального аналізу її виконання можна додати виведення повідомлень при старті та завершенні потоків. З метою очікування завершення кожного з новостворених потоків доцільно застосувати метод потоку `Join()`.

```
using System;
using System.Threading;

namespace Lab3
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t0 = new Thread(Print0);
            Thread t1 = new Thread(Print1);
            t0.Start();
            t1.Start();
            t0.Join();  // Блокировка основного потока,
                        // пока не завершится t0
```

```
            t1.Join();  // Блокировка основного потока,
                        // пока не завершится t1
            Console.ReadKey();
        }

        static void Print0()
        {
            Console.WriteLine("Print0 started");
            for (int i = 0; i < 10; i++)
                Console.Write('*');
            Console.WriteLine("Print0 finished");
        }

        static void Print1()
        {
            Console.WriteLine("Print1 started");
            for (int i = 0; i < 8; i++)
                Console.Write('#');
            Console.WriteLine("Print1 finished");
        }
    }
}
```

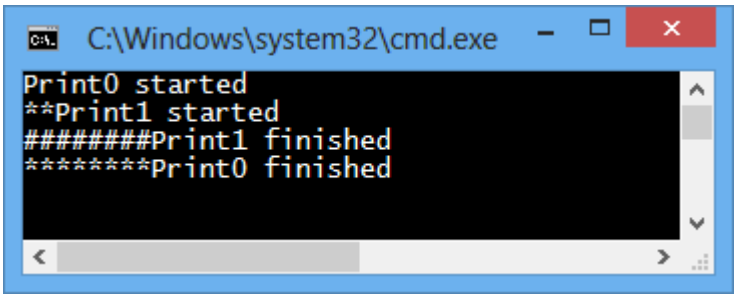


Рисунок 1.4 – Робота програми з повідомленням про початок та завершення потоків

Середовище розробки Visual Studio 12 містить корисний інструмент для аналізу ефективності виконання паралельної програми – «Візуалізатор паралелізму» (рисунки 1.5, 1.6). Для запуску інструменту на вкладці «Аналіз» необхідно запустити «Візуалізатор паралелізму» → «Виконати аналіз з поточним процесом». Інструмент запускає програму і збирає інформацію про її фактичне виконання на обчислювальної системі. Основні вкладки результатів: «Використання», «Потоки», «Ядра».

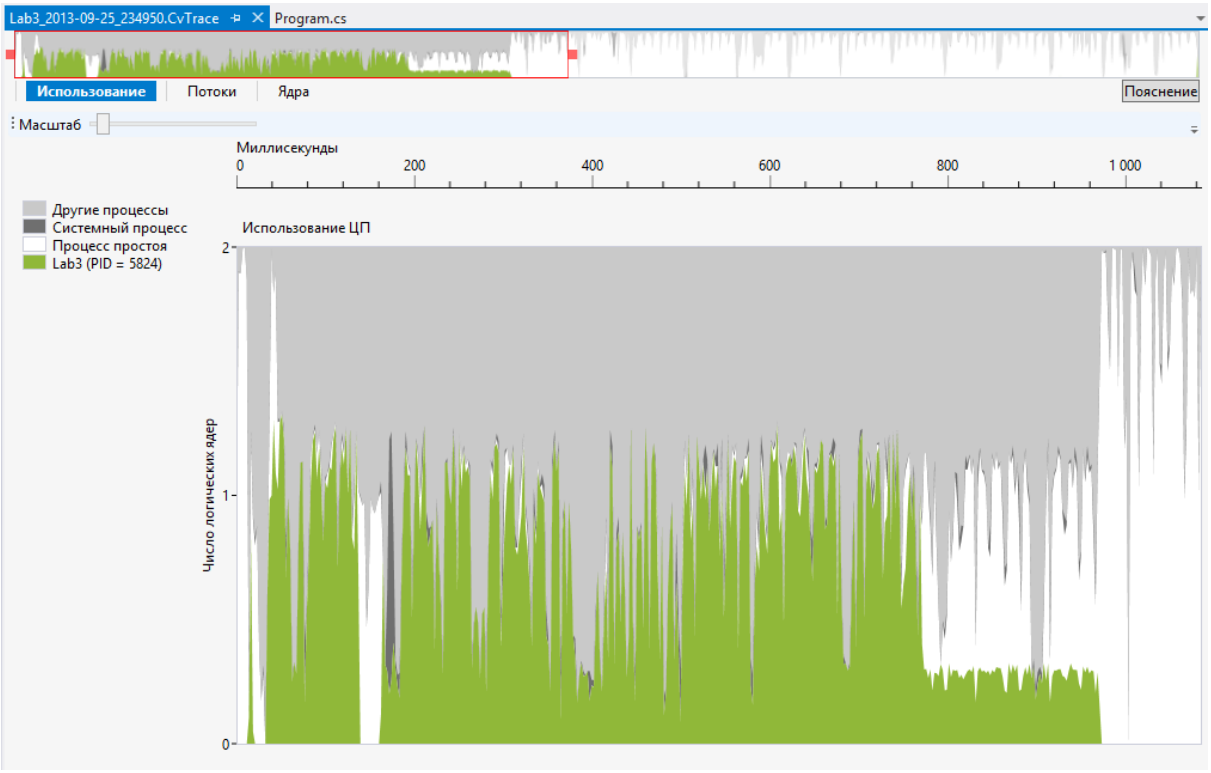


Рисунок 1.5 – Результати аналізу «Візуалізатором паралелізму»

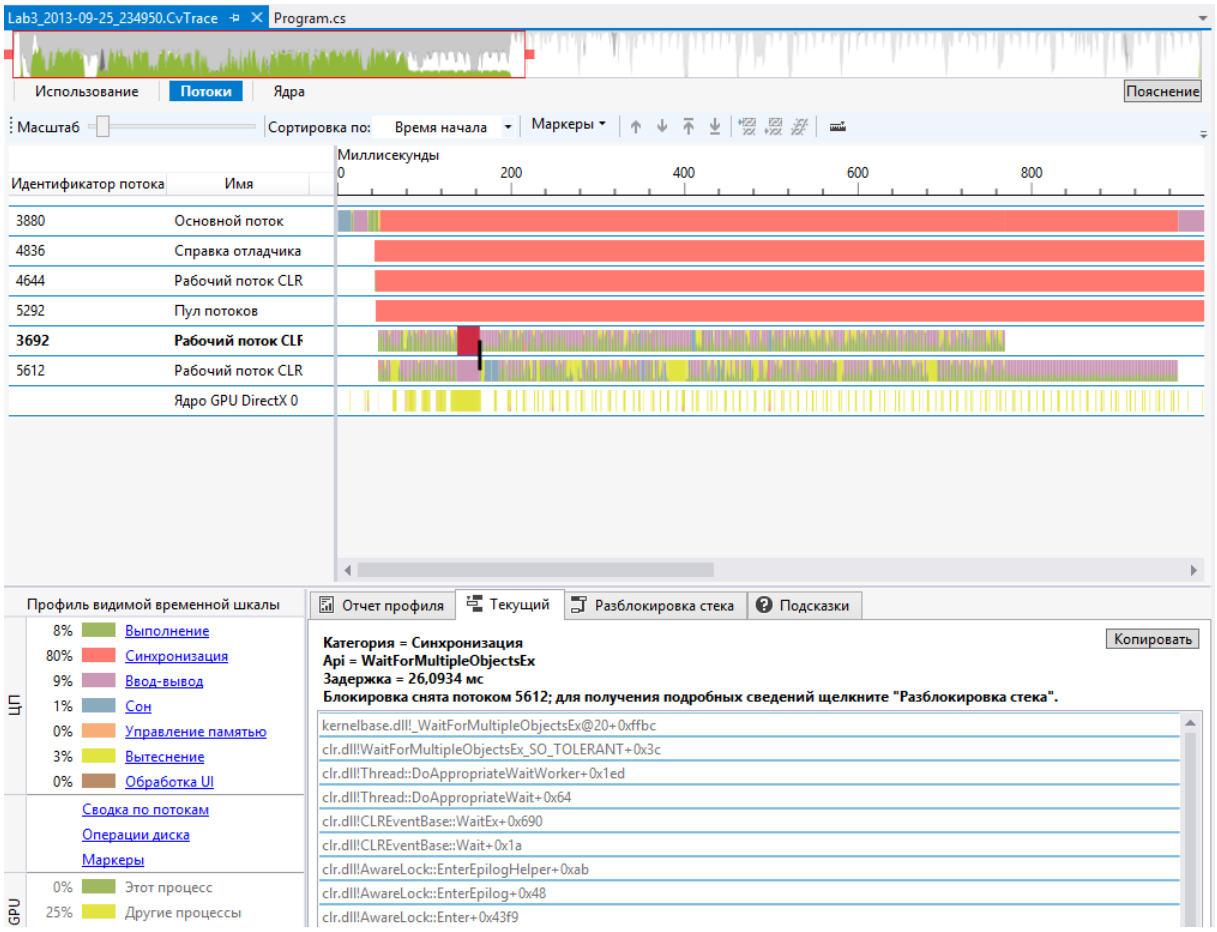


Рисунок 1.6 – Детальна інформація про хід виконання потоків у часі

Індивідуальні завдання

Відповідно до варіанту та обраного рівня складності виконати наступні завдання.

№	Рівень складності		
	Початковий	Базовий	Високий
1	Відкомпілювати та протестувати аудиторні завдання	Завдання № 1	Завдання № 2
2	–	Завдання № 2	Завдання № 3

Завдання № 1

Написати програму, в якій два потоки T0 і T1 виконують певні дії відповідно до таблиці варіантів.

№ варіанта	Завдання
1	T0: Виводить 10 випадкових цілих чисел в діапазоні від 0 до 100. T1: Виводить 20 випадкових літер англійського алфавіту.
2	T0: Виводить 5 символів «=». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
3	T0: Виводить 50 випадкових парних чисел. T1: Виводить 20 символів «@».
4	T0: Виводить 80 випадкових цілих чисел в діапазоні від -5 до 5. T1: Виводить 40 випадкових літер російського алфавіту.
5	T0: Виводить 5 символів «=». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
6	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «₴».
7	T0: Виводить 10 випадкових цілих чисел в діапазоні від -2 до 10. T1: Виводить 20 символів «\».
8	T0: Виводить 5 символів «=». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
9	T0: Виводить 50 випадкових парних чисел. T1: Виводить 20 символів «-».

№ варіанта	Завдання
10	T0: Виводить 80 випадкових цілих чисел в діапазоні від -5 до 5. T1: Виводить 40 випадкових літер російського алфавіту.
11	T0: Виводить 5 символів «+». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
12	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «.».
13	T0: Виводить 10 випадкових цілих чисел в діапазоні від -80 до 0. T1: Виводить 20 випадкових літер англійського алфавіту.
14	T0: Виводить 5 символів «=». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
15	T0: Виводить 50 випадкових парних чисел. T1: Виводить 20 символів «@».
16	T0: Виводить 50 випадкових цілих чисел в діапазоні від -5 до 5. T1: Виводить 40 випадкових літер російського алфавіту.
17	T0: Виводить 50 символів «=». T1: Виводить 24 випадкових цілих чисел в діапазоні від 0 до 10.
18	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «₴».
19	T0: Виводить 10 випадкових цілих чисел в діапазоні від -2 до 10. T1: Виводить 20 символів «\».
20	T0: Виводить 5 символів «=». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
21	T0: Виводить 50 випадкових парних чисел. T1: Виводить 20 символів «-».
22	T0: Виводить 80 випадкових цілих чисел в діапазоні від -5 до 5. T1: Виводить 40 випадкових літер російського алфавіту.
23	T0: Виводить 5 символів «+». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
24	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «.».
25	T0: Виводить 50 випадкових дробових чисел. T1: Виводить 20 символів «\\».
26	T0: Виводить 8 випадкових цілих чисел в діапазоні від -5 до 5. T1: Виводить 15 випадкових літер.

№ варіанта	Завдання
27	T0: Виводить 5 символів «+». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
28	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «!».
29	T0: Виводить 5 символів «/\». T1: Виводить 8 випадкових цілих чисел в діапазоні від 0 до 10.
30	T0: Виводить 35 випадкових непарних чисел. T1: Виводить 70 символів «\/».

Завдання № 2

Написати програму, в якій два потоки T0 і T1 виконують певні дії відповідно до таблиці варіантів.

№ варіанта	Завдання
1	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти суму всіх елементів. T1: Знайти добуток всіх елементів.
2	Створити, ініціалізувати масив з 15 цілих чисел. T0: Вивести всі елементи з парними індексами. T1: Вивести квадрати всіх елементів з непарними індексами.
3	Створити, ініціалізувати масив з 20 цілих чисел. T0: Знайти середнє арифметичне значення. T1: Знайти середнє геометричне значення.
4	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, більші за число 10. T1: Вивести на консоль середнє арифметичне значення масиву.
5	Створити, ініціалізувати масив з 18 цілих чисел у проміжку (0..90). T0: Вивести квадрати всіх елементів. T1: Вивести числа, які менші 50.

№ варіанта	Завдання
6	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, менші за число 15. T1: Знайти середнє геометричне значення.
7	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти кількість чисел більших за 10 та вивести отримане число на консоль. T1: Знайти кількість парних чисел та вивести отримане число на консоль.
8	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, менші за число 20, але більші за 10. T1: Вивести квадрати всіх елементів масиву.
9	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти добуток чисел, які менші за середнє арифметичне чисел масиву. T1: Знайти суму всіх парних елементів.
10	Створити, ініціалізувати масив з 10 чисел типу double у проміжку (0..50). T0: Знайти суму чисел більших за 15. T1: Знайти добуток чисел менших за 10.
11	Створити, ініціалізувати масив з 8 цілих чисел у проміжку (0..15). T0: Знайти їх суму. T1: Знайти кількість непарних чисел.
12	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти суму всіх елементів. T1: Знайти добуток всіх елементів.
13	Створити, ініціалізувати масив з 15 цілих чисел. T0: Вивести всі елементи з парними індексами.

№ варіанта	Завдання
	T1: Вивести квадрати всіх елементів з непарними індексами.
14	Створити, ініціалізувати масив з 20 цілих чисел. T0: Знайти середнє арифметичне значення. T1: Знайти середнє геометричне значення.
15	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, більші за число 10. T1: Вивести на консоль середнє арифметичне значення масиву.
16	Створити, ініціалізувати масив з 18 цілих чисел у проміжку (0..90). T0: Вивести квадрати всіх елементів. T1: Вивести числа, які менші 50.
17	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, менші за число 15. T1: Знайти середнє геометричне значення.
18	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти кількість чисел більших за 10 та вивести отримане число на консоль. T1: Знайти кількість парних чисел та вивести отримане число на консоль.
19	Створити, ініціалізувати масив з 15 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, менші за число 20, але більші за 10. T1: Вивести квадрати всіх елементів масиву.
20	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти добуток чисел, які менші за середнє арифметичне чисел масиву. T1: Знайти суму всіх парних елементів.
21	Створити, ініціалізувати масив з 10 чисел типу double у проміжку (0..50).

№ варіанта	Завдання
	T0: Знайти суму чисел більших за 15. T1: Знайти добуток чисел менших за 10.
22	Створити, ініціалізувати масив з 24 цілих чисел у проміжку (0..15). T0: Знайти їх суму. T1: Знайти кількість парних чисел.
23	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Знайти суму всіх елементів. T1: Знайти добуток всіх елементів.
24	Створити, ініціалізувати масив з 15 цілих чисел. T0: Вивести всі елементи з парними індексами. T1: Вивести квадрати всіх елементів з непарними індексами.
25	Створити, ініціалізувати масив з 20 цілих чисел. T0: Знайти середнє арифметичне значення. T1: Знайти середнє геометричне значення.
26	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..25). T0: Вивести на консоль усі числа, більші за число 10. T1: Вивести на консоль середнє арифметичне значення масиву.
27	Створити, ініціалізувати масив з 18 цілих чисел у проміжку (0..90). T0: Вивести квадрати всіх елементів. T1: Вивести числа, які менші 50.
28	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (10..25). T0: Вивести на консоль усі числа, менші за число 15. T1: Знайти середнє геометричне значення.
29	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (0..50). T0: Знайти кількість чисел більших за 10 та вивести отримане число на консоль. T1: Знайти кількість парних чисел та вивести отримане число на консоль.

№ варіанта	Завдання
30	Створити, ініціалізувати масив з 10 цілих чисел у проміжку (100..500). T0: Вивести на консоль усі числа, менші за число 200, але більші за 100. T1: Вивести квадрати всіх елементів масиву.

Завдання № 3

Дано два вектори X та Y, які мають розмірності N_X та N_Y відповідно. Вектори потрібно заповнити випадковими числами типу `double` або `float`. Написати програму, в якій два потоки T0 і T1 виконують обчислення виразів A та B. Ініціалізація потоків проводиться у головному потоці програми. Діапазон випадкових чисел задано параметром D. У програмі потокам необхідно задати відповідні імена та налаштувати пріоритет.

Після відлагодження коду необхідно виконати аналіз за допомогою «Візуалізатора паралелізму» Microsoft Visual Studio, порівнявши швидкість виконання та завантаження ядер процесора відповідно до пріоритету потоків.

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
1	T0	Lowest	$N_X = 10$ $D_X = [0..25]$	$A = \sum_{i=1}^{N} \sqrt[3]{x_i sin(x_i^2)}$
	T1	BelowNormal	$N_Y = 15$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} \sqrt[3]{y_i cos(y_i^2)}$
2	T0	Normal	$N_X = 8$ $D_X = [0..50]$	$A = \sum_{i=1}^{N} x_i^3 + 10$
	T1	AboveNormal	$N_Y = 18$ $D_Y = [0..100]$	$B = \sum_{i=1}^{N} y_i^2 + \sqrt[4]{y_i}$

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
3	T0	Highest	$N_X = 20$ $D_X = [0..25]$	$A = 2 + \sum_{i=1}^{N} x_i$
	T1	Lowest	$N_Y = 25$ $D_Y = [0..10]$	$B = \sqrt[3]{e + \sum_{i=1}^{N} y_i + y_i^2}$
4	T0	BelowNormal	$N_X = 14$ $D_X = [0..25]$	$A = \sum_{i=0}^{N-1} x_i + 1$
	T1	Normal	$N_Y = 35$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} y_i^2 + \sqrt[4]{y_i + 5}$
5	T0	AboveNormal	$N_X = 18$ $D_X = [0..25]$	$A = 5 + \prod_{i=1}^{N} (x_i + 1)$
	T1	Highest	$N_Y = 40$ $D_Y = [0..20]$	$B = \pi \sqrt[3]{y_i sin(e y_i^2)}$
6	T0	Lowest	$N_X = 15$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e^{2i}$
	T1	BelowNormal	$N_Y = 26$ $D_Y = [-10..10]$	$B = y_0^2 + \sqrt[3]{\sum_{i=1}^{N} y_i + i}$
7	T0	Normal	$N_X = 100$ $D_X = [0..25]$	$A = N * e + \sum_{i=1}^{N} x_i$
	T1	AboveNormal	$N_Y = 150$ $D_Y = [-10..10]$	$B = 5 + \prod_{i=1}^{N} y_i$
8	T0	Highest	$N_X = 85$ $D_X = [0..250]$	$A = \sum_{i=1}^{N} \sqrt[3]{x_i sin(e x_i^2)}$
	T1	Lowest	$N_Y = 150$ $D_Y = [-10..0]$	$B = \sum_{i=1}^{N} \sqrt[4]{y_i cos(y_i^2)}$

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
9	T0	BelowNormal	$N_X = 10$ $D_X = [0..25]$	$A = \prod_{i=1}^{N} x_i^3 + \sqrt[3]{x_i + 1}$
	T1	Normal	$N_Y = 23$ $D_Y = [-10..10]$	$B = \prod_{i=1}^{N} y_i^2 + \sqrt[4]{y_i - 1}$
10	T0	AboveNormal	$N_X = 14$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e * x_i$
	T1	Highest	$N_Y = 19$ $D_Y = [-10..10]$	$B = \prod_{i=1}^{N} \sqrt[3]{y_i sin(y_i^2)}$
11	T0	Lowest	$N_X = 25$ $D_X = [0..25]$	$A = 2 + \sum_{i=1}^{N} x_i$
	T1	BelowNormal	$N_Y = 15$ $D_Y = [-10..10]$	$B = \sqrt[3]{1 + \sum_{i=1}^{N} y_i^2}$
12	T0	Normal	$N_X = 19$ $D_X = [0..25]$	$A = \sum_{i=0}^{N-1} x_i + 1$
	T1	AboveNormal	$N_Y = 17$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} y_i^2 + \sqrt[4]{y_i + 5}$
13	T0	Highest	$N_X = 100$ $D_X = [0..25]$	$A = 5 + \prod_{i=1}^{N} (x_i + 1)$
	T1	Lowest	$N_Y = 150$ $D_Y = [-10..10]$	$B = \pi \sqrt[3]{y_i sin(e y_i^2)}$
14	T0	BelowNormal	$N_X = 10$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e^{2i}$
	T1	Normal	$N_Y = 15$ $D_Y = [-10..10]$	$B = y_0^2 + \sqrt[3]{\sum_{i=1}^{N} y_i + i}$

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
15	T0	AboveNormal	$N_X = 100$ $D_X = [0..25]$	$A = N * e + \sum_{i=1}^{N} x_i$
	T1	Highest	$N_Y = 150$ $D_Y = [-10..10]$	$B = 5 + \prod_{i=1}^{N} y_i$
16	T0	Lowest	$N_X = 40$ $D_X = [0..25]$	$A = \sum_{i=1}^{N} \sqrt[3]{x_i sin(x_i^2)}$
	T1	BelowNormal	$N_Y = 28$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} \sqrt[4]{cos(1 + y_i^2)}$
17	T0	Normal	$N_X = 120$ $D_X = [0..25]$	$A = \prod_{i=1}^{N} x_i^3 + \sqrt[3]{x_i + 1}$
	T1	AboveNormal	$N_Y = 150$ $D_Y = [-10..10]$	$B = \prod_{i=1}^{N} y_i^2 + \sqrt[4]{y_i - 1}$
18	T0	Highest	$N_X = 100$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e * x_i$
	T1	Lowest	$N_Y = 145$ $D_Y = [-10..10]$	$B = \prod_{i=1}^{N} \sqrt[3]{y_i sin(y_i^2)}$
19	T0	BelowNormal	$N_X = 200$ $D_X = [0..25]$	$A = 2 + \sum_{i=1}^{N} x_i$
	T1	Normal	$N_Y = 150$ $D_Y = [-10..10]$	$B = \sqrt[3]{e + \sum_{i=1}^{N} y_i + y_i^2}$
20	T0	AboveNormal	$N_X = 10$ $D_X = [0..25]$	$A = \sum_{i=0}^{N-1} x_i + 1$
	T1	Highest	$N_Y = 15$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} y_i^2 + \sqrt[4]{y_i + 5}$

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
21	T0	Lowest	$N_X = 80$ $D_X = [0..25]$	$A = 5 + \prod_{i=1}^{N}(x_i + 1)$
	T1	BelowNormal	$N_Y = 50$ $D_Y = [-10..10]$	$B = \pi \sqrt[3]{y_i sin(e y_i^2)}$
22	T0	Normal	$N_X = 10$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e^{2i}$
	T1	AboveNormal	$N_Y = 15$ $D_Y = [-10..10]$	$B = y_0^2 + \sqrt[3]{\sum_{i=1}^{N} y_i + i}$
23	T0	Highest	$N_X = 10$ $D_X = [0..25]$	$A = N * e + \sum_{i=1}^{N} x_i$
	T1	Lowest	$N_Y = 15$ $D_Y = [-10..10]$	$B = 5 + \prod_{i=1}^{N} y_i$
24	T0	BelowNormal	$N_X = 10$ $D_X = [0..25]$	$A = \sum_{i=1}^{N} \sqrt[3]{x_i sin(x_i^2)}$
	T1	Normal	$N_Y = 15$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} \sqrt[4]{y_i cos(y_i^2)}$
25	T0	AboveNormal	$N_X = 10$ $D_X = [0..5]$	$A = \prod_{i=1}^{N} x_i^3 + \sqrt[3]{x_i + 1}$
	T1	Highest	$N_Y = 15$ $D_Y = [-10..20]$	$B = \prod_{i=1}^{N} y_i^2 + \sqrt[4]{y_i - 1}$
26	T0	Lowest	$N_X = 10$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N}(5 - x_i)$
	T1	BelowNormal	$N_Y = 15$ $D_Y = [-10..10]$	$B = \prod_{i=1}^{N} \sqrt[3]{y_i sin(y_i^2)}$

№ варіанта	Завдання			
	Ім’я потоку	Пріоритет	Параметри	Обчислення
27	T0	Normal	$N_X = 10$ $D_X = [0..12]$	$A = 2 + \sum_{i=1}^{N} x_i$
	T1	AboveNormal	$N_Y = 15$ $D_Y = [0..10]$	$B = \sqrt[3]{e + \sum_{i=1}^{N} y_i + y_i^2}$
28	T0	Highest	$N_X = 10$ $D_X = [0..25]$	$A = \sum_{i=0}^{N-1} x_i + 1$
	T1	Lowest	$N_Y = 15$ $D_Y = [-10..10]$	$B = \sum_{i=1}^{N} y_i^2 + \sqrt[4]{y_i + 5}$
29	T0	BelowNormal	$N_X = 10$ $D_X = [0..25]$	$A = 5 + \prod_{i=1}^{N} (x_i + 1)$
	T1	Normal	$N_Y = 15$ $D_Y = [-10..10]$	$B = \pi \sqrt[3]{y_i sin(e y_i^2)}$
30	T0	AboveNormal	$N_X = 10$ $D_X = [0..25]$	$A = N + \sum_{i=1}^{N} e^{2i}$
	T1	Highest	$N_Y = 15$ $D_Y = [-10..10]$	$B = y_0^2 + \sqrt[3]{\sum_{i=1}^{N} y_i + i}$

Контрольні питання

1. Коротко охарактеризуйте засоби паралельних обчислень платформи Microsoft .NET Framework.
2. Які методи класу Thread існують для керування потоками у C#?
3. Що таке критична секція коду?
4. Як налаштовується пріоритет потоку?
5. Як впливає пріоритет на час виконання задачі?

6. За допомогою яких класів .NET Framework можна одержати інформацію про кількість процесорів, ядер та їх характеристики?

7. У чому відмінність процесу та потоку?

8. Коротко проаналізуйте можливості візуалізатора паралелізму Visual Studio.

ЛАБОРАТОРНА РОБОТА № 2

Тема: засоби синхронізації потоків в C#.

Мета: виконати паралельні обчислення з використанням декількох потоків та засобів синхронізації (семафорів, м'ютексів, моніторів).

Теоретичні відомості

У наступних таблицях наведено інформацію про інструменти.NET для координації (синхронізації) потоків:

Таблиця 2.1 – Найпростіші методи блокування

Конструкція	Призначення
Sleep	Блокування на вказаний час
Join	Очікування закінчення іншого потоку

Таблиця 2.2 – Конструкції блокування

Конструкція	Призначення	Доступність з інших процесів	Швидкість
Lock (Monitor.Enter / Monitor.Exit)	Гарантує, що тільки один потік може отримати доступ до ресурсу або секції коду	ні	швидко
Mutex	Гарантує, що тільки один потік може отримати доступ до ресурсу або секції коду. Може використовуватися для запобігання запуску декількох екземплярів програми	так	середня
Semaphore	Гарантує, що не більше заданого числа потоків може отримати доступ до ресурсу або секції коду	так	середня

Таблиця 2.3 – Сигнальні конструкції

Конструкція	Призначення	Доступність з інших процесів	Швидкість
EventWaitHandle	Дозволяє потоку очікувати сигналу від іншого потоку	так	середня
Wait and Pulse	Дозволяє потоку очікувати, поки не виконається задана умова блокування	ні	середня

Таблиця 2.4 – Конструкції синхронізації без блокування

Конструкція	Призначення	Доступність з інших процесів	Швидкість
Interlocked	Виконання простих неблокуючих атомарних операцій	так, через пам’ять, що розділяється	дуже швидко
volatile	Для безпечного доступу до полів без блокування	так, через пам’ять, що розділяється	дуже швидко

Блокування

Коли потік зупинений в результаті використання конструкцій, перерахованих у вищенаведених таблицях, кажуть, що він блокований. Будучи блокованим, потік негайно перестає отримувати час CPU, встановлює властивість `ThreadState` в `WaitSleepJoin` і залишається в такому стані, поки не розблокується. Розблокування може відбутися в наступних чотирьох випадках:

- виконається умова розблокування;
- закінчиться таймаут операції (якщо він був заданий);
- по перериванню через `Thread.Interrupt`;
- по аварійному завершенню через `Thread.Abort`.

Потік не вважається блокованим, якщо його виконання призупинено нерекомендованим методом `Suspend`.

Sleeping або Spinning

Виклик `Thread.Sleep` блокує поточний потік на вказаний час (або до переривання):

```
static void Main()
{
  Thread.Sleep(0);     // отказаться от одного кванта
времени CPU
  Thread.Sleep(1000);  // заснуть на 1000 миллисекунд
  Thread.Sleep(TimeSpan.FromHours(1));  // заснуть на 1 час
  Thread.Sleep(Timeout.Infinite);       // заснуть до
прерывания
}
```

Якщо бути більш точним, `Thread.Sleep` звільняє CPU і повідомляє, що потоку не потрібно виділяти час у вказаний період. `Thread.Sleep(0)` звільняє CPU для виділення одного кванта часу наступного потоку в черзі на виконання.

Унікальність `Thread.Sleep` серед інших методів блокування в тому, що він призупиняє передачу повідомлень Windows у додатках Windows Forms. Через це тривале блокування головного (UI) потоку додатку Windows Forms призводить до того, що додаток перестає відповідати, тому використання `Thread.Sleep` треба уникати незалежно від того, чи дійсно передача черги повідомлень технічно припинена.

Клас `Thread` також надає метод `SpinWait`, який не відмовляється від часу CPU, а навпаки, завантажує процесор в циклі на задану кількість ітерацій. 50 ітерацій еквівалентні паузі приблизно в мікросекунду, хоча це залежить від швидкості і завантаження CPU. Технічно `SpinWait` – неблокуючий метод: `ThreadState` такого потоку не встановлюється в `WaitSleepJoin` і потік не може бути перерваний з іншого потоку. `SpinWait` рідко використовується, його головне застосування це очікування ресурсу, який повинен звільниться дуже скоро (протягом декількох мікросекунд) без виклику `Sleep` і витрат процесорного часу на перемикання потоку. Однак ця методика вигідна тільки на багатопроцесорних комп’ютерах, на однопроцесорному

комп’ютері у ресурсу немає ніякого шансу звільнитися. Часті або тривалі виклики `SpinWait` даремно розтрачують час CPU.

Блокування проти очікування у циклі

Потік може очікувати виконання деякої умови, безпосередньо прокручуючи цикл перевірки, наприклад:

```
while (!proceed)
{
    …
};
```

або

```
while (DateTime.Now < nextStartTime)
{
    …
};
```

Проте це дуже марна витрата процесорного часу: оскільки CLR і операційна система переконані, що потік виконує важливі обчислення, йому виділяються відповідні ресурси. Потік, який крутиться в такому стані, не вважається заблокованим, на відміну від потоку, що очікує на EventWaitHandle (конструкції, що зазвичай використовуються для таких завдань).

Іноді використовується гібрид блокування та очікування у циклі:

```
while (!proceed)
  Thread.Sleep(x);    // "Spin-Sleeping!"
```

Чим більше `x`, тим ефективніше використовується CPU. Платою за компроміс стає збільшення латентності. При перевищенні 20 мс накладні витрати незначні, якщо умова в `while` не дуже складна.

За винятком незначної затримки ця комбінація блокування та періодичних опитувань може працювати досить непогано.

Очікування завершення потоку

Потік можна заблокувати до завершення іншого потоку викликом методу `Join`:

```
class JoinDemo
{
  static void Main()
  {
    Thread t = new Thread(delegate() { Console.ReadLine();
});
    t.Start();
    t.Join();    // ожидать, пока поток не завершится
    Console.WriteLine("Thread t’s ReadLine complete!");
  }
}
```

Метод Join може також брати в якості аргументу `timeout` – час в мілісекундах. Якщо вказаний час минув, а потік не завершився, `Join` повертає `false`. `Join` з `timeout` функціонує як `Sleep` – фактично наступні два рядки коду призводять до однакового результату:

```
Thread.Sleep(1000);
Thread.CurrentThread.Join(1000);
```

Блокування і потокова безпека

Блокування забезпечує монопольний доступ і використовується, щоб забезпечити виконання однієї секції коду тільки одним потоком одночасно. Для прикладу розглянемо наступний клас:

```
class ThreadUnsafe
{
  static int val1, val2;

  static void Go()
  {
    if (val2 != 0)
```

```
      Console.WriteLine(val1 / val2);
    val2 = 0;
  }
}
```

Він не є потокобезпечним: якби метод `Go` викликався двома потоками одночасно, можна було б отримати помилку ділення на 0, так як змінна `val2` могла бути встановлена в 0 в одному потоці, в той час коли інший потік знаходився б між `if` і `Console.WriteLine`.

Ось як за допомогою блокування можна вирішити цю проблему:

```
class ThreadSafe
{
  static object locker = new object();
  static int val1, val2;

  static void Go()
  {
    lock (locker)
    {
      if (val2 != 0)
        Console.WriteLine(val1 / val2);
      val2 = 0;
    }
  }
}
```

Тільки один потік може одноразово заблокувати об’єкт синхронізації (в даному випадку `locker`), а всі інші конкуруючі потоки будуть припинені, поки блокування не буде знято. Якщо за блокування борються кілька потоків, вони ставляться в чергу очікування – «ready queue» і обслуговуються, як тільки це стає можливим, за принципом «першим прийшов – першим обслужений». Ексклюзивне блокування, як уже говорилося, забезпечує послідовний доступ до того, що воно захищає, тому виконувані потоки вже не можуть накластись один на одного. У даному випадку ми захистили логіку всередині методу `Go`, так само, як і поля `val1` і `val2`.

Потік, заблокований на час очікування звільнення блокування, має властивість `ThreadState`, встановлену в `WaitSleepJoin`. Пізніше буде

показано, як потік, заблокований в такому стані, може бути примусово звільнений з іншого потоку викликом методів `Interrupt` або `Abort`. Це досить потужна можливість, що використовується зазвичай для завершення робочого потоку.

Оператор `lock` мови C # фактично є синтаксичним скороченням для викликів методів `Monitor.Enter` і `Monitor.Exit` у межах блоків `try/finally`. Ось у такий фрагмент коду фактично розгортається реалізація методу `Go` з попереднього прикладу :

```
Monitor.Enter(locker);
try
{
  if (val2 != 0)
    Console.WriteLine(val1 / val2);
  val2 = 0;
}

finally { Monitor.Exit(locker); }
```

Виклик `Monitor.Exit` без попереднього виклику `Monitor.Enter` для того ж об’єкта синхронізації викличе виняток.

`Monitor` також надає метод `TryEnter`, що дозволяє задати час очікування в мілісекундах або у вигляді `TimeSpan`. Метод повертає `true`, якщо блокування було отримано, і `false`, якщо блокування не було отримано за заданий час. `TryEnter` може також бути викликаний без параметрів і в цьому випадку повертає управління негайно.

Вибір об’єкта синхронізації

Будь-який об’єкт, який видимий взаємодіючим потокам, може бути використаний як об’єкт синхронізації, якщо це об’єкт посилального типу. Також строго рекомендується, щоб об’єкт синхронізації був private-полем класу, щоб уникнути випадкового впливу зовнішнього коду, блокуючого цей об’єкт. Згідно з цими правилами, об’єктом синхронізації цілком може стати сам об’єкт, що захищається, як, наприклад, список в наступному прикладі:

```
class ThreadSafe()
{
  List <string> list = new List <string>();

  void Test()
  {
    lock (list)
    {
      list.Add("Item 1");
     ...
```

Зазвичай використовується виділене поле (як locker в попередніх прикладах), тому що це дозволяє точніше контролювати область видимості і ступінь деталізації блокування. Використання об'єкта або типу в якості об'єкта синхронізації не схвалюється, оскільки передбачає public-область видимості об'єкта синхронізації:

```
lock (this)
{
 ...
}
```

або

```
lock (typeof(Widget)) // Для защиты статических данных
{
 ...
}
```

Блокування не забороняє взагалі будь-який доступ до об'єкта. Іншими словами, виклик `x.ToString()` не буде заблокований із-зі того, що інший потік викликав `lock(x)`.

Вкладені блокування

Потік може неодноразово блокувати один і той же об'єкт багаторазовими викликами `Monitor.Enter` або вкладеними `lock`-ами. Об'єкт буде звільнений, коли буде виконано відповідну кількість разів `Monitor.Exit` або відбудеться вихід із самої зовнішньої конструкції

lock. Тому припустима природна семантика, коли один метод викликає інший наступним чином:

```
static object x = new object();

static void Main()
{
  lock (x)
  {
     Console.WriteLine(“I have the lock”);
     Nest();
     Console.WriteLine(“I still have the lock”);
  } // Здесь блокировка будет снята!
}

static void Nest()
{
  lock (x)
  {
   ...
  }

  //Блокировка еще не снята!
}
```

Потік може бути блокований тільки на самому першому, зовнішньому операторі lock.

Коли потрібно блокувати

Як правило, будь-яке поле, доступне декільком потокам, має читатися і записуватися з блокуванням. Навіть в самому простому випадку, операції привласнення одиночному полю, необхідна синхронізація. У наступному класі ні прирощення, ні присвоювання не є потокобезпечними:

```
class ThreadUnsafe
{
  static int x;
```

```
  static void Increment() { x++; }
  static void Assign()    { x = 123; }
}
```

А ось їх потокобезпечні варіанти:

```
class ThreadUnsafe
{
  static object locker = new object();
  static int x;

  static void Increment()
  {
    lock (locker)
      x++;
  }

  static void Assign()
  {
    lock (locker)
      x = 123;
  }
}
```

В якості альтернативи блокуванню в таких простих випадках можна використовувати неблокиучі конструкції синхронізації.

Блокування і атомарность

Якщо група змінних завжди читається і записується в межах одного блокування, можна сказати, що змінні читаються і пишуться атомарно. Припустимо, що поля x і y завжди читаються і пишуться з блокуванням на об’єкті locker:

```
lock (locker)
{
  if (x != 0)
    y /= x;
}
```

Можна сказати, що доступ до x і y атомарний, оскільки даний фрагмент коду не може бути перерваний діями іншого потоку, які б змінили x, y або результат операції. Неможливо отримати помилку ділення на нуль, якщо звернення до x і y проводиться в ексклюзивному блокуванні.

Продуктивність при блокуванні

Блокування саме по собі дуже швидка операція: вона вимагає десятків наносекунд, якщо власне блокування не відбувається. Якщо потрібно блокування, то подальше перемикання завдань займає вже мікросекунди або навіть мілісекунди на перепланування потоків. Однак порівняйте це з годинами, які ви повинні будете витратити, не поставивши `lock` там, де треба.

При неправильному використанні у блокуваннях можуть бути і негативні наслідки – зменшення можливості паралельного виконання потоків, взаємоблокуванні, гонки блокувань. Можливості для паралельного виконання зменшуються, коли занадто багато коду поміщено в конструкцію `lock`, змушуючи інші потоки простоювати весь час, поки цей код виконується. Взаємоблокування настає, коли кожен з двох потоків очікує на блокуванні іншого потоку і, таким чином, ні той, ні інший не можуть рушити далі. Гонкою блокувань називається ситуація, коли будь-який з двох потоків може першим отримати блокування, однак програма видає помилковий результат, якщо першим це зробить «неправильний» потік.

Взаємоблокування – загальний синдром багатьох об’єктів синхронізації. Хороше правило, що допомагає уникати взаємоблокувань, полягає в тому, щоб починати з блокування мінімальної кількості об’єктів, і збільшувати ступінь деталізації блокувань, коли розмір коду в блокуванні надмірно збільшується.

Mutex

М’ютекс забезпечує ті ж самі функціональні можливості, що і оператор `lock` в C#, що робить його не дуже затребуваним. Єдина перевага полягає в тому, що `Mutex` доступний з різних процесів, забезпечуючи блокування на рівні комп’ютера, на відміну від оператора `lock`, який діє тільки на рівні додатку.

Блокування `lock` швидше в сотні разів за `Mutex`. Отримання м’ютекса займає кілька мікросекунд, виклик lock – десятки наносекунд (якщо не відбувається саме блокування).

Метод `WaitOne` для `Mutex` отримує виняткове блокування, блокуючи потік, якщо це необхідно. Виняткове блокування може бути зняте викликом методу `ReleaseMutex`. Так само як оператор `lock` в C#, `Mutex` може бути звільнений тільки з того ж потоку, що його захопив.

Типове використання м’ютекса для взаємодії процесів – забезпечення можливості запуску тільки одного примірника програми. Ось як це робиться:

```
class OneAtATimePlease
{
  // Используем уникальное имя приложения,
  // например, с добавлением имени компании
  static Mutex mutex = new Mutex(false, "oreilly.com
OneAtATimeDemo");

  static void Main()
  {
    // Ожидаем получения мьютекса 5 сек – если уже есть
запущенный
    // экземпляр приложения - завершаемся.
    if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false))
    {
      Console.WriteLine("В системе запущен другой экземпляр
программы!");
      return;
    }
```

```
        try
        {
          Console.WriteLine("Работаем - нажмите Enter для
выхода...");
          Console.ReadLine();
        }
        finally { mutex.ReleaseMutex(); }
      }
    }
```

Корисна властивість Mutex-а – якщо додаток завершується без виклику ReleaseMutex, CLR звільняє м'ютекс автоматично.

Semaphore

Semaphore схожий на нічний клуб – він має певну місткість, яку забезпечує охоронець. Після заповнення ніхто вже не може увійти в нічний клуб, черга утворюється зовні. Далі, якщо одна людина покидає клуб, один з початку черги може пройти всередину. Конструктор Semaphore приймає мінімум два параметри: число ще доступних місць і загальну місткість нічного клубу.

Semaphore з ємністю, що дорівнює одиниці, подібний Mutex або lock, за винятком того, що він не має потоку-власника. Будь потік може викликати Release для Semaphore, в той час як у випадку з Mutex або lock тільки потік, який захопив ресурс, може його звільнити.

У наступному прикладі по черзі запускаються десять потоків, що виконують виклик Sleep. Semaphore гарантує, що не більше трьох потоків можуть викликати Sleep одночасно:

```
    class SemaphoreTest
    {
      static Semaphore s = new Semaphore(3, 3);  //
Available=3; Capacity=3

      static void Main()
      {
        for (int i = 0; i < 10; i++)
          new Thread(Go).Start();
```

```
    }

    static void Go()
    {
      while (true)
      {
        s.WaitOne();
        // Только 3 потока могут находиться здесь
одновременно
        Thread.Sleep(100);
        s.Release();
      }
    }
  }
```

Потокова безпека

Потокобезпечний код – це код, який не має ніяких невизначеностей за будь-яких сценаріїв багатопоточного виконання. Потокобезпечність досягається насамперед блокуваннями і скороченням можливостей взаємодії між потоками.

Метод, який є потокобезпечним за будь-яких сценаріїв, називається реєнтерабельним. Типи загального призначення рідко є повністю потокобезпечними з наступних причин:

– розробка з урахуванням повної потокової безпеки може бути дуже трудомісткою, особливо якщо тип має багато полів (кожне поле потенційно може брати участь в багатопотоковій взаємодії);

– потокова безпека може позначитися на продуктивності (незалежно від того, чи використовується реально багатопоточність);

– потокобезпечний тип не обов’язково робить всю програму потокобезпечною, а подальша робота щодо її забезпечення може зробити потокобезпечність типу надлишковою.

Тому потокобезпечність реалізується зазвичай тільки там, де вона дійсно потрібна в багатопотоковому сценарії.

Є, однак, кілька обхідних шляхів для отримання великих і складних класів, безпечних в багатопотоковому оточенні. Один з них – пожертвувати деталюванням, блокуючи великі секції коду, аж до цілого об'єкта, і примушуючи до послідовного доступу до нього на високому рівні. Ця тактика також є ключовою при використанні непотокобезпечних об'єктів у потокобезпечному коді.

Інший обхідний шлях полягає в мінімізацію взаємодії потоків через мінімізацію загальних даних. Це чудовий підхід, який використовується в додатках середньої ланки без збереження стану і web-серверах. Оскільки запити безлічі клієнтів можуть прийти одночасно, кожен запит обробляється в своєму власному потоці (у відповідності з архітектурою ASP.NET, Web-служб та Remoting), і це означає, що викликані при цьому методи повинні бути потокобезпечними. Дизайн без використання стану (популярний з причини універсальності) дійсно обмежує взаємодію, так як класи не зберігають дані між запитами. Взаємодія потоків обмежена тільки статичними полями, створеними, наприклад, для кешування часто використовуваних даних, і наданими інфраструктурою сервісами типу аутентифікації і аудиту.

Потокобезпечність і типи. NET Framework

Для перетворення коду в потокобезпечна можна використовувати блокування. Наприклад, майже всі непрімітівние типи.NET Framework не є потокобезпечними, і все ж вони можуть використовуватися в багатопотоковому коді, якщо будь-який доступ до будь-якого об'єкту захищений блокуванням. Ось приклад, в якому два потоки одночасно додають елементи в один і той же список, а потім перераховують всі елементи списку:

```
class ThreadSafe()
{
  static List <string> list = new List <string>();

  static void Main()
  {
```

```
    new Thread(AddItems).Start();
    new Thread(AddItems).Start();
  }

  static void AddItems()
  {
    for (int i = 0; i < 100; i++)
      lock (list)
        Add("Item " + list.Count);

    string[] items;

    lock (list)
      items = list.ToArray();

    foreach (string s in items)
      Console.WriteLine(s);
  }
}
```

У даному випадку блокування відбувається на самому об'єкті-списку, що прекрасно працює в цьому простому сценарії. У випадку двох взаємодіючих списків блокування довелося б робити на одному загальному об'єкті, можливо, виділеному полі, якби один із списків не виявив себе як явний кандидат.

Перерахування. NET-колекцій також не є потокобезпечною операцією, тому якщо інший потік змінює список в процесі перерахування, генерується виключення. Щоб не ставити блокування на весь процес перерахування, в даному прикладі елементи копіюються в масив. Це дозволяє уникнути надмірної блокування в тому випадку, якщо дії з елементами при перерахуванні забирають надто багато часу.

А ось цікаве питання: якби клас `List` був повністю потокобезпечним, що це змінило б? Потенційно дуже небагато! Для прикладу розглянемо додавання елемента до нашого гіпотетичного потокобезпечног списку:

```
if (!myList.Contains(newItem))
```

```
myList.Add(newItem);
```

Незалежно від потокобезпечності власне списку дана конструкція безумовно не потокобезпечна! Заблокований повинен бути весь цей код цілком, щоб запобігти витіснення потоку між перевіркою і додаванням нового елемента. Також блокування має бути використано скрізь, де змінюється список. Наприклад, наступна конструкція має бути загорнута в блокування:

```
myList.Clear();
```

для гарантії, що її виконання не буде перервано. Іншими словами блокування довелося б використовувати точно так само, як з існуючими непотокобезопасними класами. Вбудована потокобезпечність фактично була б марною тратою часу!

Цей момент може бути спірним при написанні замовних компонентів – навіщо потрібна вбудована потокобезпечність, якщо вона, швидше за все, виявиться надлишковою?

Є й контраргумент: зовнішнє блокування об’єкта працює, тільки якщо всі конкуруючі потоки знають про її необхідність і використовують її, а це може бути не так при широкому використанні об’єкта. Найгірше справи йдуть зі статичними полями в публічних типах. Для прикладу уявіть, що статична властивість структури `DateTime DateTime.Now` непотокобезпечна, і два паралельних запити можуть призвести до неправильних результатів або виключення. Єдина можливість виправити становище з використанням зовнішнього блокування – `lock(typeof(DateTime))` при кожному зверненні до `DateTime.Now` спрацювала б, якби всі програмісти погодилися робити так і тільки так. Але це навряд чи можливо, тому що багато хто вважає блокування типу поганим пройомом програмування.

З цієї причини статичні поля структури `DateTime` гарантовано потокобезпечні. Це звичайна поведінка типів у.NET Framework – статичні члени потокобезпечні, нестатичні – ні. Так само слід проектувати і власні типи, щоб уникнути нерозв’язних загадок потокобезпечності.

Крім того, при створенні компонентів для широкого використання ефективна стратегія полягає в тому, щоб програмувати, принаймні, не

перешкоджаючи потокобезпечності. Це означає, що потрібно бути особливо обережним зі статичними типами – неважливо, чи використовуються вони тільки приватно або доступні зовні.

Interrupt і Abort

Заблокований потік може бути передчасно розблоковано двома шляхами:

– за допомогою `Thread.Interrupt`;

– за допомогою `Thread.Abort`.

Це має бути зроблено з іншого потоку; потік, який очікує, безсилий що-небудь зробити в блокованому стані.

Виклик Interrupt для блокованого потоку примусово звільняє його з генерацією винятку `ThreadInterruptedException`, як показано в наступному прикладі:

```
class Program
{
  static void Main()
  {
    Thread t = new Thread(delegate()
      {
        try
        {
          Thread.Sleep(Timeout.Infinite);
        }
        catch(ThreadInterruptedException)
        {
          Console.Write("Error! ");
        }

        Console.WriteLine("Woken!");
      });

    t.Start();
    t.Interrupt();
  }
```

```
}
```

Консольний результат:

```
Error! Woken!
```

Переривання потоку звільняє його тільки від поточного (або наступного) очікування, але не завершує потік (якщо, звичайно, `ThreadInterruptedException` не залишиться необробленим).

Якщо `Interrupt` викликається для неблокованого потоку, потік продовжує виконання до точки наступного блокування, в якій і генерується виключення `ThreadInterruptedException`. Ця поведінка звільняє від необхідності вставляти код перевірки:

```
if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)
  worker.Interrupt();
```

який не є потокобезпечним, так як можуть бути перерваний іншим потоком між оператором `if` і `worker.Interrupt`.

Переривати виконання потоку безпечно, якщо ви точно знаєте, чим зараз зайнятий потік.

Блокований потік також може бути примусово звільнений за допомогою методу `Abort`. Ефект аналогічний `Interrupt`, тільки замість `ThreadInterruptedException` генерується `ThreadAbortException`. Крім того, цей виняток буде повторно згенеровано в кінці блоку `catch`, якщо тільки в блоці catch не буде викликаний `Thread.ResetAbort`. До виклику `Thread.ResetAbort` `ThreadState` буде мати значення `AbortRequested`.

Велика відмінність між `Interrupt` і `Abort` полягає в тому, що відбувається, якщо їх викликати для неблокованого потоку. Якщо `Interrupt` нічого не робить, поки потік не дійде до наступного блокування, то `Abort` генерує виняток безпосередньо в тому місці, де зараз знаходиться потік – може бути, навіть не в вашому коді.

Аудиторні завдання

Завдання № 1

Програмно реалізувати схему, коли один потік, чекає іншого як показано в таблиці.

T0	T1
1) Затримка на 2 секунди. 2) Сигнал в T1 (SignT1).	1) Очікувати T0 (WaitT0). 2) Вивести на екран «OK».

Розв'язання

На рисунку 2.1 представлено схему взаємодії потоків відповідно до завдання.

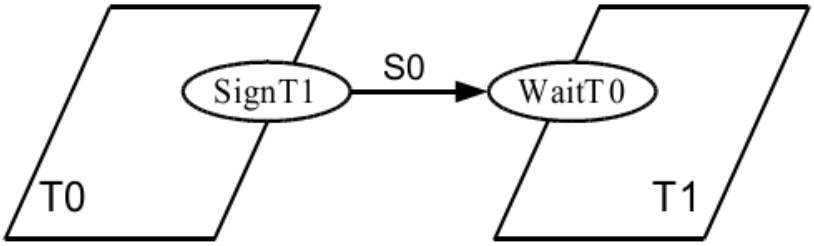


Рисунок 2.1 – Схема взаємодії потоків

У якості сигналу SignT1 використовується вивільнення семафора. Один із варіантів реалізації програми подано нижче:

```
namespace Lab4
{
    class Program
    {
        static Semaphore S0 = new Semaphore(0, 1);

        static void Main(string[] args)
        {
            Thread T0 = new Thread(Func0);
            Thread T1 = new Thread(Func1);
            T0.Start();
            T1.Start();
            T0.Join();
            T1.Join();
        }
```

```
        static void Func0()
        {
            Thread.Sleep(2000);
            S0.Release();  // звільнити семафор
        }

        static void Func1()
        {
            S0.WaitOne();  // блокувати потік
            Console.WriteLine("OK");
        }
    }
}
```

Завдання № 2

Програмно реалізувати схему, коли один потік, чекає іншого як показано в таблиці.

T0	T1	T2
1) Затримка на 2 с. 2) Сигнал в T1 (SignT1). 3) Сигнал в T2 (SignT2).	1) Очікувати T0 (WaitT0). 2) Вивести на екран «OK1».	1) Очікувати T0 (WaitT0). 2) Вивести на екран «OK2».

Розв’язання

На рисунку 2.2 представлено схему взаємодії потоків відповідно до завдання.

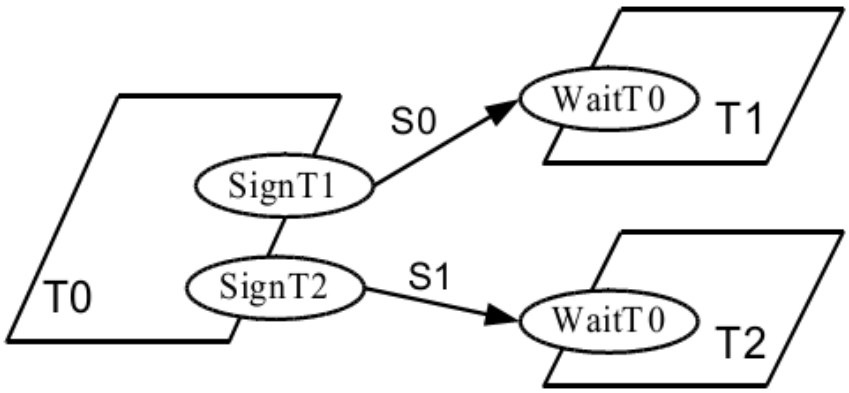


Рисунок 2.2 – Схема взаємодії потоків

Для вирішення даного завдання використаємо два окремі семафори.

```
class Program
 {
     static Semaphore S0 = new Semaphore(0, 1);
     static Semaphore S1 = new Semaphore(0, 1);

     static void Main(string[] args)
     {
         Thread T0 = new Thread(Func0);
         Thread T1 = new Thread(Func1);
         Thread T2 = new Thread(Func2);
         T0.Start();
         T1.Start();
         T2.Start();
         T0.Join();
         T1.Join();
         T2.Join();
     }

     static void Func0()
     {
         Thread.Sleep(2000);
         S0.Release();
         S1.Release();
     }

     static void Func1()
     {
         S0.WaitOne();
         Console.WriteLine("OK1");
     }

     static void Func2()
     {
         S1.WaitOne();
         Console.WriteLine("OK2");
```

```
        }
    }
```

Завдання № 3

Попереднє завдання реалізувати з використанням множинних семафорів.

Розв’язання

При створенні об’єкта семафора задається початкова кількість запитів та ємність семафора. Використаємо один семафор для керування різними потоками.

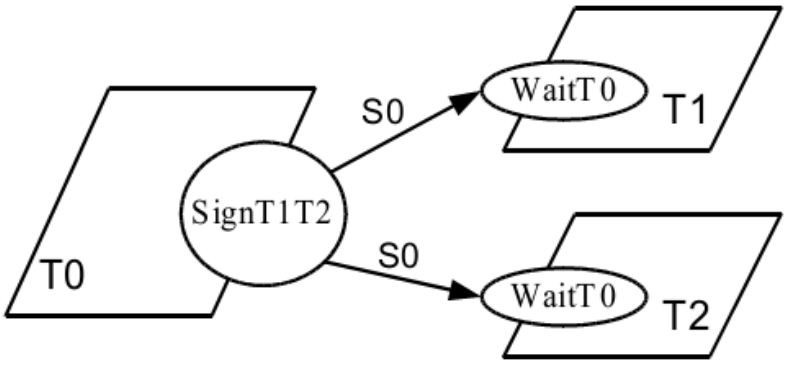


Рисунок 2.3 – Схема взаємодії потоків для множинного семафора

```
class Program
{
    static Semaphore S0 = new Semaphore(0, 2);

    static void Main(string[] args)
    {
        Thread T0 = new Thread(Func0);
        Thread T1 = new Thread(Func1);
        Thread T2 = new Thread(Func2);
        T0.Start();
        T1.Start();
        T2.Start();
        T0.Join();
        T1.Join();
        T2.Join();
    }
```

```
        static void Func0()
        {
            Thread.Sleep(2000);
            S0.Release(2);
        }

        static void Func1()
        {
            S0.WaitOne();
            Console.WriteLine("OK1");
        }

        static void Func2()
        {
            S0.WaitOne();
            Console.WriteLine("OK2");
        }
    }
```

Завдання № 4

Задано граф передування (рисунок 2.4). Використовуючи семафори, виконати синхронізацію потоків. Змінні x, y та z є глобальними для всіх потоків та проініціалізовані наступним чином: x = 4, y = 2, z = 5.

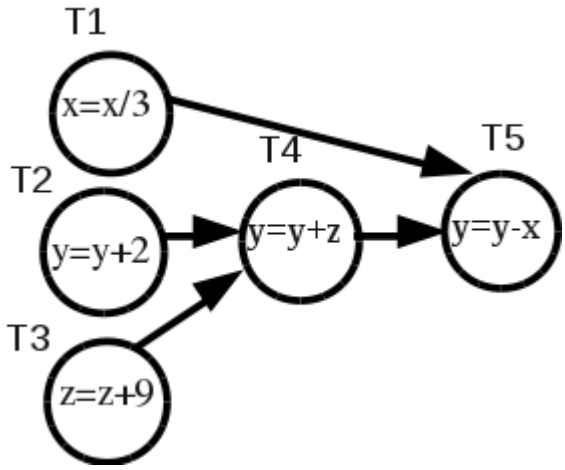


Рисунок 2.4 – Граф передування

Розв’язання

Для визначення необхідної кількості семафорів та послідовності блокування потоків перетворимо граф передування у схему взаємодії потоків (рисунок 2.5).

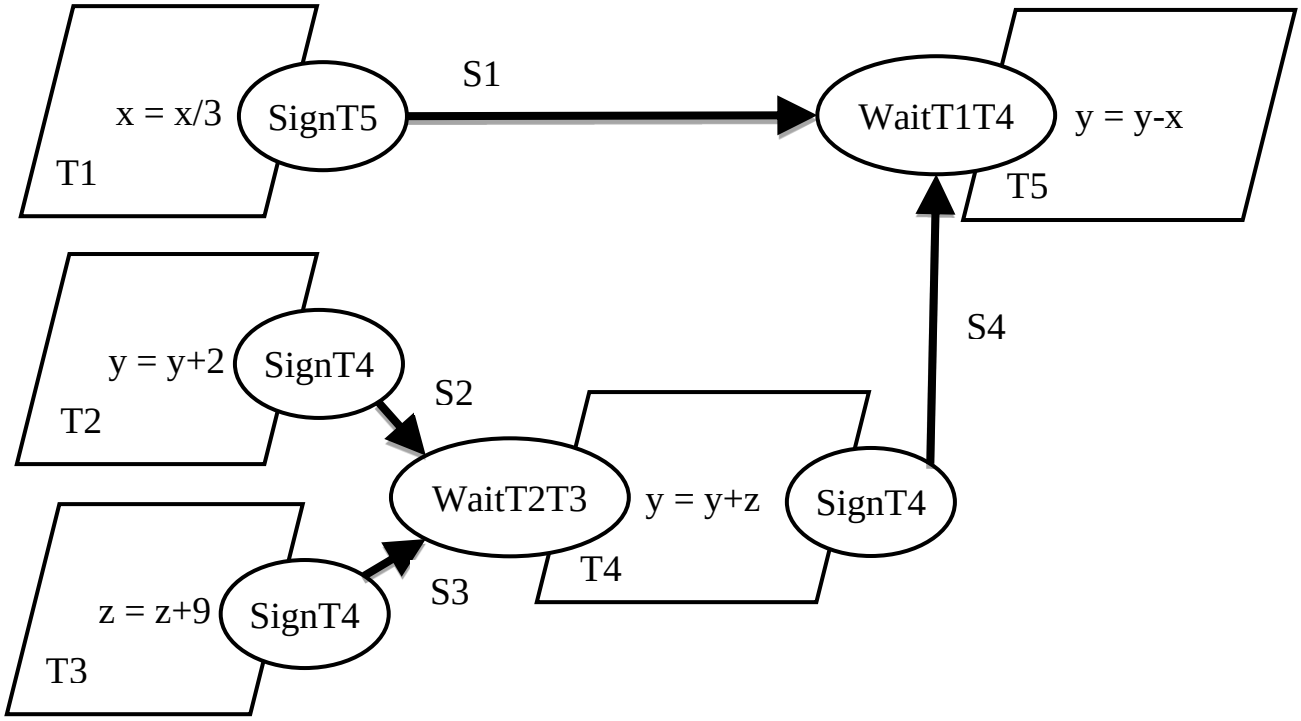


Рисунок 2.5 – Схема взаємодії потоків

У якості аргументів конструкторів потоків використаємо лямбда-вирази (локальні анонімні функції). Програмний код подано нижче.

```
class Program
{
    static double x = 4, y = 2, z = 5;
    static Semaphore S1 = new Semaphore(0, 1);
    static Semaphore S2 = new Semaphore(0, 1);
    static Semaphore S3 = new Semaphore(0, 1);
    static Semaphore S4 = new Semaphore(0, 1);

    static void Main(string[] args)
    {
        Thread T1 = new Thread(() =>
        {
            x /= 3;
            S1.Release();
```

```
            });
            Thread T2 = new Thread(() =>
            {
                y += 2;
                S2.Release();
            });
            Thread T3 = new Thread(() =>
            {
                z += 9;
                S3.Release();
            });
            Thread T4 = new Thread(() =>
            {
                S2.WaitOne();
                S3.WaitOne();
                y += z;
                S4.Release();

            });
            Thread T5 = new Thread(() =>
            {
                S1.WaitOne();
                S4.WaitOne();
                y -= x;
            });
            Console.WriteLine("x = {0}; y = {1}; z = {2}", x,
y, z);
            T1.Start();
            T2.Start();
            T3.Start();
            T4.Start();
            T5.Start();
            T1.Join();
            T2.Join();
            T3.Join();
            T4.Join();
            T5.Join();
            Console.WriteLine("x = {0}; y = {1}; z = {2}", x,
y, z);
```

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

Слід зазначити, що дану задачу можна вирішити без застосування семафорів, більш простим методом – очікувати завершення відповідних потоків за допомогою методу `Join()`.

```
    class Program
    {
        static double x = 4, y = 2, z = 5;
        static Thread T1, T2, T3, T4, T5;

        static void Main(string[] args)
        {
            Console.WriteLine("x = {0}; y = {1}; z = {2}", x,
y, z);
            T1 = new Thread(() => { x /= 3; });
            T1.Start();
            T2 = new Thread(() => { y += 2; });
            T2.Start();
            T3 = new Thread(() => { z += 9; });
            T3.Start();
            T4 = new Thread(() =>
            {
                T2.Join();
                T3.Join();
                y += z;
            });
            T4.Start();
            T5 = new Thread(() =>
            {
                T1.Join();
                T4.Join();
                y -= x;
            });
            T5.Start();
            T5.Join();
            Console.WriteLine("x = {0}; y = {1}; z = {2}", x,
y, z);
```

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

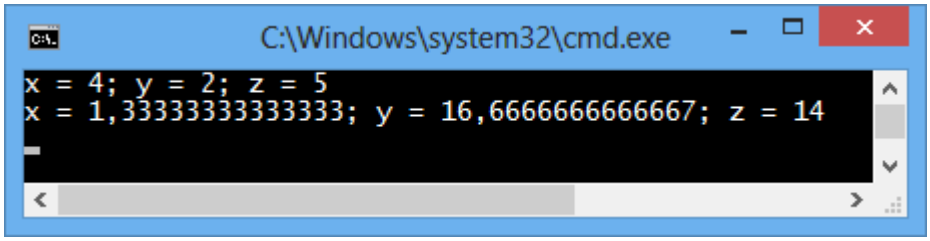


Рисунок 2.6 – Результат обчислень

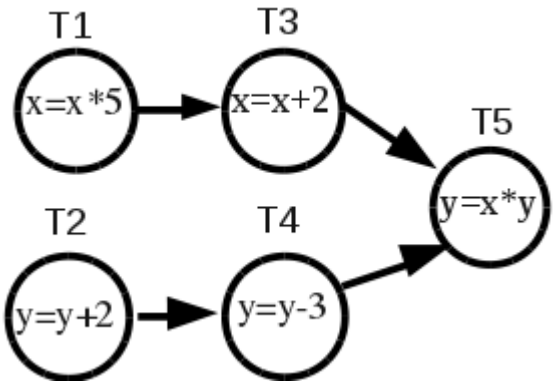
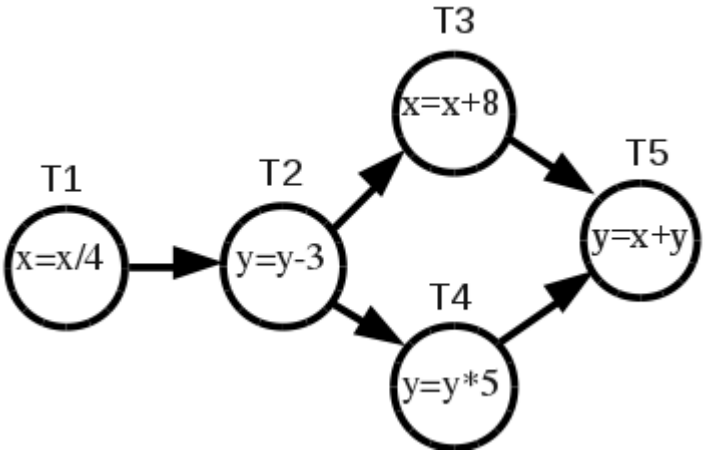
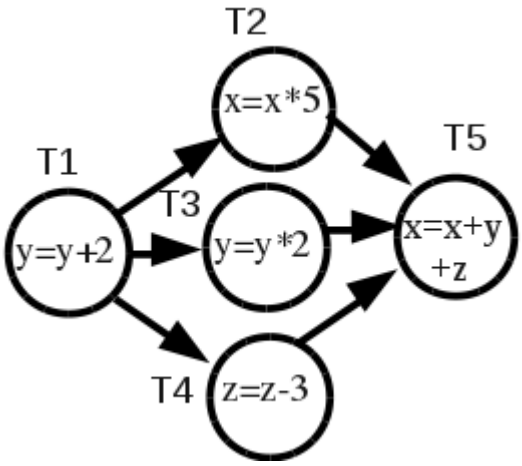
Індивідуальні завдання

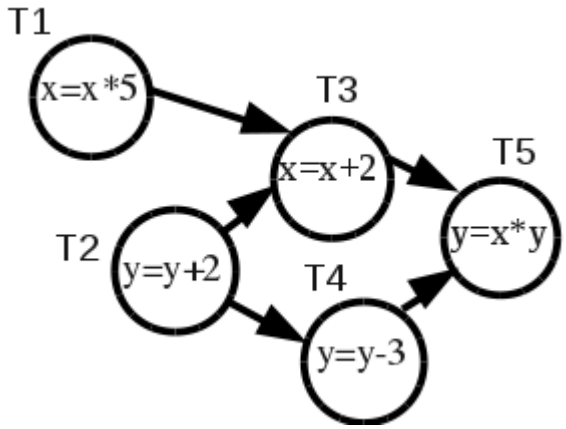
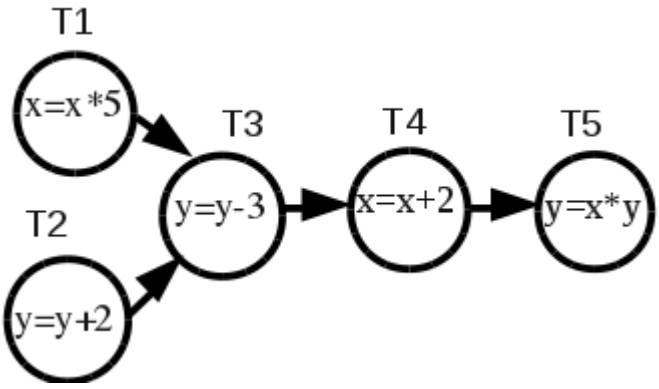
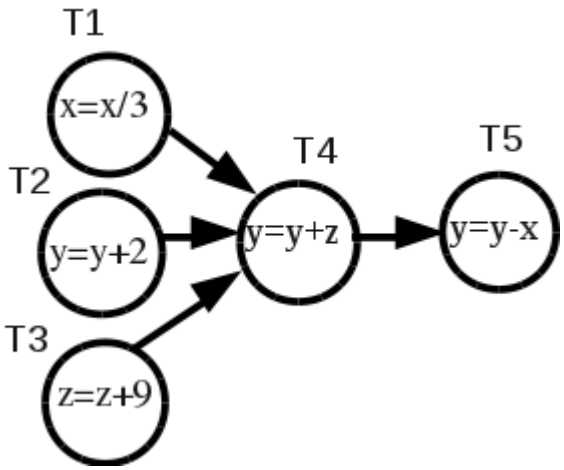
Відповідно до варіанту та обраного рівня складності виконати наступні завдання.

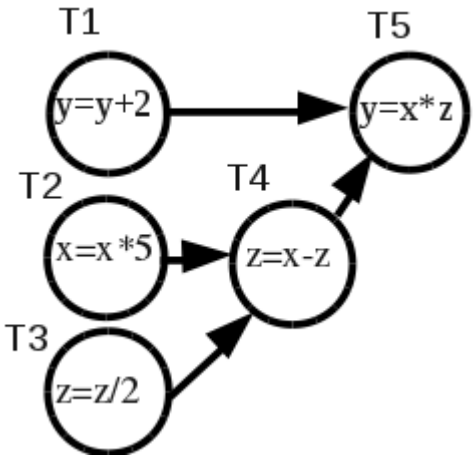
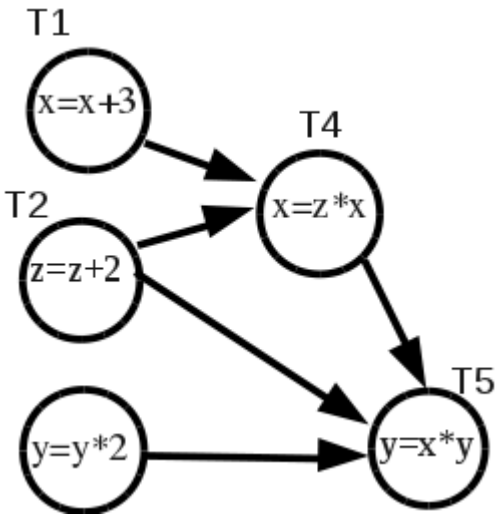
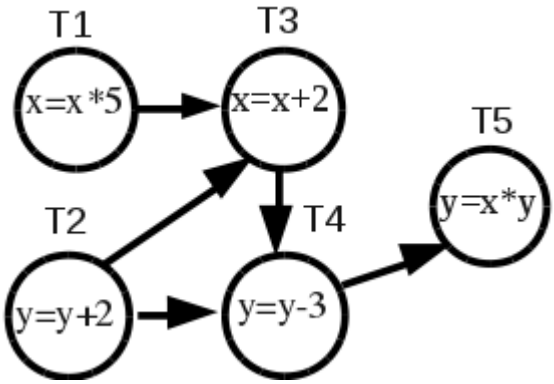
№	Рівень складності		
	Початковий	Базовий	Високий
1	Відкомпілювати та протестувати аудиторні завдання	Завдання № 1	Завдання № 1
2	–	Завдання № 2	Завдання № 2
3	–	–	Завдання № 3

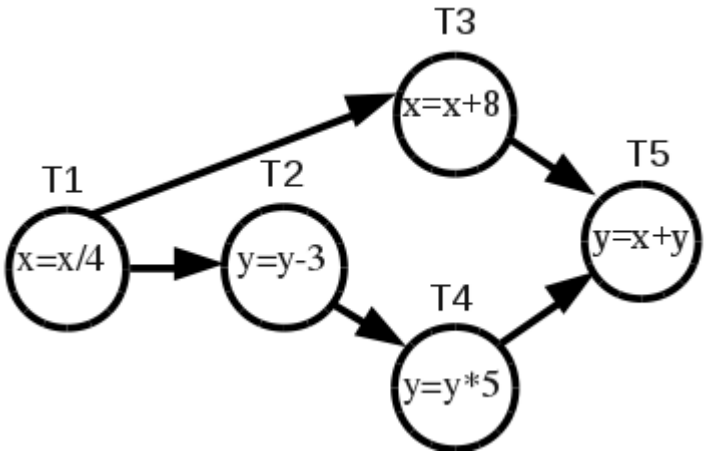
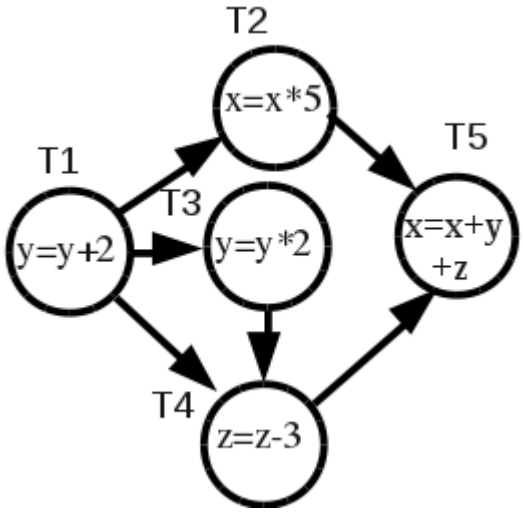
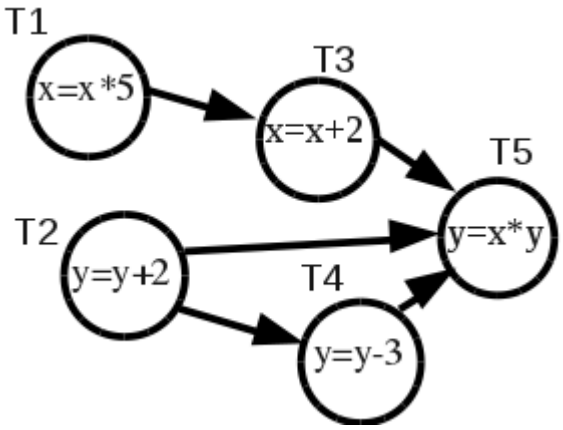
Завдання № 1

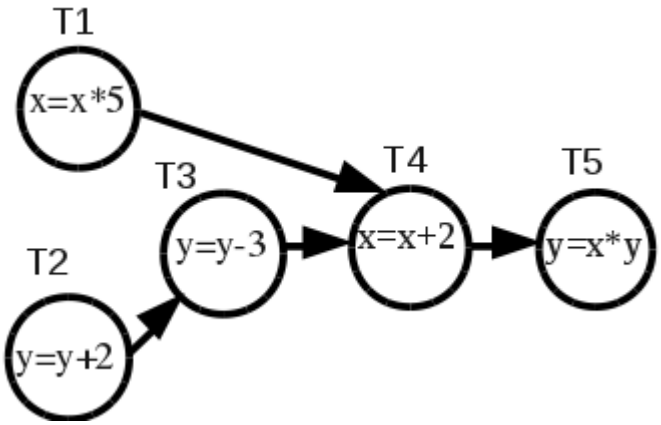
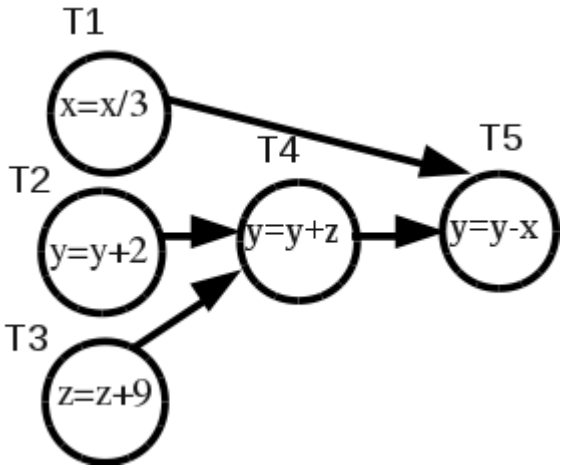
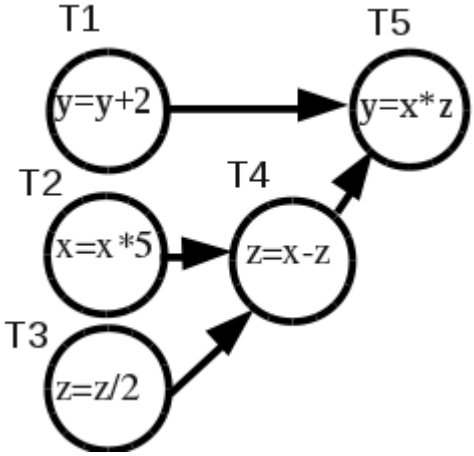
Задано граф роботи потоків. Кожна задача в графі виконується в окремому потоці. Використовуючи семафори, виконати синхронізацію потоків. Змінні x, y та z є глобальними для всіх потоків та проініціалізовані наступним чином: x = номер варіанту, y = 5, z = 4.

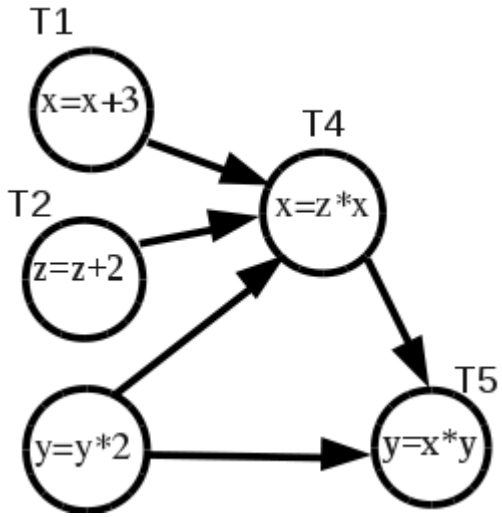
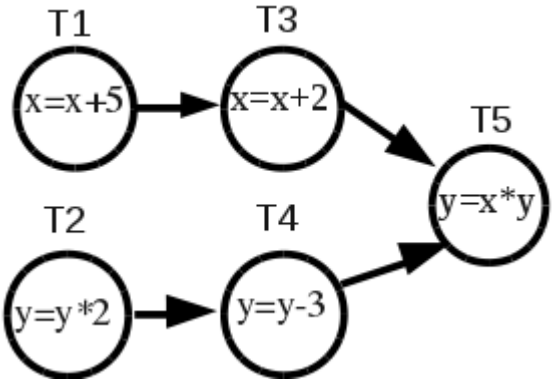
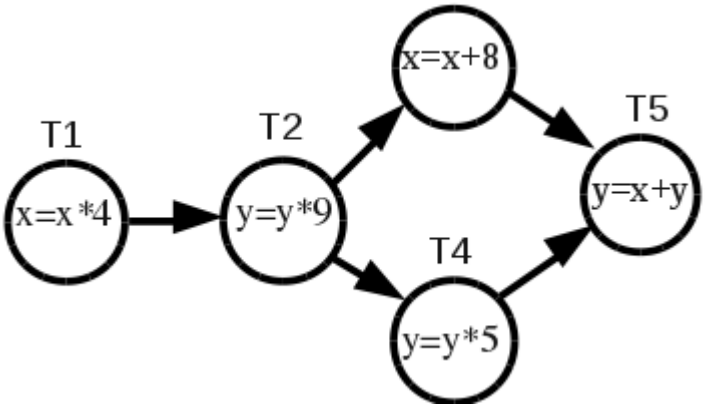
№ варіанта	Граф роботи потоків
1	
2	
3	

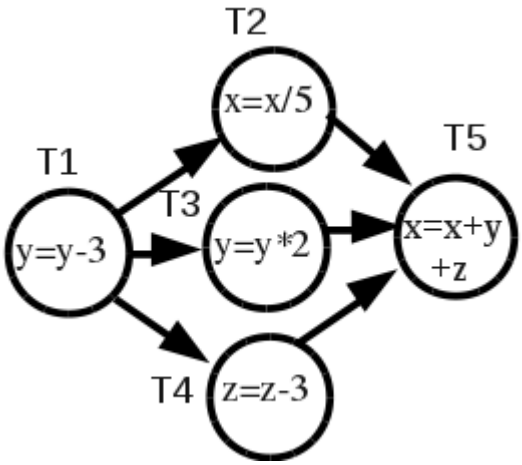
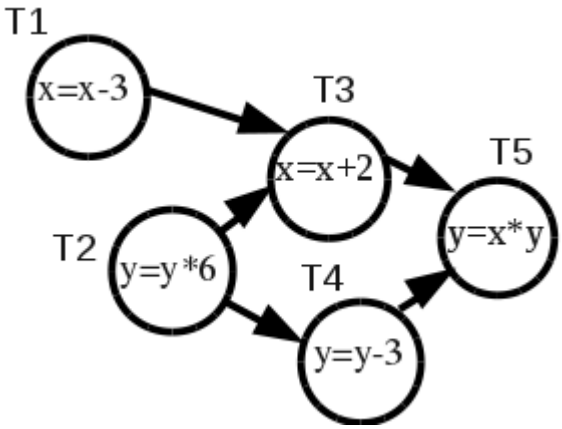
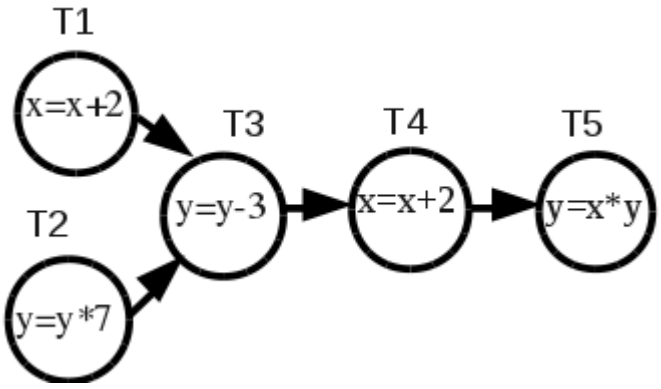
№ варіанта	Граф роботи потоків
4	
5	
6	

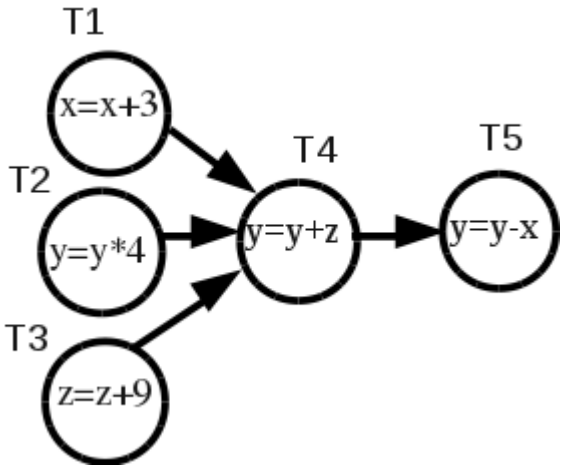
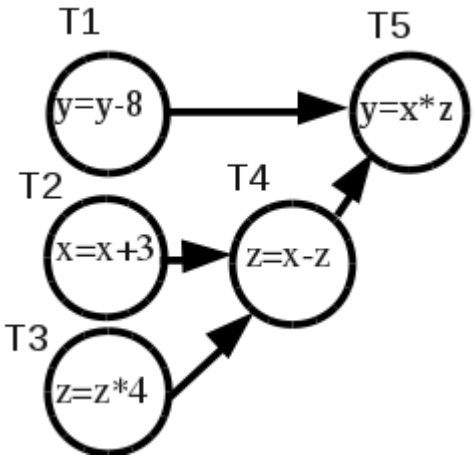
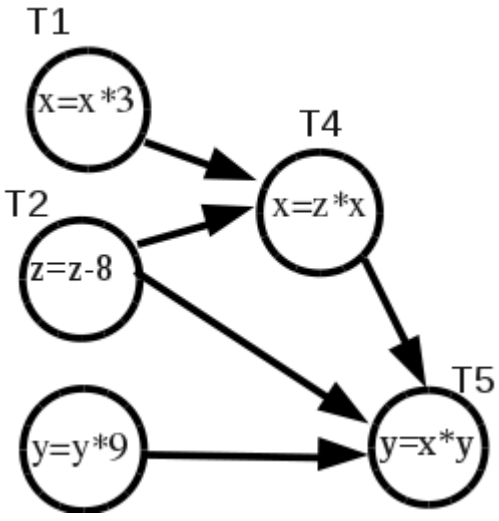
№ варіанта	Граф роботи потоків
7	
8	
9	

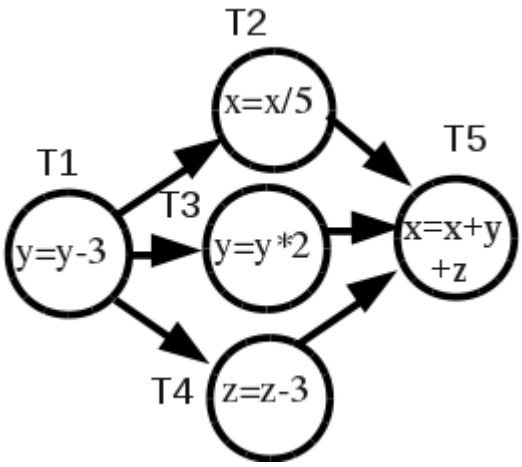
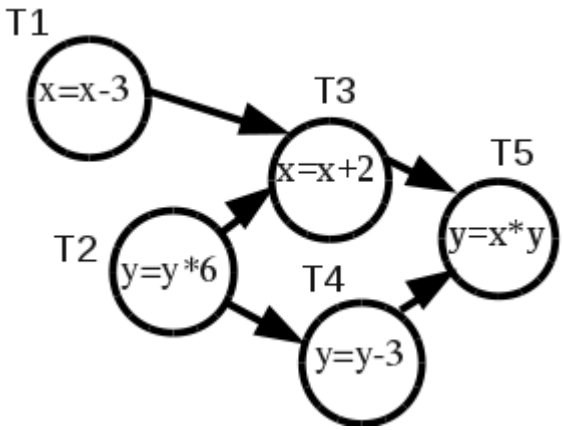
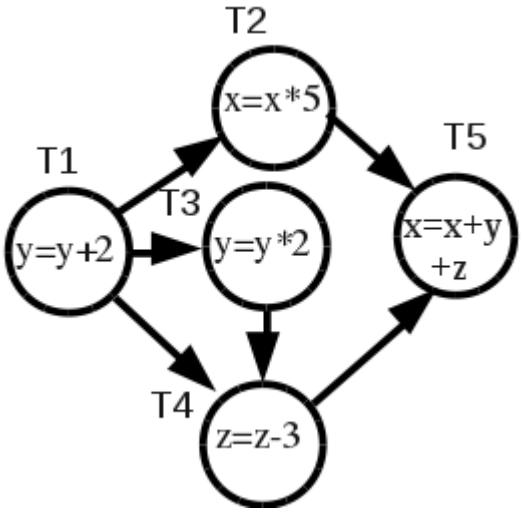
№ варіанта	Граф роботи потоків
10	
11	
12	

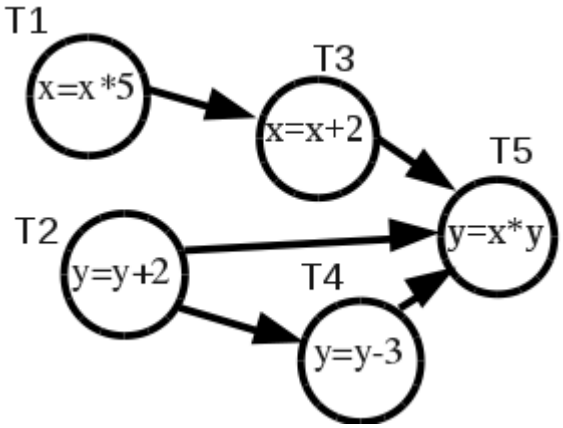
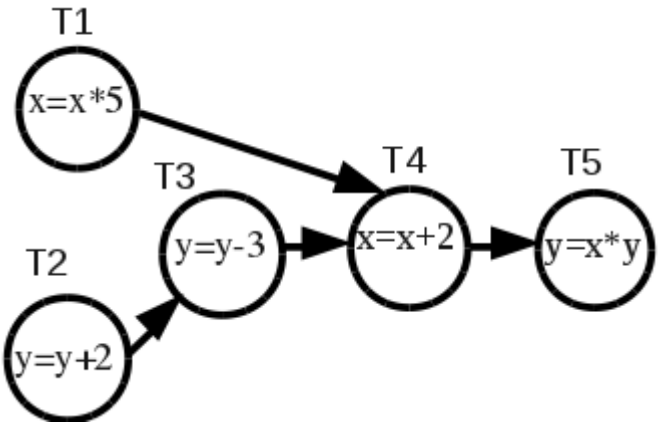
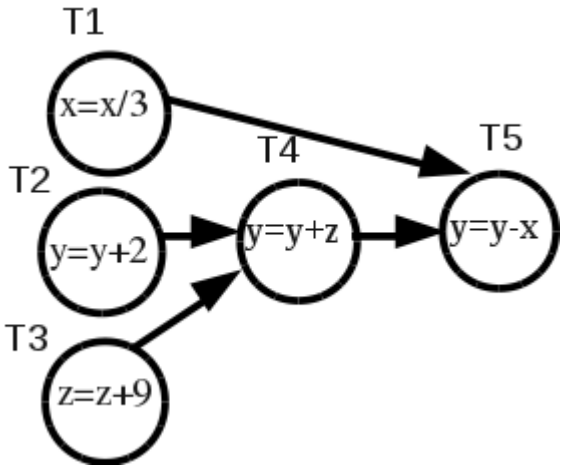
№ варіанта	Граф роботи потоків
13	
14	
15	

№ варіанта	Граф роботи потоків
16	
17	
18	

№ варіанта	Граф роботи потоків
19	
20	
21	

№ варіанта	Граф роботи потоків
22	
23	
24	

№ варіанта	Граф роботи потоків
25	
26	
27	

№ варіанта	Граф роботи потоків
28	
29	
30	

Завдання № 2

Написати паралельну програму, яка двома потоками обчислює векторну операцію, де X, Y, Q – вектори розміром N = 30. Вектор X ініціалізується одиницями, а Y – числами від 1 до N. Ініціалізацію векторів виконати паралельно в різних потоках.

№ варіанта	Векторна операція
1	$Q = 3X + 5Y$
2	$Q = 7X + 2Y$
3	$Q = X + 2Y$
4	$Q = 2X + 3Y$
5	$Q = 5X + 2Y$
6	$Q = X + 9Y$
7	$Q = 10X + 7Y$
8	$Q = 12X + Y$
9	$Q = 2X + 8Y$
10	$Q = 9X + 21Y$
11	$Q = 3X + 17Y$
12	$Q = 18X + 7Y$
13	$Q = 15X + Y$
14	$Q = 6X + 11Y$
15	$Q = 2X + 5Y$
16	$Q = 11X + 11Y$
17	$Q = 14X + 5Y$
18	$Q = 19X + 2Y$
19	$Q = 9X + 11Y$
20	$Q = 17X + 3Y$
21	$Q = 8X + 5Y$
22	$Q = 7X + 9Y$
23	$Q = 8X + 12Y$
24	$Q = 7X + 16Y$
25	$Q = 12X + 17Y$
26	$Q = 18X + 6Y$

№ варіанта	Векторна операція
27	$Q = 13X + Y$
28	$Q = 6X + 12Y$
29	$Q = 2X + 4Y$
30	$Q = 17X + 7Y$

Завдання № 3

Попереднє завдання реалізувати для кількості потоків рівній трьом: на початку програми двома потоками виконується ініціалізація, а потім трьома потоками виконуються обчислення.

Контрольні питання

1. Які засоби синхронізації потоків та процесів існують у C#?
2. Чому при роботі з потоками не рекомендуються використовувати м’ютекси?
3. Яку інформацію містить контекст потоку?
4. У чому відмінність роботи з м’ютексами, моніторами та семафорами?
5. Якими засобами можна визначити прискорення виконання програми при застосуванні паралельнх обчислень?
6. Що описує граф передування?
7. Запропонувати алгоритм, який гарантуватиме запуск тільки одного екземпляра програми.
8. Сформулювати закон Амдала.

ЛАБОРАТОРНА РОБОТА № 3

Тема: засоби синхронізації потоків на основі повідомлень в C#.

Мета: виконати паралельні обчислення з використанням декількох потоків та засобів синхронізації на основі повідомлень.

Теоретичні відомості

Стан потоку

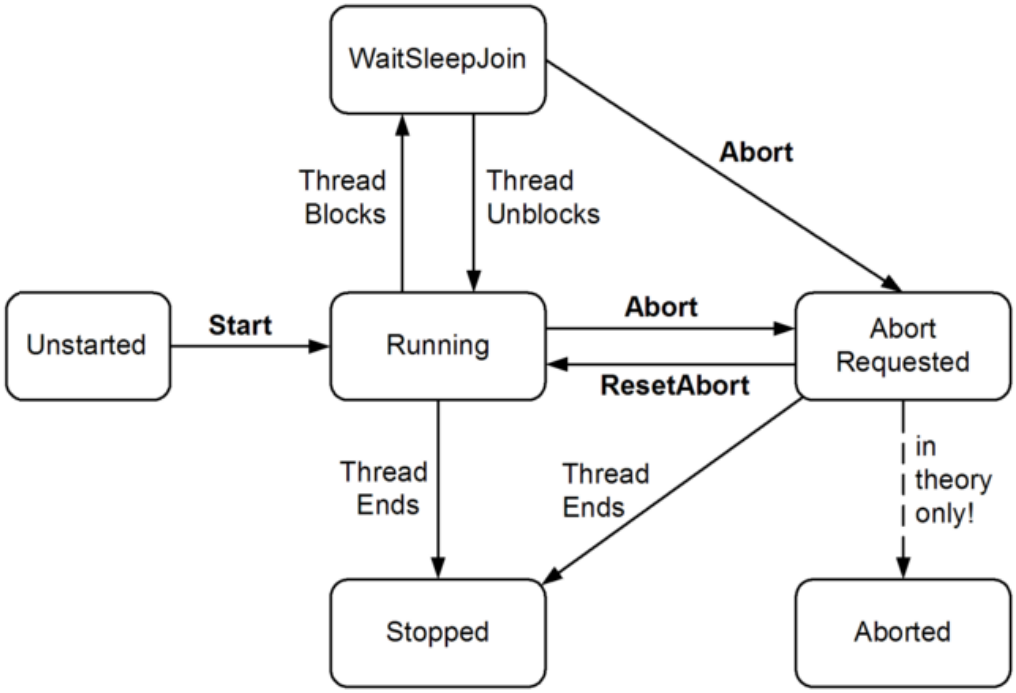


Рисунок 3.1 – Граф станів потоку

Запросити стан потоку можна за допомогою його властивості `ThreadState`. Рисунок 3.1 демонструє «рівні» перерахування `ThreadState`. `ThreadState` спроектований не досить зручно, це комбінація трьох рівнів станів з використанням бітових прапорів, члени перерахування в межах кожного рівня є взаємовиключними. Ось ці три рівні:

– `running/blocking/aborting` (як показано на рисунку 3.1);

– `Background/foreground` (ThreadState.Background);

– призупинення з використанням нерекомендованого методу `Suspend` (`ThreadState.SuspendRequested` і `ThreadState.Suspended`).

У результаті ThreadState – бітова комбінація з членів кожного рівня! Ось приклади ThreadState:

- Unstarted;
- Running;
- WaitSleepJoin;
- Background, Unstarted;
- SuspendRequested, Background, WaitSleepJoin.

Перерахування також має два члени, які ніколи не використовуються в потоковій реалізації CLR: StopRequested і Aborted.

І це ще не все. ThreadState.Running має значення 0, так що наступний вираз працювати не буде:

```
if ((t.ThreadState & ThreadState.Running) > 0)...
```

і замість цього потрібно перевіряти наявність ThreadState.Running шляхом виключення або в якості альтернативи, використовувати властивість IsAlive. Властивість IsAlive, однак, може вам не підійти, так як повертає true, якщо потік блокований або припинений (false повертається тільки до того, як потік почався, і після того, як він завершиться).

ThreadState неоціненний при налагодженні або пошуку вузьких місць в продуктивності. Однак він погано підходить для координації дій декількох потоків, оскільки не існує механізму, який би дозволив протестувати ThreadState і потім діяти на основі цієї інформації без потенційного зміни ThreadState в цей проміжок часу.

Wait Handles

Оператор lock (Monitor.Enter / Monitor.Exit) – один з прикладів конструкцій синхронізації потоків. Lock є найбільш підходящим засобом для організації монопольного доступу до ресурсу або секції коду, але існують завдання синхронізації (типу подачі сигналу початку роботи потоку, який очікує), для яких lock буде не самим адекватним і зручним засобом.

У Win32 API є багатий набір конструкцій синхронізації, і вони доступні в. NET Framework у вигляді класів `EventWaitHandle`, `Mutex` і `Semaphore`. Деякі з них практичніше інших: `Mutex`, наприклад, здебільшого дублює можливості `lock`, в той час як `EventWaitHandle` надає унікальні можливості сигналізації.

Всі три класи засновані на абстрактному класі `WaitHandle`, але досить сильно відрізняються за поведінкою. Одна із загальних особливостей – це здатність іменування, що робить можливою роботу з потоками не тільки одного, а й різних процесів.

`EventWaitHandle` має два похідних класи: `AutoResetEvent` і `ManualResetEvent` (що не мають ніякого відношення до подій і делегатів C#). Обом класам доступні всі функціональні можливості базового класу, єдина відмінність полягає у виклику конструктора базового класу з різними параметрами.

У частині продуктивності, всі `WaitHandle` зазвичай виконуються в межах декількох мікросекунд. Це рідко має значення з урахуванням контексту, в якому вони застосовуються.

`AutoResetEvent` – найбільш часто використовуваний `WaitHandle`-клас і основна конструкція синхронізації, поряд з `lock`.

AutoResetEvent

`AutoResetEvent` дуже схожий на турнікет – один квиток дозволяє пройти одній людині. Приставка «auto» в назві ставиться для того, щоб показати, що відкритий турнікет автоматично закривається або «скидається» після того, як дозволяє кому-небудь пройти. Потік блокується біля турнікету викликом `WaitOne` (чекати (wait) у даного (one) турнікета, поки він не відкриється), а квиток вставляється викликом методу `Set`. Якщо кілька потоків викликають `WaitOne`, за турнікетом утворюється черга. Будь-який потік може «вставити» квиток – іншими словами, будь-який (неблокований) потік, що має доступ до об’єкта `AutoResetEvent`, може викликати `Set`, щоб пропустити один блокований потік.

Якщо Set викликається, коли немає потоків, які очікують, хендл буде знаходитися у відкритому стані, поки який-небудь потік не викличе WaitOne. Ця особливість допомагає уникнути змагань між потоком, який підходить до турнікету, і потоком, який вставляє квиток. Однак багаторазовий виклик Set для вільного турнікета не дозволяє пропустити за раз цілу юрбу – зможе пройти лише одна людина, всі інші квитки будуть витрачені даремно.

WaitOne приймає необов'язковий параметр timeout. Метод повертає false, якщо очікування закінчується при досягненні таймаута, а не з отриманням сигналу. WaitOne також можна навчити виходити з поточного контексту синхронізації для продовження очікування (якщо використовується режим з автоматичним блокуванням) щоб уникнути надмірного блокування.

Метод Reset забезпечує закриття відкритого турнікета, без жодних очікувань і блокувань.

AutoResetEvent може бути створений двома шляхами. По-перше, за допомогою свого конструктора :

```
EventWaitHandle wh = new AutoResetEvent(false);
```

Якщо аргумент конструктора true, метод Set буде викликаний автоматично відразу після створення об'єкта.

Інший метод полягає у створенні об'єкта базового класу, EventWaitHandle:

```
EventWaitHandle wh = new EventWaitHandle(false,
EventResetMode.Auto);
```

Конструктор EventWaitHandle також може використовуватися для створення об'єкта ManualResetEvent (якщо задати в якості параметра EventResetMode.Manual).

Метод Close потрібно викликати відразу ж, як тільки WaitHandle стане не потрібен для звільнення ресурсів операційної системи. Однак якщо WaitHandle використовується протягом усього життя програми, цей крок можна опустити, так як він буде виконаний автоматично при руйнуванні домену програми.

У наступному прикладі запускається робочий потік, який просто чекає сигналу від іншого потоку.

```
class BasicWaitHandle
{
  static EventWaitHandle wh = new AutoResetEvent(false);

  static void Main()
  {
    new Thread(Waiter).Start();
    Thread.Sleep(1000);                  // Подождать
некоторое время...
    wh.Set();                             // OK – можно
разбудить
  }

  static void Waiter()
  {
    Console.WriteLine("Ожидание...");
    wh.WaitOne();                         // Ожидать сигнала
    Console.WriteLine("Получили сигнал");
  }
}
```

Консольний результат:

```
Ожидание... (пауза) Получили сигнал
```

Створення міжпроцесних EventWaitHandle

Конструктор `EventWaitHandle` також дозволяє створювати іменовані `EventWaitHandle`, які можуть працювати через кордони процесів. Ім’я задається звичайної рядком і може бути будь-яким. Якщо задається ім’я, яке вже використовується на комп’ютері, то повертається посилання на існуючий `EventWaitHandle`, в іншому випадку операційна система створює новий. Ось приклад:

```
EventWaitHandle wh = new EventWaitHandle(false,
EventResetMode.Auto, "MyCompany.MyApp.SomeName");
```

Виконавши цей код, два додатки отримали б можливість сигналізувати один одному з будь-якого потоку обох процесів.

Принцип «Отримайте і розпишіться»

Припустимо, ми хочемо виконувати завдання у фоновому режимі без витрат на створення щоразу нового потоку для нового завдання. Цієї мети можна досягти, використовуючи єдиний робочий потік з постійним циклом, що очікують появу завдання. Отримавши завдання, він приступає до її виконання. Після закінчення виконання потік знову переходить в режим очікування. Це звичайний багатопоточний сценарій. Поряд з позбавленням від накладних витрат на створення потоків ми отримуємо послідовне виконання завдань, усуваючи потенційні проблеми взаємодії між потоками і надмірне споживання ресурсів.

Однак потрібно вирішити, що робити, якщо робочий потік ще зайнятий виконанням попередньої задачі, а вже з'явилася наступна. Можна, наприклад, блокувати виконання, поки не завершена попередня завдання. Це можна реалізувати, використовуючи два об'єкти типу `AutoResetEvent` – `ready`, який відкривається (встановлюється шляхом виклику методу `Set`) робочим потоком, коли він готовий до роботи, і `go`, який відкривається викликаючим потоком, коли з'являється нова задача. У наступному прикладі для демонстрації завдання використовується просте строкове поле (оголошене з ключовим словом `volatile` для гарантії того, що обидва потоки будуть бачити його в одному і тому ж стані):

```
class AcknowledgedWaitHandle
{
  static EventWaitHandle ready = new AutoResetEvent(false);
  static EventWaitHandle go = new AutoResetEvent(false);
  static volatile string task;

  static void Main()
  {
    new Thread(Work).Start();
```

```
        // Сигнализируем рабочему потоку 5 раз
        for (int i = 1; i <= 5; i++)
        {
          ready.WaitOne();  // Сначала ждем, когда рабочий
поток будет готов
          task = "a".PadRight(i, 'h'); // Назначаем задачу
          go.Set();         // Говорим рабочему потоку, что
можно начинать
        }

        // Сообщаем о необходимости завершения рабочего потока,
        // используя null-строку
        ready.WaitOne();
        task = null;
        go.Set();
      }

      static void Work()
      {
        while (true)
        {
          ready.Set();                    // Сообщаем о
готовности
          go.WaitOne();                   // Ожидаем сигнала
начать...

          if (task == null)
            return;                       // Элегантно
завершаемся

          Console.WriteLine(task);
        }
      }
    }
```

Консольний результат:

```
ah
ahh
ahhh
```

```
ahhhh
```

Зверніть увагу, що для завершення робочого потоку використовується завдання зі значенням `null`. Для робочого потоку в даному випадку успішно можна було б використовувати виклик `Interrupt` або `Abort` – але тільки відразу після `ready.WaitOne`, так як в цьому випадку нам відомо стан робочого потоку – безпосередньо перед `go.WaitOne` або на цьому виклику – і можна уникнути ускладнень при перериванні невідомого коду. Використання `Interrupt` або `Abort` також вимагало б додати обробку винятків у робочому потоці.

Черга «Постачальник/Споживач (Producer/Consumer)»

Ще один поширений сценарій роботи з потоками – фонова обробка завдань з черги. Це називається чергою Постачальник/Споживач: Постачальник ставить завдання в чергу, Споживач витягує завдання з черги в робочому потоці. Дуже схоже на попередній приклад, за винятком того, що викликаючий потік не блокується, якщо робітник все ще зайнятий попереднім завданням.

Черга Постачальник/Споживач може масштабуватися: споживачів може бути кілька, кожен обслуговує одну і ту ж чергу, але в своєму потоці. Це хороший спосіб використання переваг багатопроцесорних систем, але воно все ж обмежує кількість паралельно працюючих потоків, щоб уникнути витрат на переключення контексту і боротьби за ресурси.

У наступному прикладі один `AutoResetEvent` використовується для сигналізації робочому потоку, який зупиняється, тільки якщо йому більше нічого виконувати (черга порожня). Для черзі використовується `Queue<>`, доступ до нього захищений блокуванням для забезпечення потокової безпеки. Робочий потік завершується, якщо зустрічає `null`-задачу:

```
using System;
using System.Threading;
using System.Collections.Generic;
```

```
    class ProducerConsumerQueue : IDisposable
    {
      EventWaitHandle wh = new AutoResetEvent(false);
      Thread worker;
      object locker = new object();
      Queue<string> tasks = new Queue<string>();

      public ProducerConsumerQueue()
      {
        worker = new Thread(Work);
        worker.Start();
      }

      public void EnqueueTask(string task)
      {
        lock (locker)
          tasks.Enqueue(task);

        wh.Set();
      }

      public void Dispose()
      {
        EnqueueTask(null);      // Сигнал Потребителю на
завершение
        worker.Join();          // Ожидание завершения
Потребителя
        wh.Close();             // Освобождение ресурсов
      }

      void Work()
      {
        while (true)
        {
          string task = null;

          lock (locker)
          {
            if (tasks.Count > 0)
```

```
            {
              task = tasks.Dequeue();
              if (task == null)
                return;
            }
          }

          if (task != null)
          {
            Console.WriteLine("Выполняется задача: " + task);
            Thread.Sleep(1000); // симуляция работы...
          }
          else
            wh.WaitOne();       // Больше задач нет, ждем
сигнала...
        }
      }
    }
```

А це код тестування черги:

```
    class Test
    {
      static void Main()
      {
        using(ProducerConsumerQueue q = new
ProducerConsumerQueue())
        {
          q.EnqueueTask("Привет!");

          for (int i = 0; i < 10; i++)
            q.EnqueueTask("Сообщение " + i);

          q.EnqueueTask("Пока!");
        }
        // Выход из using приводит к вызову Dispose, который
ставит
        // в очередь null-задачу и ожидает, пока Потребитель не
завершится.
      }
```

```
}
```

Консольний результат:

```
Выполняется задача: Привет!
Выполняется задача: Сообщение 0
Выполняется задача: Сообщение 1
Выполняется задача: Сообщение 2
Выполняется задача: Сообщение 3
Выполняется задача: Сообщение 4
Выполняется задача: Сообщение 5
Выполняется задача: Сообщение 6
Выполняется задача: Сообщение 7
Выполняется задача: Сообщение 8
Выполняется задача: Сообщение 9
Выполняется задача: Пока!
```

Зверніть увагу, що `WaitHandle` явно закривається, коли для `ProducerConsumerQueue` викликається `Dispose()`, так як в перебігу життя програми можливе створення і руйнування багатьох об'єктів типу `ProducerConsumerQueue`.

ManualResetEvent

`ManualResetEvent` – це різновид `AutoResetEvent`. Відмінність полягає в тому, що він не скидається автоматично, після того як потік проходить через `WaitOne`, і діє як шлагбаум – `Set` відкриває його, дозволяючи пройти будь-якій кількості потоків, що викликали `WaitOne`. `Reset` закриває шлагбаум, потенційно накопичуючи чергу очікуючих наступного відкриття.

`ManualResetEvent` може використовуватися для сигналізації про завершення якої-небудь операції або ініціалізації потоку і готовності до виконання роботи.

WaitAny, WaitAll і SignalAndWait

Крім `Set` і `WaitOne`, є ще кілька статичних методів класу `WaitHandle` для більш вимогливих завдань синхронізації. Методи `WaitAny`, `WaitAll` і `SignalAndWait` полегшують взаємодію декількох `WaitHandle`, можливо різних типів.

`SignalAndWait`, можливо, самий корисний метод. Він у межах єдиної атомарної операції викликає `WaitOne` для одного `WaitHandle`, і `Set` – для іншого. Класичним варіантом використання цього методу є використання з парою `EventWaitHandle` для підготовки зустрічі двох потоків в потрібній точці в потрібний час. Підійдуть і `AutoResetEvent` і `ManualResetEvent`. Перший потік робить наступне:

```
WaitHandle.SignalAndWait(wh1, wh2);
```

в той час як другий потік - навпаки:

```
WaitHandle.SignalAndWait(wh2, wh1);
```

`WaitHandle.WaitAny` очікує звільнення одного (кожного) `WaitHandle` з переданого йому списку, `WaitHandle.WaitAll` очікує звільнення відразу всіх переданих йому `WaitHandle`. Використовуючи аналогію з турнікетом метро, ці методи організовують загальну чергу одночасно для всіх турнікетів – з проходженням через перший турнікет, який відкриється (`WaitAny`) або з очікуванням, поки вони не відкриються усі (`WaitAll`).

Аудиторні завдання

Завдання № 1

Написати паралельну програму, яка обчислює значення виразу $F = x_1*x_2+(x_3+x_1)*x_4$. Кожна арифметична операція має бути виконана в окремому потоці. Змінні ініціалізуються в тому ж потоці, в якому вони вперше використовуються. Ініціалізувати змінні наступними значеннями: $x_1 = 10$, $x_2 = 20$, $x_3 = 30$, $x_4 = 40$.

Розв’язання

```
    using System;
    using System.Threading;

    namespace Lab3_1
    {
        class Program
        {
            static EventWaitHandle wh1 = new
AutoResetEvent(false),
                                    wh2 = new
AutoResetEvent(false),
                                    wh3 = new
AutoResetEvent(false);

            static private int x1, x2, x3, x4;
            static int A, B;

            static void Main(string[] args)
            {
                var T0 = new Thread(Func0);
                var T1 = new Thread(Func1);
                var T2 = new Thread(Func2);
                var T3 = new Thread(Func3);
                T0.Start();
                T1.Start();
                T2.Start();
                T3.Start();
                T3.Join();
                Console.ReadKey();
            }

            static void Func0()
            {
                x1 = 10;
                x2 = 20;
                A = x1*x2;
                wh1.Set();
            }
```

```
        static void Func1()
        {
            x3 = 30;
            B = x3 + x1;
            wh2.Set();
        }

        static void Func2()
        {
            x4 = 40;
            wh2.WaitOne();
            B *= x4;
            wh3.Set();
        }

        static void Func3()
        {
            wh1.WaitOne();
            wh3.WaitOne();
            Console.WriteLine("F = {0}", A + B);
        }
    }
}
```

Індивідуальні завдання

Відповідно до варіанта та обраного рівня складності виконати наступні завдання.

№	Рівень складності		
	Початковий	Базовий	Високий
1	Відкомпілювати та протестувати аудиторні завдання	Завдання № 1	Завдання № 2

Завдання № 1

Написати паралельну програму, яка обчислює значення виразу F за варіантом. Кожна арифметична операція має бути виконана в

окремому потоці. Змінні ініціалізуються в тому ж потоці, в якому вони вперше використовуються. Ініціалізувати змінні наступними значеннями: $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$, $x_5 = 5$, $x_6 = 6$.

№ варіанта	Вираз F
1	$x_1*x_2+x_3+x_4*x_5+x_6$
2	$x_1+x_2*x_3+x_4+x_5+x_6$
3	$x_1*x_2+x_3*x_4+x_5+x_6$
4	$x_1+x_2+x_3+(x_4+x_5)*x_6$
5	$(x_1+x_2)*(x_3+x_4)+x_5*x_6$
6	$(x_1+x_2+x_3)*(x_4+x_5+x_6)$
7	$(x_1+x_2)*(x_3+x_4)+x_5+x_6$
8	$(x_1+x_2)*(x_3+x_4)*x_5+x_6$
9	$(x_1+x_2)*(x_3+x_4)*(x_5+x_6)$
10	$(x_1+x_2)*(x_3+x_4+x_5+x_6)$
11	$(x_1*x_2+x_3)*x_4+x_5+x_6$
12	$(x_1+x_2)+x_3*(x_4+x_5+x_6)$
13	$x_1*x_2+(x_3+x_4+x_5)*x_6$
14	$(x_1+x_2*x_3+x_4+x_5)*x_6$
15	$x_1*(x_2+x_3+x_4+x_5)*x_6$
16	$(x_1*x_2+x_3)*(x_4+x_5+x_6)$
17	$x_1+((x_2+x_3)*x_4+x_5)*x_6$
18	$x_1*x_2*x_3+x_4*x_5*x_6$
19	$x_1*x_2+(x_3+x_4+x_5)*x_6$
20	$(x_1*x_2+x_3)*(x_4+x_5)*x_6$
21	$(x_1+x_2)*x_3*x_4+x_5*x_6$
22	$(x_1+x_2+x_3)*x_4*(x_5+x_6)$
23	$(x_1*x_2+(x_3+x_4)+x_5)*x_6$
24	$x_1+(x_2+x_3)+(x_4+x_5)*x_6$
25	$(x_1+x_2)+(x_3*x_4+x_5*x_6)$
26	$x_1*x_2+(x_3+x_4+x_5)*x_6$
27	$(x_1+x_2*x_3+x_4+x_5)*x_6$
28	$x_1*(x_2+x_3+x_4+x_5)*x_6$

№ варіанта	Вираз F
29	$(x_1*x_2+x_3)*(x_4+x_5+x_6)$
30	$x_1+((x_2+x_3)*x_4+x_5)*x_6$

Завдання № 2

Написати паралельну програму, яка обчислює матричний вираз за варіантом. Всі матриці є квадратними, мають розмірність N і задаються випадковими цілими числами у діапазоні [–10; 10]. Програма повинна вирішувати такі завдання:

1) Кількість робочих потоків P, якими виконують паралельні обчислення, має задаватися користувачем;

2) Потрібно побудувати графік залежності часу виконання програми від розмірності матриць (N = 10^3, 10^4, 10^5 …) при однопоточному режимі та кількості потоків, яка відповідає кількості логічних ядер персонального комп’ютера.

№ варіанта	Вираз, що необхідно обчислити
1	$(M_1+M_2)*(M_3+M_4)+M_5*M_6$
2	$(M_1+M_2+M_3)*(M_4+M_5+M_6)$
3	$(M_1+M_2)*(M_3+M_4)+M_5+M_6$
4	$(M_1+M_2)*(M_3+M_4)*M_5+M_6$
5	$(M_1+M_2)*(M_3+M_4)*(M_5+M_6)$
6	$(M_1+M_2)*(M_3+M_4+M_5+M_6)$
7	$(M_1*M_2+M_3)*M_4+M_5+M_6$
8	$(M_1+M_2)+M_3*(M_4+M_5+M_6)$
9	$M_1*M_2+(M_3+M_4+M_5)*M_6$
10	$(M_1+M_2*M_3+M_4+M_5)*M_6$
11	$M_1*(M_2+M_3+M_4+M_5)*M_6$
12	$(M_1*M_2+M_3)*(M_4+M_5+M_6)$
13	$M_1+((M_2+M_3)*M_4+M_5)*M_6$
14	$M_1*M_2*M_3+M_4*M_5*M_6$
15	$M_1*M_2+(M_3+M_4+M_5)*M_6$

№ варіанта	Вираз, що необхідно обчислити
16	$(M_1*M_2+M_3)*(M_4+M_5)*M_6$
17	$(M_1+M_2)*M_3*M_4+M_5*M_6$
18	$(M_1+M_2+M_3)*M_4*(M_5+M_6)$
19	$(M_1*M_2+(M_3+M_4)+M_5)*M_6$
20	$M_1+(M_2+M_3)+(M_4+M_5)*M_6$
21	$(M_1+M_2)+(M_3*M_4+M_5*M_6)$
22	$M_1*M_2+(M_3+M_4+M_5)*M_6$
23	$(M_1+M_2*M_3+M_4+M_5)*M_6$
24	$M_1*(M_2+M_3+M_4+M_5)*M_6$
25	$(M_1*M_2+M_3)*(M_4+M_5+M_6)$
26	$M_1+((M_2+M_3)*M_4+M_5)*M_6$
27	$M_1*M_2+M_3+M_4+M_5*M_6$
28	$M_1+M_2*M_3+M_4+M_5+M_6$
29	$M_1*M_2+M_3*M_4+M_5+M_6$
30	$M_1+M_2+M_3+(M_4+M_5)*M_6$

Контрольні питання

1. Для чого призначений клас EventWaitHandle?
2. Які класи .NET Framework можна застосувати, щоб оцінити час виконання програми у багатопоточному режимі?
3. Наведіть паралельний алгоритм додавання двох векторів.
4. Наведіть паралельний алгоритм множення двох матриць.
5. У чому полягає сутність принципу організації параельного алгоритму «Producer/Consumer»?
6. Обгрунтуйте або спростуйте можливість створення двох різних паралельних алгоритмів, які одну й ту ж саму задачу вирішують за приблизно рівний час.
7. Які програмні бібілотеки використовуються для створення паралельних програм на основі передач повідомлень?
8. У чому відміннсть класів AutoResetEvent та ManualResetEvent?

ЛАБОРАТОРНА РОБОТА № 4

Тема: основи розподілених обчислень на базі моделі клієнт-сервер, програмування мережних додатків на мові C# із використанням сокетів.

Мета: розробити клієнтську та серверну частину додатку для реалізації розподілених обчислень.

Теоретичні відомості

Мережні сокети та порти

Мережний сокет (network socket) – це кінцева точка двухсторонньої комунікаційної взаємодії комп'ютерів в мережі.

Інтерфейсний сокет (socket API) – програмний інтерфейс, який надає операційна система, та дозволяє програмним додаткам керувати та використовувати мережні сокети.

Адреса сокету (socket address) – поєднання IP-адреси та номера порту. Базуючись на цій адресі інтернет-сокети доставляють відповідному програмному процесу чи потоку пакети даних, що надходять з мережі.

Слід розрізняти клієнтські і серверні сокети. Клієнтські сокети можна порівняти з кінцевими апаратами телефонної мережі, а серверні – з комутаторами. Клієнтський додаток, наприклад браузер, використовує тільки клієнтські сокети, а серверний, наприклад веб-сервер, якому браузер посилає запити – як клієнтські, так і серверні сокети.

Інтерфейс сокетів вперше з'явився в BSD Unix. Програмний інтерфейс сокетів описаний в стандарті POSIX.1 і в тій чи іншій мірі підтримується всіма сучасними операційними системами. Кожен процес може створити прослуховуючий сокет або серверний сокет і прив'язати його до якого-небудь порту операційної системи (Рисунок 4.1). Процес, що прослуховує канал, зазвичай знаходиться в циклі очікування, тобто прокидається при появі нового з'єднання. При цьому

зберігається можливість перевірити наявність з'єднань на даний момент, встановити тайм-аут для операції.

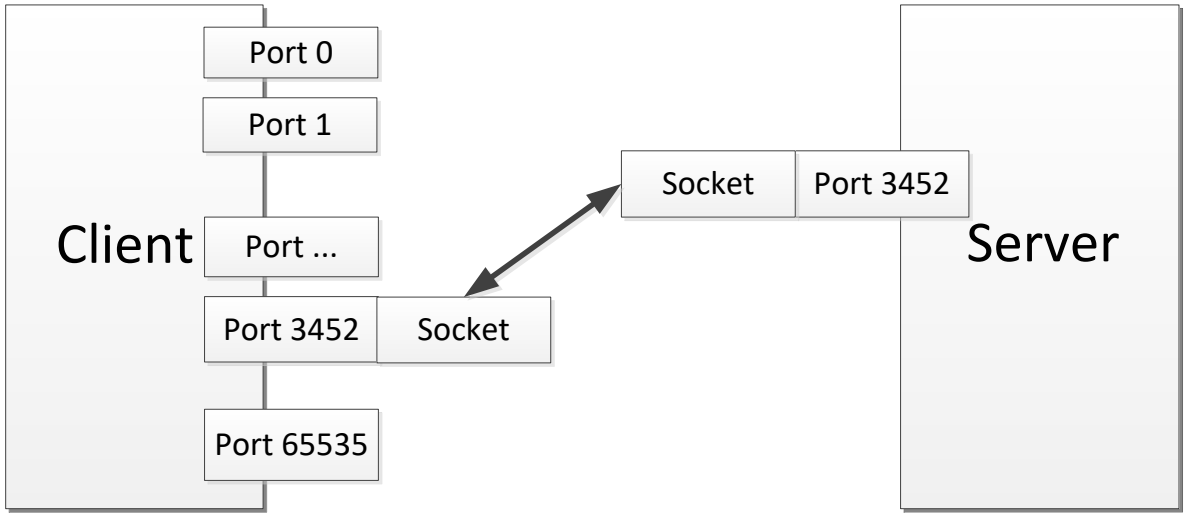


Рисунок 4.1 – Загальна схема роботи сокетів

Кожен сокет має свою адресу. Операційні системи сімейства UNIX можуть підтримувати багато типів адрес, але обов'язковими є INET-адреса і UNIX-адреса. Якщо прив'язати сокет за UNIX-адресою, то буде створений спеціальний файл або файл сокету по заданому шляху, через який зможуть «спілкуватися» будь-які локальні процеси шляхом читання/запису з нього. Сокети типу INET доступні з мережі і вимагають виділення номера порту. Зазвичай клієнт явно під'єднується до слухача, після чого будь-яке читання або запис через його файловий дескриптор буде передаватися між ним і сервером. Загальновідомі сокети представляють собою зручний механізм апріорного прив'язування адреси сокету до якого завгодно стандартного сервісу. Наприклад, процес-сервер для програми Telnet жорстко зв'язаний з конкретним сокетом (Рисунок 4.2). Адреса сокету може бути зарезервована для доступу до процедури перегляду, яка могла б вказувати сокет, крізь який можна було б отримати новоутворені послуги.

В протоколах TCP/UDP порт – це системний ресурс, який має номер та виділяється програмі для зв'язку з додатками, що виконуються на інших мережних хостах (Таблиця 4.1). Порт може бути зайнятий тільки однією програмою і в цей час не може

використовуватися іншою. Всі програми для зв'язку між собою за допомогою мережі використовують порти (до 65536 портів).

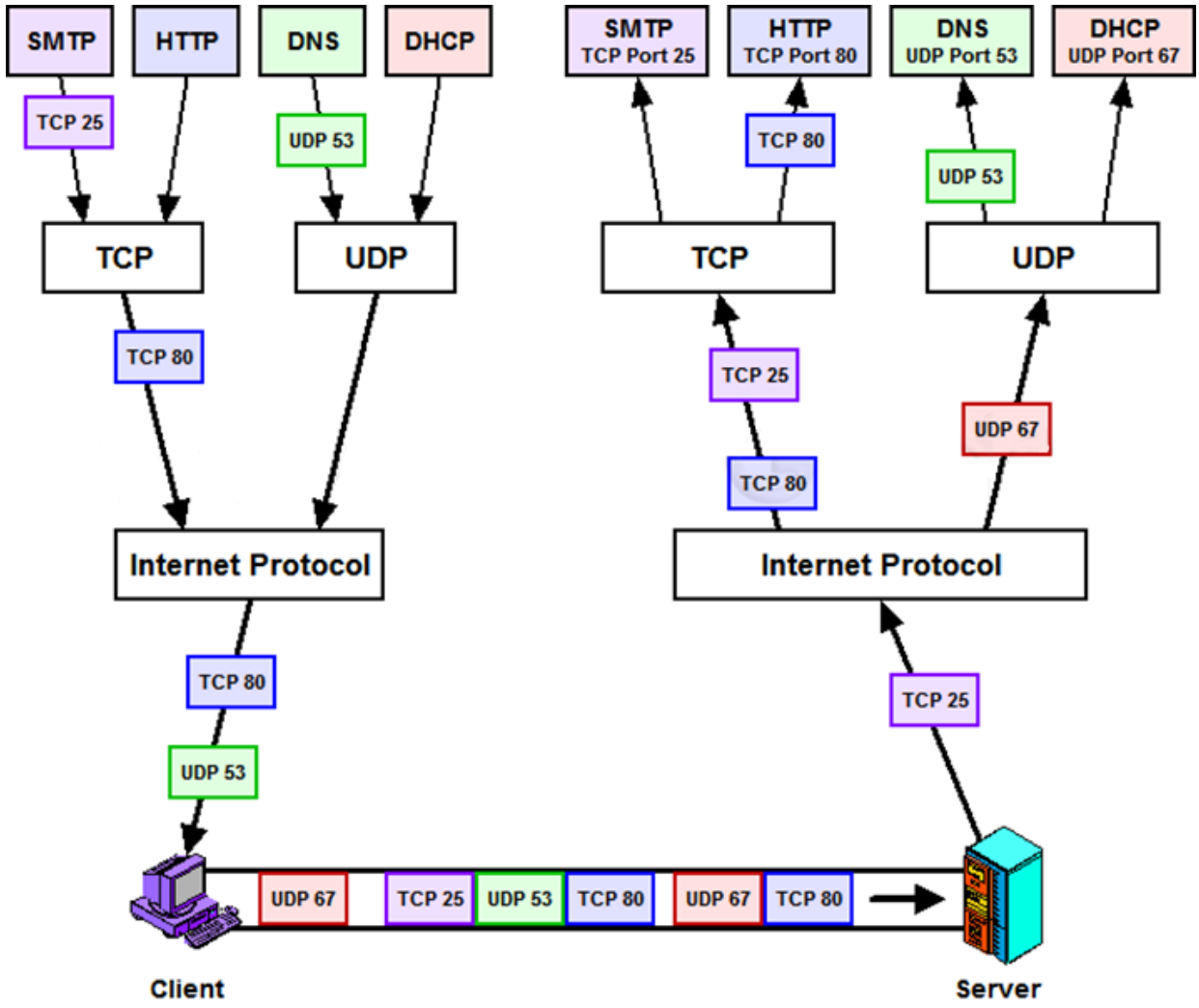


Рисунок 4.2 – Пояснення схеми роботи портів

Таблиця 4.1 – Деякі загальновідомі порти

Порт	TCP	UDP	Опис
0	√	√	Зарезервовано
20	√		Протокол FTP (дані)
21	√		Протокол FTP (команди)
22	√	√	Протокол SSH
23	√	√	Протокол Telnet
53		√	Протокол DNS
80	√	√	Протокол HTTP
443	√		Протокол HTTPS
6969	√		Клієнт BitTorrent
33434	√	√	Службова утиліта Traceroute

Стек протоколів TCP/IP. Протоколи TCP та UDP

TCP/IP – це абревіатура терміну Transmission Control Protocol / Internet Protocol (Протокол керування передачею / міжмережевий протокол). Фактично TCP/IP не один протокол, а декілька (таблиця 5.2). Саме тому ви часто чуєте, як його називають набором, або комплектом протоколів, серед яких TCP і IP – два основні. Фактично TCP/IP представляє цей базовий набір протоколів, відповідальний за розбивання вихідного повідомлення на пакети (TCP), доставку пакетів на вузол адресата(IP) і збирання (відновлення) вихідного повідомлення з пакетів (TCP).

Таблиця 4.2 – Співвідношення моделі OSI та стеку протоколів TCP/IP

<table>
<tr><th>OSI Model</th><th colspan="2">TCP/IP Model</th></tr>
<tr><td>Application</td><td rowspan="3">Application</td><td rowspan="3">Telnet, SMTP, POP3, FTP, HTTP, SNMP, DNS, SSH, …</td></tr>
<tr><td>Presentation</td></tr>
<tr><td>Session</td></tr>
<tr><td>Transport</td><td>Transport</td><td>TCP, UDP</td></tr>
<tr><td>Network</td><td>Internet</td><td>IP, ICMP, ARP, DHCP</td></tr>
<tr><td>Data Link</td><td rowspan="2">Network Access</td><td rowspan="2">Ethernet, ADSL, …</td></tr>
<tr><td>Physical</td></tr>
</table>

Прикладний рівень

Протоколи прикладного рівня TCP/IP визначають процедури стосовно організації взаємодії прикладних процесів (програм) різних мережних комп’ютерів і форми подання інформації за такої взаємодії. По ознакам взаємодії прикладних процесів виділяють два типи прикладного програмного забезпечення: програма-клієнт та програма-сервер. Протоколи прикладного рівня зорієнтовано на конкретні прикладні завдання. Серед традиційних послуг, котрі забезпечують протоколи прикладного рівня із сімейства TCP/IP, сьогодні найпопулярнішими є електронна пошта – протоколи SMTP та POP3,

передавання файлів – FTP та TFTP, емуляції віддаленого терміналу – TELNET тощо.

В інтернеті активно використовуються послуги, які базуються на технології WWW, яка ґрунтується на протоколі передавання гіпертексту HTTP. Сьогодні є популярні послуги пакетної IP-телефонії на базі стандартів IETF, яку стосуються спеціальних протоколів прикладного, транспортного і мережного рівнів, наприклад сигналізації SIP, передавання в режимі реального часу RTP та RTCP, резервування ресурсів RSVP, рекомендацій ITU H.323 тощо.

Транспортний рівень

Протоколи транспортного рівня TCP/IP надають транспортні послуги прикладним процесам. Основними протоколами транспортного рівня TCP/IP є протокол керування передаванням TCP (Transmission Control Protocol) і протокол користувальницьких дейтаграм UDP (User Datagram Protocol). Транспортні послуги цих протоколів суттєво відрізняються. Протокол UDP доставляє дейтаграми без установлення з'єднання. При цьому він не гарантує їхнього доставляння. Протокол TCP забезпечує надійне доставляння байтових потоків (сегментів) із попереднім встановленням транспортного дуплексного з'єднання (віртуального каналу) між модулями TCP мережних комп'ютерів. Для розв'язання транспортних завдань протоколи TCP та UDP в перебігу передавання даних формують і додають до даних свої заголовки обсягом 20 байт та 8 байт відповідно.

Кожен прикладний процес взаємодіє з модулем транспортного рівня TCP або UDP через окремий порт, що дозволяє при взаємодії систем однозначно ідентифікувати прикладні процеси. Ці порти нумеруються починаючи з нуля. При передаванні запиту прикладної програми клієнта до прикладної програми сервера транспортний модуль, формуючи дейтаграму чи сегмент, вказує номери портів програмних модулів прикладних протоколів сервера й клієнта. З цією метою в заголовку пакета протоколу транспортного рівня виділено два

поля – «порт одержувача» і «порт відправника», обсягом по 2 байти. Номери портів TCP та UDP до прикладних протоколів сервера стандартизовано IETF. Для цього надано номери в діапазоні від 1 до 1023. Наприклад, програмний модуль TCP сервера взаємодіє з модулем протоколу HTTP через порт з номером 80. Взаємодія модуля TCP чи UDP клієнта з будь-яким модулем прикладного протоколу відбувається через порт, якому надається вільний номер, за значенням більший ніж 1023.

Мережний рівень

Протоколи мережного рівня TCP/IP забезпечують взаємодію поміж мережами різної архітектури тощо. Основним протоколом мережного рівня технології TCP/IP є міжмережний протокол IP та його допоміжні протоколи: адресний протокол ARP; реверсний адресний протокол RARP (Reverse ARP); протокол діагностичних повідомлень ICMP (Internet Control Message Protocol), який надсилає повідомлення вузлам мережі про помилки на маршруті, які виникають при передаванні пакетів тощо.

Головне завдання міжмережного протоколу IP – це маршрутизація пакетів даних поміж різнотипними комп'ютерними мережами. Для розв'язання цього завдання протокол IP підтримує IP-адресацію мереж та вузлів, використовує таблицю маршрутизації пакетів, виконує, за необхідності, фрагментацію та дефрагментацію цих пакетів.

Функціонування мережного рівня також забезпечує низка протоколів динамічної маршрутизації RIP, OSPF, які динамічно формують маршрути таблиці маршрутизації за алгоритмами вектора VDA (Vector Distance Algorithm) і стану зв'язку LSA (Link State Algorithm) відповідно, протоколів політики зовнішньої маршрутизації EGP (Exterior Gateway Protocol), BGP (Border Gateway Protocol) тощо.

Засоби мережевого рівня забезпечують доставку даних між пристроями в складових мережі, а саме комп'ютерами, маршрутизаторами і т.д. Однак не слід забувати, що на одному вузлі може функціонувати паралельно декілька програм, яким потрібен

доступ до мережі. Отже, дані всередині комп'ютерної системи повинні розподілятися між програмами. Тому, при передачі даних по мережі недостатньо просто адресувати конкретний вузол. Необхідно також ідентифікувати програму-одержувача, що неможливо здійснити засобами мережевого рівня.

Іншою серйозною проблемою протоколів мережевого рівня є відсутність засобів, що дозволяють передавати великі масиви даних. Коли вихідні дані перевищують максимально допустимий розмір пакета мережного рівня, то ці дані повинні бути розбиті на порції, кожна з яких передається в мережу окремим пакетом. Проте кожен пакет мережевого рівня передається по мережі як єдиний, незалежний від інших блок даних. У разі якщо будь-які пакети "загубилися", то модуль мережевого протоколу на приймаючій стороні не зможе виявити втрату, і, отже – виявити порушення цілісності загального масиву даних. Тому кошти транспортного рівня забезпечують відсутність втрат інформації. Такий режим передачі даних отримав назву гарантованої доставки.

Таким чином, засоби транспортного рівня представляють собою функціональну надбудову над мережним рівнем і вирішують дві основні задачі:

1) забезпечення доставки даних між конкретними програмами, що функціонують, в загальному випадку, на різних вузлах мережі;

2) забезпечення гарантованої доставки масивів даних довільного розміру.

В даний час в Інтернет використовуються два транспортних протоколу – UDP, що забезпечує негарантовану доставку даних між програмами, і TCP, що забезпечує гарантовану доставку з встановленням віртуального з'єднання.

Для ідентифікації програм протоколи транспортного рівня в мережі Інтернет (TCP і UDP), використовують унікальні числові значення, так звані порти. Номери портів призначаються програмами відповідно до її функціонального призначення на основі певних стандартів. Для кожного протоколу існують стандартні списки відповідності номерів портів і програм. Так, наприклад, програмне

забезпечення WWW, що працює через транспортний протокол TCP, використовує TCP-порт 80, модулі протоколу FTP – TCP-порт 21, а служба DNS взаємодіє з транспортними протоколами TCP і UDP через TCP-порт 53 і UDP-порт 53 відповідно.

Таким чином, протокол мережевого рівня IP і транспортні протоколи TCP і UDP реалізують дворівневу схему адресації: номери TCP-і UDP-портів дозволяють однозначно ідентифікувати програму в рамках вузла, а сам вузол однозначно визначається IP-адресою. Отже, комбінація IP-адреси і номера порту дозволяє однозначно ідентифікувати програму в мережі Інтернет. Така комбінована адреса називається сокетом або socket.

Протокол UDP (User Datagram Protocol) – протокол транспортного рівня, що входить в стек протоколів TCP/IP, що забезпечує негарантовану доставку даних без встановлення віртуального з'єднання (рисунок 4.3).

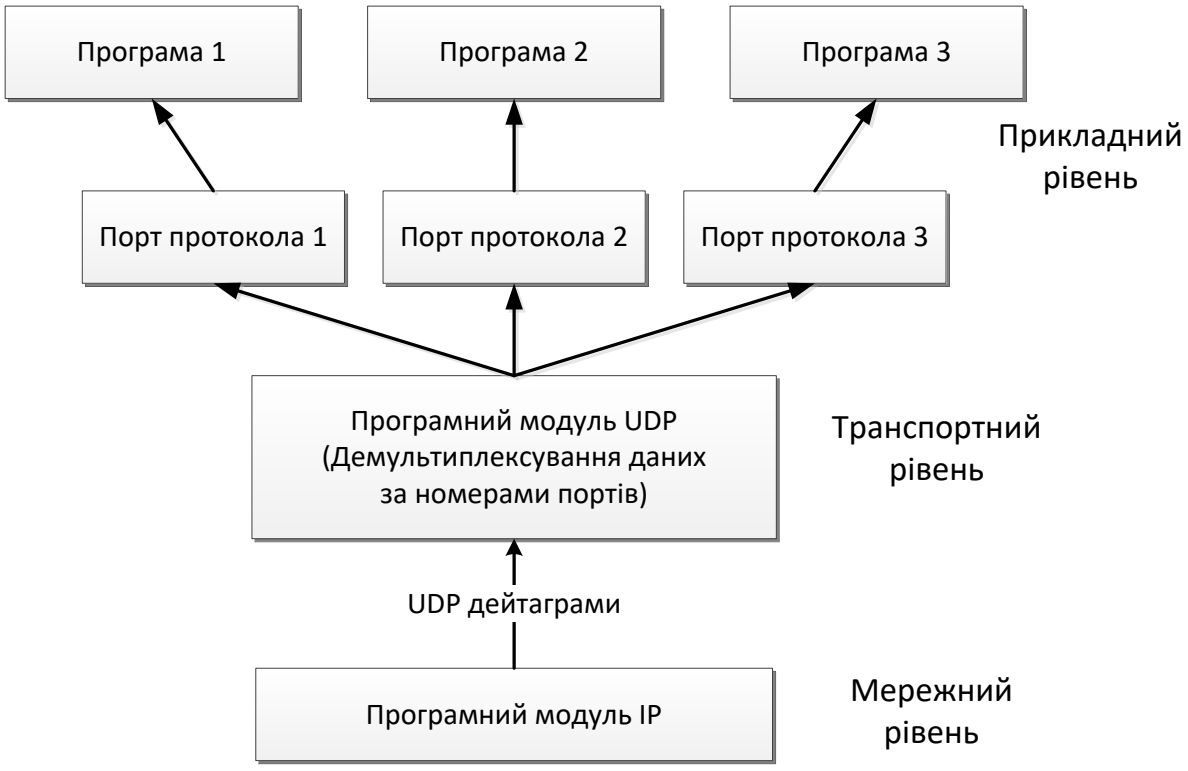


Рисунок 4.3 – Робота додатку за UDP протоколом

Оскільки на протокол не покладається завдань щодо забезпечення гарантованої доставки, а лише потрібно забезпечувати зв'язок між

різними програмами, то структура заголовка дейтаграми UDP, таку назву має пакет протоколу, виглядає досить просто – вона включає в себе всього чотири поля. Перші два поля містять номери UDP-портів програми-відправника та програми-одержувача. Два інших поля в структурі заголовка дейтаграми призначені для управління обробкою – це загальна довжина дейтаграми і контрольна сума заголовка.

Протокол TCP (Transmission Control Protocol) є транспортним протоколом стека протоколів TCP/IP, що забезпечує гарантовану доставку даних з встановленням віртуального з'єднання (рисунок 4.4).

Протокол надає програмам, що використовують його, можливість передачі безперервного потоку даних. Дані, що підлягають відправці в мережу,розбиваються на порції, кожна з яких забезпечується службовою інформацією, тобто формуються пакети даних. У термінології TCP пакет називається сегментом.

Відповідно до функціонального призначення протоколу структура TCP-сегмента передбачає наявність наступних інформаційних полів:

1) номер порту-відправника і номер порту-одержувача – номери портів, що ідентифікують програми, між якими здійснюється взаємодія;

2) поля, призначені для забезпечення гарантованої доставки: розмір вікна, номер послідовності і номер підтвердження;

3) керуючі прапори – спеціальні бітові поля, що управляють протоколом.

Для забезпечення гарантованої доставки протокол TCP використовує механізм відправки підтвердження. З метою зниження завантаження мережі протокол TCP допускає посилку одного підтвердження відразу для декількох отриманих сегментів. Обсяг даних, які можуть бути передані в мережу відправником до отримання підтвердження, визначається спеціальним параметром протоколу TCP – розміром вікна. Розмір вікна узгоджується при встановленні з'єднання між відправником та одержувачем і може автоматично змінюватися програмними модулями протоколу TCP в залежності від стану каналу зв'язку. Якщо в процесі передачі даних втрати

відбуваються досить часто, то розмір вікна зменшується, і навпаки – вікно може мати великий розмір, якщо висока надійність каналу даних.

Для того, щоб дані могли бути правильно зібрані одержувачем в потрібному порядку, в заголовку TCP-сегмента присутня інформація, яка визначає положення вкладених даних в загальному потоці. Відправляючи підтвердження, одержувач вказує положення даних, які він очікує отримати в наступному сегменті, тим самим побічно повідомляючи відправнику, який фрагмент загального потоку був успішно прийнятий. Відповідні поля заголовка TCP-сегмента отримали назву номер послідовності і номер підтвердження.

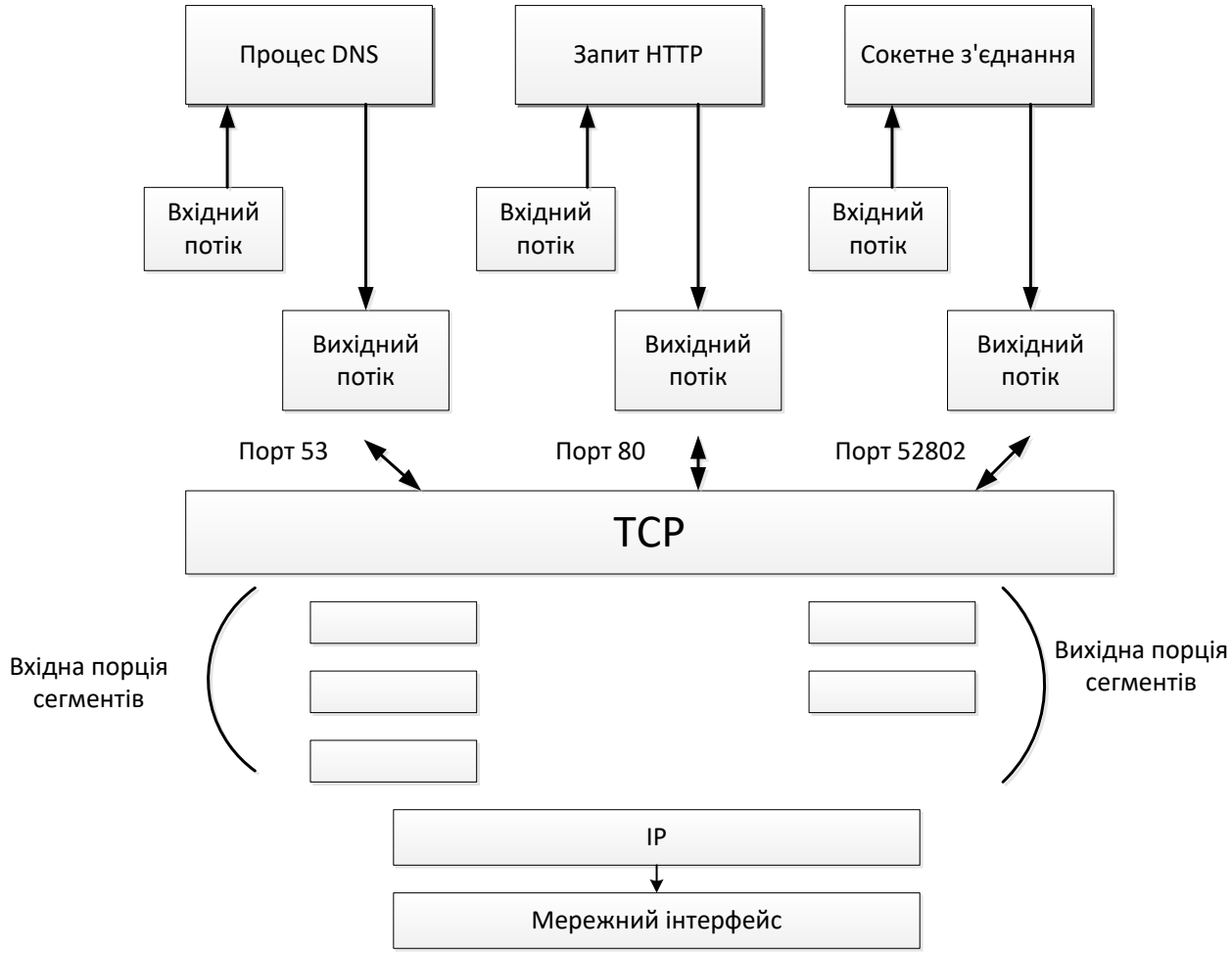


Рисунок 4.4 – Робота додатку за TCP протоколом

Таблиця 4.3 – Порівняння протоколів TCP та UDP

TCP	UDP
Надійний протокол	Ненадійний протокол
Встановлення віртуального з’єднання	Без встановлення віртуального з’єднання
Повторна передача сегментів та керування потоком	Без повторної передачі повідомлень
Використовується впорядкування сегментів	Випадковий порядок отримання сегментів
Для контролю слугують сегменти підтвердження	Підтвердження передачі відсутнє

Перед початком передачі потоку даних абоненти повинні узгодити параметри передачі: розмір вікна та початкові номери послідовностей, щодо яких буде відраховуватися положення переданих в сегментах даних усередині загального потоку. Очевидно, що таке узгодження передбачає обмін спеціальними сегментами і виділення ресурсів, зокрема, блоків пам’яті, необхідних для прийому та обробки даних і підтверджень. Відповідна послідовність дій називається встановленням віртуального з’єднання.

Мережні засоби платформи .NET Framework

У таблиці 4.1 коротко подано перелік класів бібліотеки .NET, з використанням яких можна побудувати мережний додаток.

Таблиця 4.1. – Основні класи для роботи з сокетами в .NET

Клас .NET	Опис
`Socket`	Забезпечує базову функціональність сокета для додатка
`TcpClient`	Побудований на класі Socket, щоб забезпечити TCP обслуговування на більш високому рівні. TcpClient надає декілька методів для відправки та отримання даних мережею

Клас .NET	Опис
`TcpListener`	Побудований на низькорівневому класі Socket. Його основне призначення – серверні додатки. Він очікує вхідні запити на з'єднання від клієнтів та повідомляє додатки про будь-які з'єднання
`UdpClient`	Призначений для реалізації UDP обслуговування
`SocketException`	Цей виняток генерується, коли у сокеті виникає помилка

Базовим класом при подубові мережних додатків є клас System.Net.Sockets.Socket, деякі властивості та методи якого представлено в таблиці 4.2.

Таблиця 4.2 – Короткий опис класу `System.Net.Sockets.Socket`

Властивості та методи	Короткий опис та призначення
`AddressFamily`	Сімейство адрес сокета (значення із перерахунку Socket.AddressFamily)
`Available`	Об'єм даних, які можна зчитати
`Blocking`	Чи знаходиться сокет в блокуючому режимі
`Connected`	Чи з'єднаний сокет з віддаленим хостом
`LocalEndPoint`	Локальна кінцева точка сокета
`ProtocolType`	Тип протокола сокета
`RemoteEndPoint`	Віддалена кінцева точка сокета
`SocketType`	Тип сокета
`Accept()`	Створює новий сокет для обробки вхідного запиту на зєднання
`Bind()`	Звязує сокет з локальною кінцевою точкою для очікування вхідних запитів на зєднання
`Close()`	Закриває сокет
`Connect()`	Встановлює з'єднання з віддаленим хостом
`Listen()`	Переводить сокет в режим прослуховування
`Receive()`	Отримує дані від приєднаного сокета

Властивості та методи	Короткий опис та призначення
`Poll()`	Визначає статус сокета
`Send()`	Відправляє дані з'єднаному сокету
`ShutDown()`	Забороняє операції відправки та отримання даних на сокеті

Аудиторні завдання

TCP сервер, побудований з використанням класу `Socket`

```
static void Main(string[] args)
{
    //Sockets.NET Server

    IPEndPoint ipEndPoint =
        new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8000);
    Socket serverSock = new Socket(AddressFamily.InterNetwork,
                        SocketType.Stream, ProtocolType.Tcp);
    serverSock.Bind(ipEndPoint);
    Console.WriteLine("Ожидание входящего TCP подключения...");
    serverSock.Listen(10);

    Socket clientSock = serverSock.Accept();
    IPEndPoint remote = (IPEndPoint)clientSock.RemoteEndPoint;
    Console.WriteLine("Соединение установлено по адресу {0}.
              Входящее сообщение:",
remote.Address.ToString());
    byte[] bufRecieve = new byte[8];
    clientSock.Receive(bufRecieve);
    Console.WriteLine(Encoding.UTF8.GetString(bufRecieve));
    Console.WriteLine("Для завершения нажмите любую
клавишу...");
    Console.ReadKey();
    serverSock.Close();
    clientSock.Close();
}
```

TCP клієнт, побудований з використанням класу Socket

```
static void Main(string[] args)
{
    //NET.Socket Client
    Socket cliSock = new Socket(AddressFamily.InterNetwork,
                    SocketType.Stream, ProtocolType.Tcp);
    IPAddress ipServ = IPAddress.Parse("127.0.0.1");
    IPEndPoint ipEndP = new IPEndPoint(ipServ, 8000);
    try
    {
        cliSock.Connect(ipEndP);
    }
    catch (SocketException ex)
    {
        Console.WriteLine(ex);
    }
    byte[] pack = Encoding.UTF8.GetBytes("Hello");
    cliSock.Send(pack);
    Console.WriteLine("Передача завершена. Для продолжения
нажмите любую клавишу...");
    Console.ReadKey();
    cliSock.Close();
  }
```

TCP сервер, побудований з використанням класу TcpListener

```
static void Main(string[] args)
{
    //TCPListener .NET
    TcpListener tcpServer =
       new TcpListener(IPAddress.Parse("127.0.0.1"), 8000);
    tcpServer.Start();
    Console.WriteLine("Ожидание подключения...");
    TcpClient tcpClient = tcpServer.AcceptTcpClient();
    Console.WriteLine("Подключение установлено,
                      входное сообщение:");
    byte[] buffer = new byte[8];
    NetworkStream streamTcp = tcpClient.GetStream();
```

```
    streamTcp.Read(buffer, 0, buffer.Length);
    Console.WriteLine(Encoding.UTF8.GetString(buffer));
    streamTcp.Close();
    tcpClient.Close();
    tcpServer.Stop();
    Console.WriteLine("Нажмите любую клавишу для
завершения...");
    Console.ReadKey();
}
```

TCP клієнт, побудований з використанням класу `TcpListener`

```
static void Main(string[] args)
{
    //TCPClient .NET
    TcpClient clientTcp = new TcpClient();
    clientTcp.Connect(IPAddress.Parse("127.0.0.1"), 8000);
    NetworkStream streamTcp = clientTcp.GetStream();
    byte[] buffer = new byte[8];
    buffer = Encoding.UTF8.GetBytes("Hello");
    streamTcp.Write(buffer, 0, buffer.Length);
    streamTcp.Close();
    clientTcp.Close();
    Console.WriteLine("Передача выполнена.
                    Нажмите любую клавишу для
завершения...");
    Console.ReadKey();
}
```

Індивідуальні завдання

Відповідно до варіанта та обраного рівня складності виконати наступні завдання.

Рівень завдань		
Початковий	Базовий	Високий
Відкомпілювати та протестувати аудиторні завдання	Завдання № 1	Завдання № 2

Завдання № 1

Розробити клієнтський та серверний додаток шляхом внесення незначних змін в програми аудиторного завдання.

№ варіанта	Завдання
1	Ввести своє ім’я та вік з консолі, відправити їх з клієнта на сервер.
2	Відправити рядок текста з сервера на клієнт, на клієнті вивести довжину рядка в символах.
3	Відправити масив із 10 цілих чисел з клієнта на сервер.
4	Відправити масив із 5 чисел типу `double` з сервера на клієнт.
5	Відправити два числа з клієнта на сервер та знайти їх добуток.
6	Відправити три числа з сервера на клієнт та знайти їх середнє арифметичне значення.
7	Відправити з сервера на клієнт повідомлення та вивести на клієнті його у зворотному порядку.
8	Відправити своє прізвище, ім’я, по батькові від клієнта до сервера та вивести на консоль.
9	Знайти максимальний елемент масиву, відправити масив та це значення з серверу на клієнт.
10	Відправити свою IP-адресу на сервер та знайти суму октетів на ньому.
11	Ввести своє ім’я та вік з консолі, відправити їх з сервера на клієнт.
12	Відправити з сервера на клієнт повідомлення та вивести на клієнті його прописними літерами.
13	Відправити масив із 8 беззнакових цілих чисел з клієнта на сервер.
14	Відправити масив із 6 чисел типу `byte` з клієнта на сервер.

№ варіанта	Завдання
15	Ввести своє прізвище та групу з консолі, відправити їх з клієнта на сервер.
16	Відправити рядок текста з клієнта на сервер, на сервері вивести довжину рядка в символах.
17	Відправити масив із 5 символів з клієнта на сервер.
18	Відправити масив із 10 чисел типу `double` з клієнта на сервер.
19	Відправити два числа з клієнта на сервер та знайти остачу від їх частки.
20	Відправити три числа з сервера на клієнт та знайти їх середнє геометричне значення.
21	Відправити з сервера на клієнт повідомлення та вивести на клієнті його у без пробілів.
22	Відправити своє прізвище ім’я, по батькові від клієнта до сервера та вивести на консоль прізвище та ініціали.
23	Відправити масив з серверу на клієнт та знайти мінімальний елемент масиву.
24	Відправити свою IP-адресу на сервер та знайти на ньому побітову операцію АБО октетів.
25	Ввести своє прізвище та факультет з консолі, відправити їх з сервера на клієнт.
26	Відправити три числа з клієнта на сервер та знайти суму тільки додатніх.
27	Відправити два рядки текста з сервера на клієнт, на клієнті вивести довжину рядків в символах.
28	Відправити масив із 10 символів з клієнта на сервер, на сервері змінити їх регістр на протилежний.
29	Відправити масив із 5 чисел типу `float` з клієнта на сервер.
30	Відправити два числа з клієнта на сервер та знайти їх різницю.

Завдання № 2

Розробити клієнтський та серверний додаток на базі TCP протоколу.

№ варіанту	Завдання
1	Відправити з клієнта на сервер своє ім’я, на сервері додати до імені своє прізвище та відправити результат на клієнт.
2	Відправити на сервер два числа, знайти їх суму та повернути результат клієнту.
3	Відправити на клієнт два числа, знайти їх добуток та повернути результат серверу.
4	Відправити на сервер масив, знайти його максимальний елемент та повернути результат клієнту.
5	Відправити на сервер число, сервер перевіряє, чи є воно більшим за введене на ньому значення, якщо так, то повертає на клієнт своє число, інакше повертає клієнту його ж число.
6	Відправити на сервер число, сервер перевіряє, чи є це число паліндромом. Якщо так, то відправляє на клієнт «Yes», якщо ні, то повертає «No».
7	Відправити з сервера на клієнт масив чисел, відсортувати їх за зростанням та повернути на сервер.
8	Відправити на сервер повідомлення, знайти у ньому кількість літер «а», та відправити на клієнт результат.
9	Відправити на клієнт повідомлення, знайти його довжину і відправити відповідну кількість нулів на сервер.
10	Відправити повідомлення на сервер, перевірити чи є в ньому цифри, якщо так, то відправити повідомлення «Yes» на клієнт, інакше «No»
11	Відправити з клієнта на сервер своє прізвище, на сервері додати до прізвища своє ім’я та відправити результат на клієнт.
12	Відправити на сервер три числа, знайти їх суму та повернути результат клієнту.

№ варіанту	Завдання
13	Відправити на клієнт три числа, знайти добуток від’ємних чисел та повернути результат серверу.
14	Відправити на сервер масив, знайти його мінімальний та максимальний елементи та повернути результати клієнту
15	Клієнт відправляє чотири числа на сервер, сервер додає до чисел їх середнє значення та повертає клієнту.
16	Відправити на сервер число, сервер перевіряє, чи є це число простим. Якщо так, то відправляє на клієнт «Yes», якщо ні, то повертає «No»
17	Відправити з сервера на клієнт масив цілих чисел, який вводить користувач, відсортувати їх за спаданням та повернути на сервер.
18	Відправити на сервер повідомлення, знайти у ньому кількість пробілів та відправити на клієнт результат.
19	Відправити на клієнт повідомлення, знайти його довжину і відправити відповідну кількість символів «X» на сервер.
20	Відправити повідомлення на сервер, перевірити чи є в ньому пробіли, якщо так, то відправити повідомлення «Yes» на клієнт, інакше «No»
21	Відправити на сервер число, розкласти його на прості множники, результат повернути клієнту у вигляді масиву.
22	Відправити на сервер число, сервер перевіряє, чи є це число квадратом іншого цілого числа. Якщо так, то відправляє на клієнт «Yes», якщо ні, то повертає «No».
23	Відправити з сервера на клієнт масив чисел, замінити від’ємні значення нулем та повернути на сервер.
24	Відправити на сервер повідомлення, знайти у ньому кількість голосних літер та відправити на клієнт результат.
25	Відправити на клієнт повідомлення, знайти його довжину і відправити відповідну кількість випадкових чисел на сервер.

№ варіанту	Завдання
26	Відправити повідомлення на сервер, перевірити чи містить воно знаки пунктуації, якщо так, то відправити повідомлення «Yes» на клієнт, інакше «No»
27	Відправити з клієнта на сервер рядок тексту, на сервері вилучити всі прописні літери, результат повернути клієнту.
28	Відправити на сервер три числа, знайти їх середнє геометричне та повернути результат клієнту.
29	Відправити на клієнт два повідомлення, якщо вони однакові, то серверу надіслати рядок тексту «Failed».
30	Відправити на сервер масив, знайти добуток його елементів та повернути результати клієнту

Контрольні питання

1. Що таке мережний сокет?
2. Які основні принципи мережної взаємодії додатків із застосуванням портів?
3. Коротко охарактеризуйте стек протоколів TCP/IP.
4. Перерахуйте номери портів таких протоколів та служб: HTTP, DNS, FTP, Telnet, ICQ, Skype.
5. Як співвідноситься модель відкритої взаємодії OSI та стек протоколів TCP/IP?
6. Чим відрізняються протоколи транспортного рівня TCP та UDP?
7. Коротко опишіть схему створення сокетного TCP з’єднання.
8. Які засоби платформи .NET Framework призначені для передачі даних між вузлами комп’ютерної мережі?
9. Запропонуйте підхід до створення багатоклієнтського сервера на основі протокола TCP.
10. Яким чином можна реалізувати луна-сервер та визначити час передачі даних між вузлами?

ЛАБОРАТОРНА РОБОТА № 5

Тема: основи розподілених обчислень із застосуванням технології Windows Communication Foundation (WCF) на мові C#.

Мета: розробити службу WCF, яка надає певні методи обчислень для віддалених клієнтських додатків.

Теоретичні відомості

Windows Communication Foundation (WCF) – це платформа для побудови сервісноорієнтованих додатків. За допомогою WCF можна відправляти дані у вигляді асинхронних повідомлень від однієї кінцевої точки служби до іншої. Кінцева точка служби може входити в постійно доступну службу, що розміщена в IIS, або представляти службу, яка розміщується в звичайному програмному додатку. Кінцева точка може бути клієнтом служби, яка виконує запити. Повідомлення можуть представляти одинарні символи чи одне слово, яке відправляється у форматі XML, або має вигляд складного потоку двійкових даних. Далі представлено декілька зразків сценаріїв.

1) Захищена служба для обробки бізнес-транзакцій.

2) Служба, яка передає іншим об’єктам поточні дані, такі як звіт про трафік, або інша служба спостереження.

3) Надання доступу до робочого процесу, реалізованому за допомогою Windows Workflow Foundation, у вигляді служби WCF тощо.

Такі додатки можна було створювати і до появи WCF, проте WCF істотно спрощує розробку кінцевих точок. Таким чином, платформа WCF реалізує керований підхід до створення веб-служб і клієнтів веб-служб.

Для побудови розподілених систем WCF, зазвичай, створюються три наступні взаємопов’язані збірки:

– *збірка служби WCF*. Ця бібліотека *.dll містить класи та інтерфейси, які забезпечують загальну функціональність, котрою користуються клієнти;

– *хост служби WCF*. Цей програмний модуль містить в собі збірку служби WCF і виконує роль сервера у розподіленій системі;

– *клієнт WCF*. Це додаток, який звертається до функціональності служби через проміжний проксі.

На рисунку 7.1 показано відношення між цими збірками. «За лаштунками» використовується багато низькорівневих деталей, внутрішні механізми передачі даних, налаштування транспортного рівня (фабрики, канали, слухачі).

Слід відзначити, що застосування файла *.config серверною та клієнтською стороною не є обов'язковим. За бажанням можна жорстко закодувати хост та клієнт, вказавши необхідні деталі (кінцеві точки, прив'язки, адреси). Очевидна проблема такого підходу полягає в тому, що, якщо знадобиться змінити деталі настроювання, то потрібно вносити зміни в код, проводити перекомпіляцію і наново розміщувати збірки. Використання файлів *.config робить кодову базу набагато гнучкою.

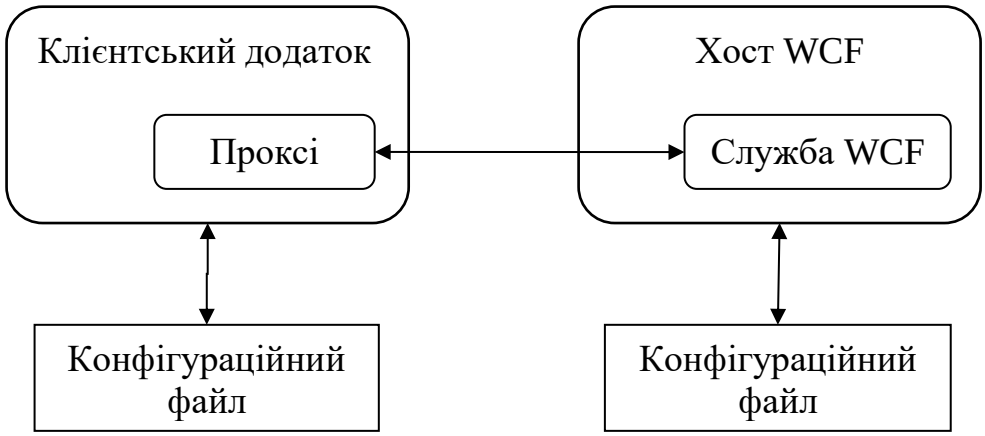


Рисунок 5.1 – Високорівневе представлення типового WCF додатку

Хости і клієнти взаємодіють один з одним з використанням адрес, прив'язок та контрактів (`address`, `binding`, `contract`):

– *адреса* описує місцезнаходження служби, в коді представлена типом `System.Uri`, проте її значення, зазвичай, зберігається у файлах *.config;

– *прив'язка*. Інфраструктура WCF надає багато різних прив'язок, які зазначають мережні протоколи, механізми кодування та тарнспортний рівень;

– *контракт* надає опис кожного методу, який відкритий зі служби WCF.

Аудиторне завдання

Побудувати розподілену систему WCF – гру «Магічний шар». Декілька користувачів підключаються до серверного додатка, задають питання та отримують на них одну випадкову відповідь, наприклад, «так», «ні», «можливо» тощо. Клієнтський додаток розробити засобами Windows Froms.

Розв'язання

Побудова служби WCF

Створимо збірку MagicBallService, яка буде надавати певний функціонал через WCF.

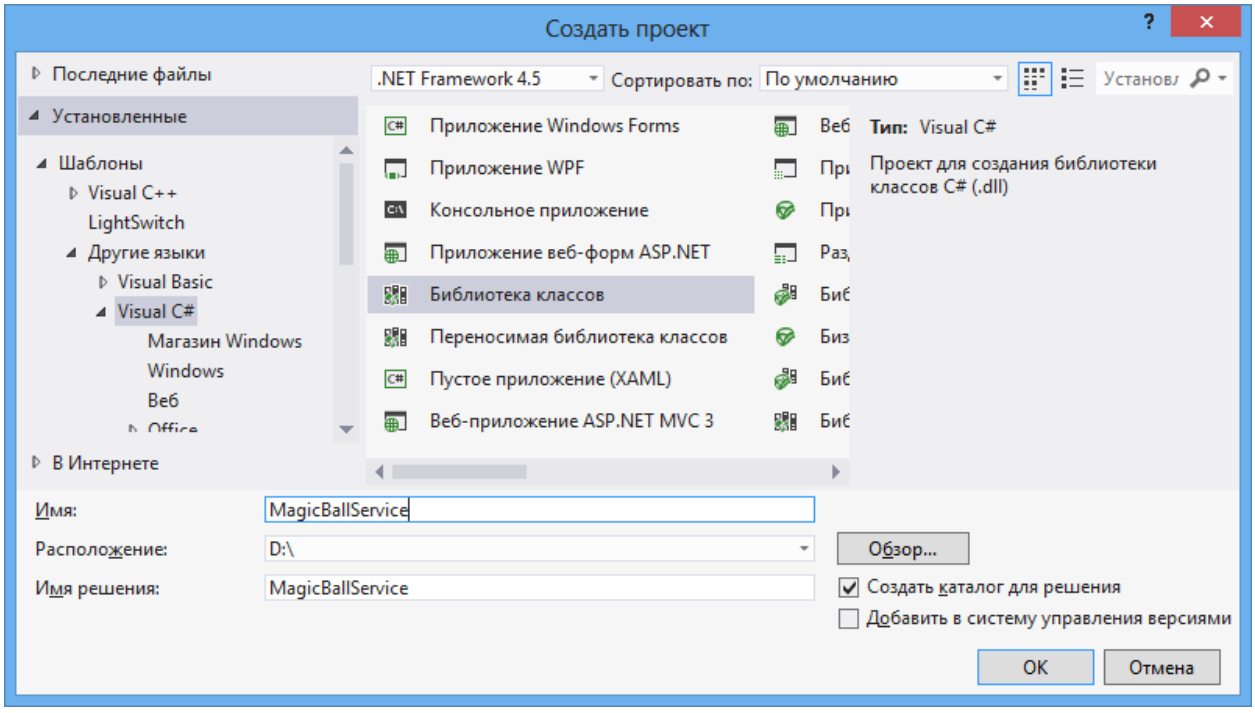


Рисунок 5.2 – Створення DLL-збірки

У створеному проекті слід видалити файл *Class1.cs* та створити свій власний. З метою більшої гнучкості доцільно функціонал WCF-служби подати у вигляді інтерфейсу `IMagicBall` та класу `MagicBall`, який реалізує даний інтерфейс.

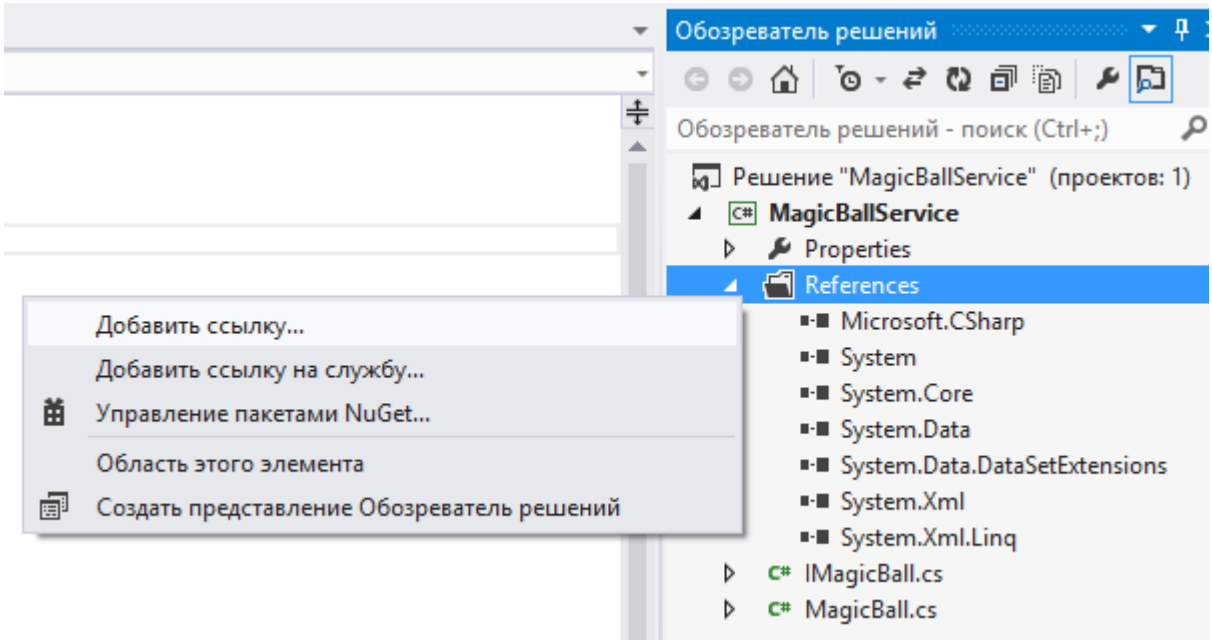


Рисунок 5.3 – Додавання посилання на збірку

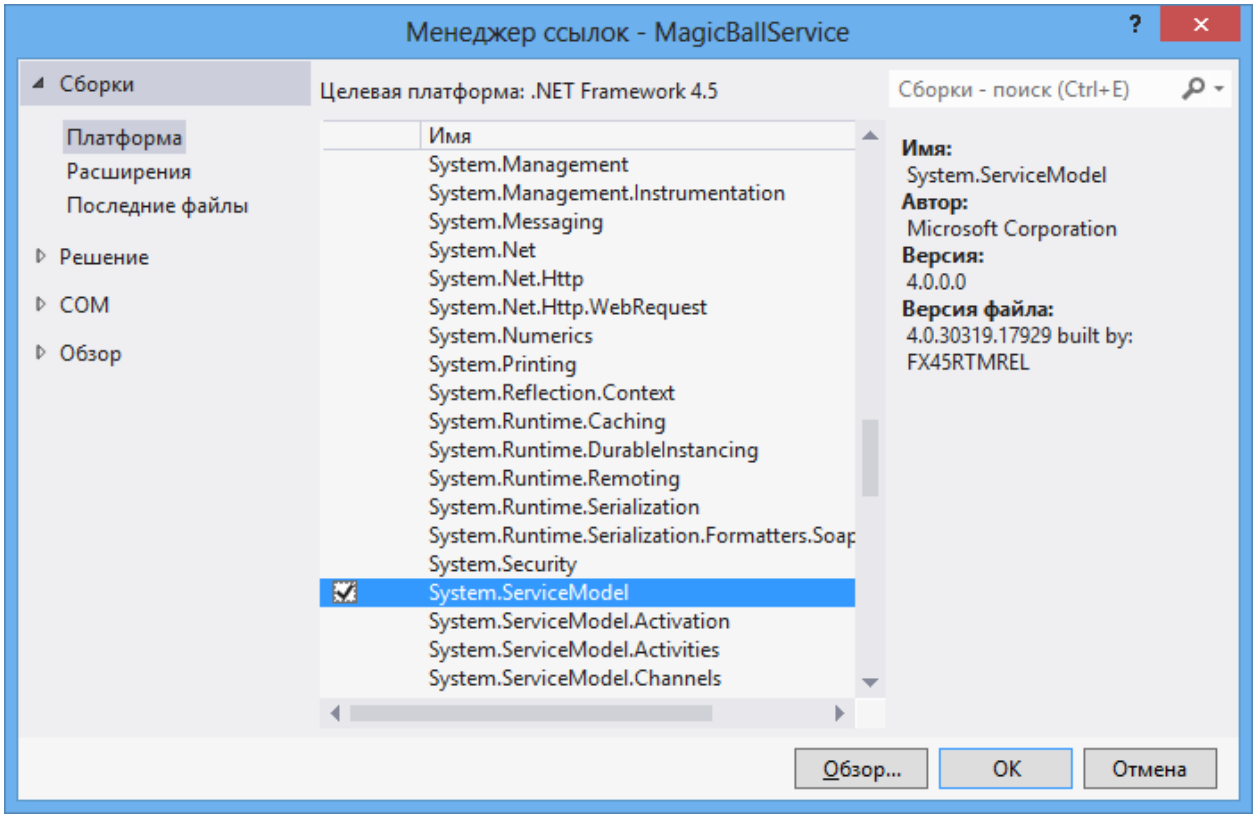


Рисунок 5.4 – Вибір збірки System.ServiceModel.dll

Для відкриття методів інтерфейсу через WCF-службу необхідно додати посилання на збірку `System.ServiceModel` (рисунки 5.3, 5.4) та позначити методи атрибутами `[ServiceContract]`, `[OperationContract]`.

Вміст створеного класу та інтерфейсу подано нижче.

Лістинг 5.1 – Код інтерфейсу `IMagicBall`

```
using System.ServiceModel;
namespace MagicBallService
{
    [ServiceContract]
    interface IMagicBall
    {
        [OperationContract]
        string GetAnswer(string userQuestion);
    }
}
```

Лістинг 5.2 – Код класу `MagicBall`

```
using System;
namespace MagicBallService
{
    public class MagicBall: IMagicBall
    {
        // для отображения на хосте
        public MagicBall()
        {
            Console.WriteLine("MagicBall ожидает вопросы от
пользователей...");
        }

        // реализация интерфейсного метода
        public string GetAnswer(string userQuestion)
        {
            string[] answers = { "Будущее неопределенно",
"Да", "Нет", "Возможно", "Задайте вопрос позже", "Определенно"
};
            // Вернуть случайный ответ
```

```
            Random rnd = new Random();
            return answers[rnd.Next(answers.Length)];
        }
    }
}
```

Після компіляції збірки служби переходимо до створення серверного хоста, де дана збірка буде розміщена.

Хостинг служби WCF

Тепер все готово для визначення хоста. Зазвичай служба промислового рівня повинна розміщуватися в службі Windows або у віртуальному каталозі IIS, проте для спрощення реалізації виконаємо хостинг служби WCF у звичайному консольному додатку. Для цього до відкритого рішення, яке було створене на попередньому кроці додаємо новий проект MagicBallHost. Обов'язково слід не забути додати посилання на збірки System.ServiceModel.dll та MagicBallService.dll (рисунок 5.5).

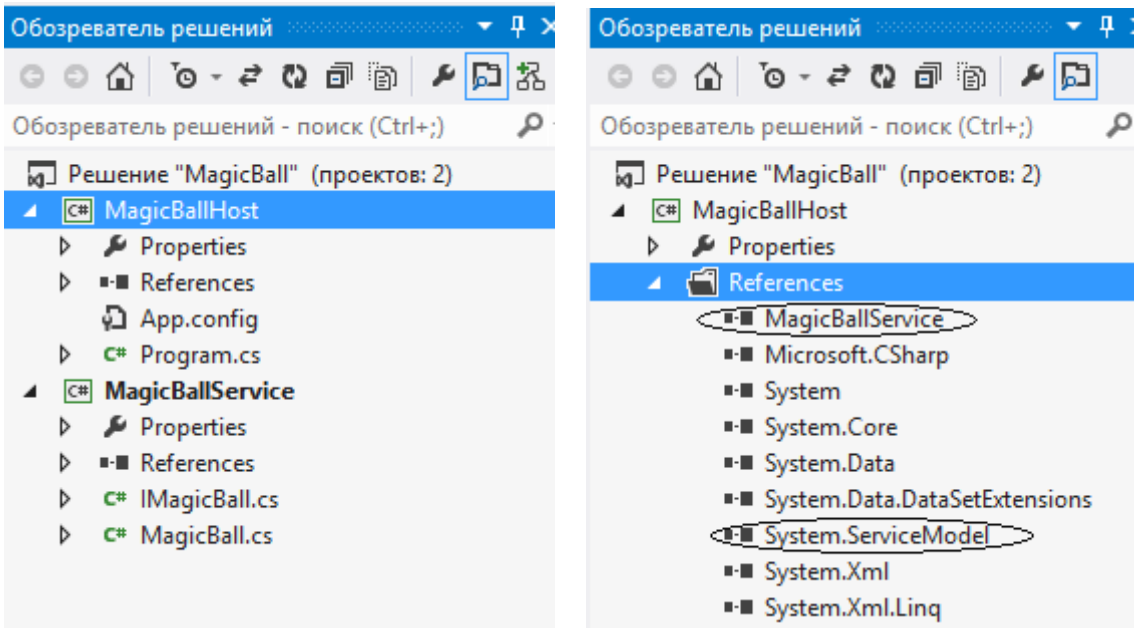


Рисунок 5.5 – Додавання до рішення нового проекту MagicBallHost та посилань на збірки

Лістинг 5.3 – Код MagicBallHost

```
using System;
using System.ServiceModel;
using MagicBallService;

namespace MagicBallHost
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("*** Console Based WCF Host
***");
            using (ServiceHost serviceHost =
                   new ServiceHost(typeof(MagicBall)))
            {
               // Открыть хост на прослушивание входных
сообщений
                serviceHost.Open();
                // Оставить службу функционируюшей до тех пор,
                   пока не будет нажата клавиша Enter
                Console.WriteLine("The service is ready");
                Console.WriteLine("Press Enter to terminate
                                   service");
                Console.ReadLine();
            }
        }
    }
}
```

У файлі App.config слід додати налаштування служби WCF, задавши адресу, протокол, контракт, поведінку при HTTP-запитах, дозволити передачу метаданих. У лістингу 5.4 жирним шрифтом виділено фрагмент, який потрібно додати до конфігураційного файлу.

Лістинг 5.4 – Код файлу App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
  <startup>
    <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5" />
  </startup>
  <system.serviceModel>
    <services>
      <service name="MagicBallService.MagicBall"
behaviorConfiguration="MagicBallServiceMexBehavior">
        <endpoint address="http://localhost:8000/MagicBallService"
                  binding="basicHttpBinding"
                  contract="MagicBallService.IMagicBall"></endpoint>
        <endpoint address="mex"
                  binding="mexHttpBinding"
                  contract="IMetadataExchange"></endpoint>
        <host>
          <baseAddresses>
            <add
baseAddress="http://localhost:8000/MagicBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MagicBallServiceMexBehavior">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Для того, щоб запустити на виконання консольний проект його потрібно назначити головним (рисунок 5.6). Крім того середовище Visual Studio необхідно запустити з правами адміністратора.

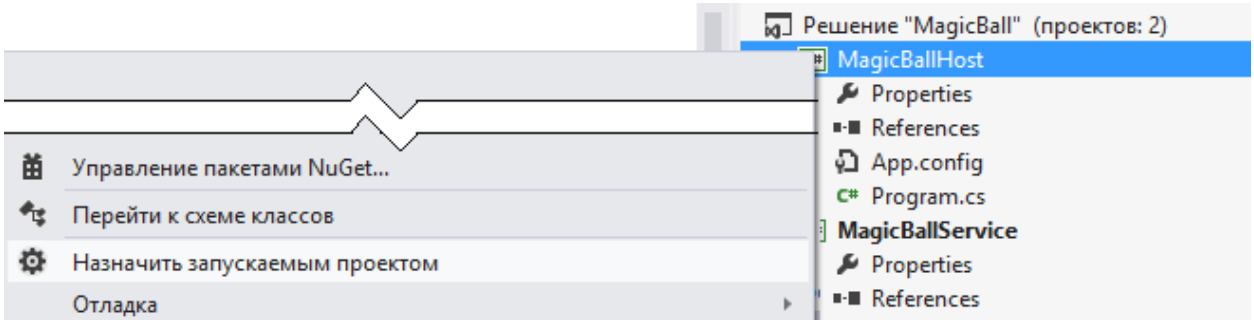


Рисунок 5.5 – Призначення проекту на виконання MagicBallHost

Після запуску хост-додатку можна перевірити працездатність служби WCF за допомогою браузера (рисунок 5.6).

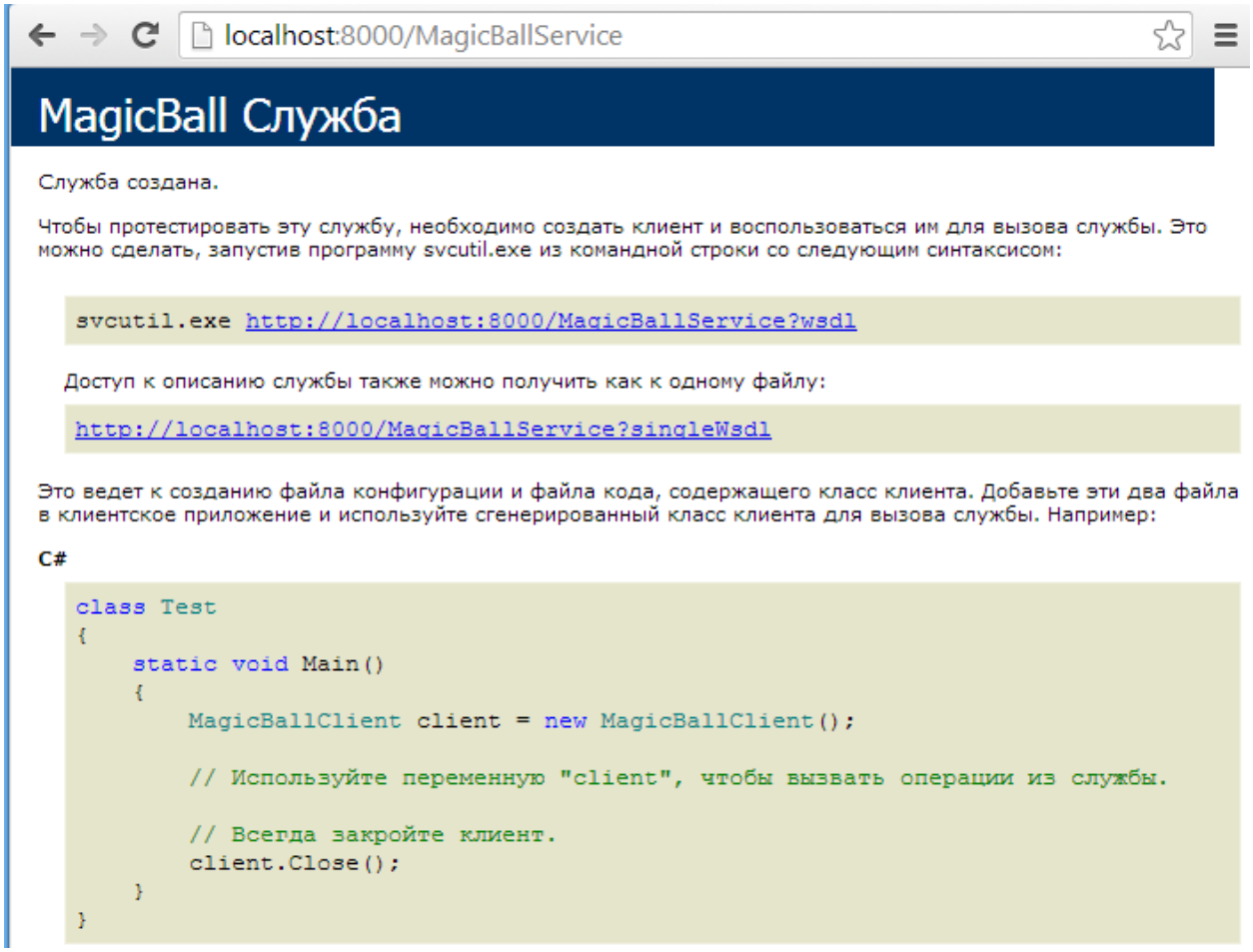


Рисунок 5.6 – Перегляд метаданих служби за допомогою браузера

Побудова клієнтського додатку WCF

Клієнтом служби WCF може бути будь-який додаток (консольний, WinFroms, WPF, веб-додаток тощо). У даному прикладі створимо Windows Forms клієнт з графічним інтерфейсом. До рішення слід

додати новий проект MagicBallClient та призначити його головним для виконання.

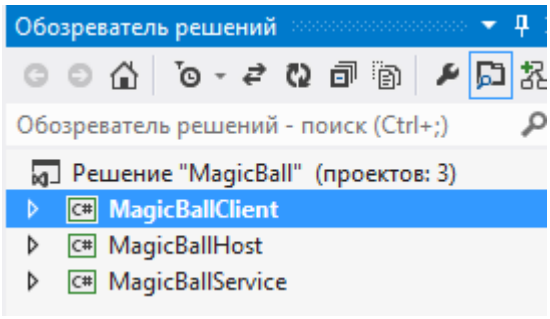


Рисунок 5.7 – Кінцева структура рішення MagicBall зі службою, хостом та клієнтом

Для використання можливостей WCF-служби у клієнтському додатку потрібно спершу запустити хост служби (файл MagicBallHost.exe), потім згенерувати у поточний проект код проксі за допомогою пункту меню Visual Studio «Проект/Добавить ссылку на службу…» (рисунок 5.8).

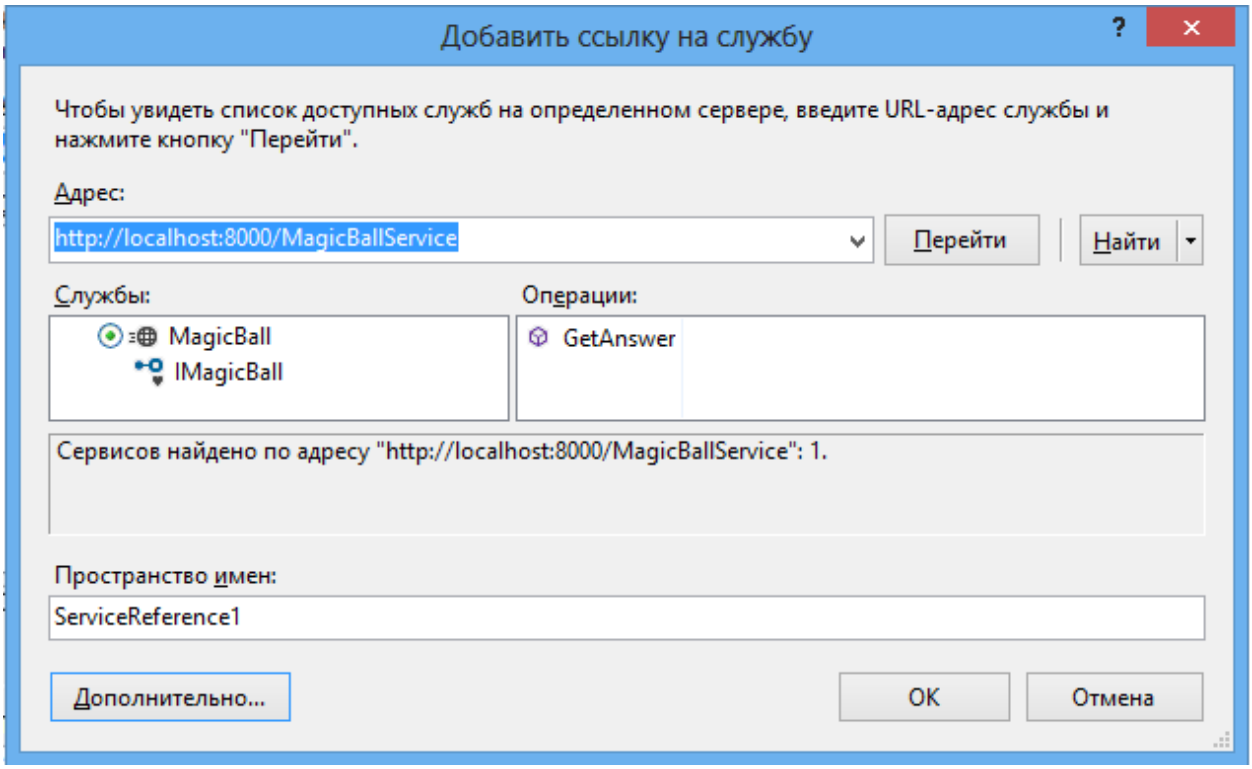


Рисунок 5.8 – Генерування проксі засобами Visual Studio

Проектування форми клієнтського додатку проводиться в конструкторі (рисунок 5.9).

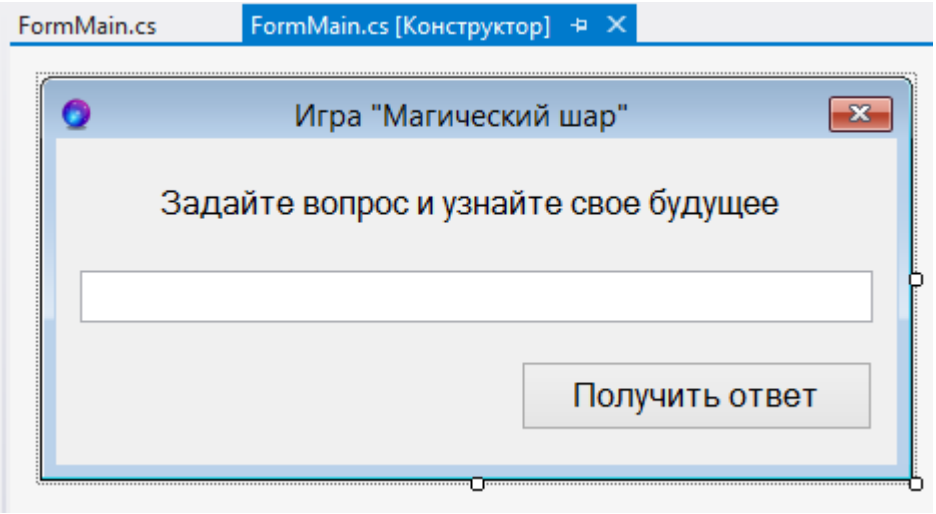


Рисунок 5.9 – Проектування форми клієнтського додатку

Програмний код клієнта представлено нижче.

Лістинг 5.5 – Код головного вікна програми

```
using System;
using System.Windows.Forms;
namespace MagicBallClient
{
    public partial class FormMain : Form
    {
        public FormMain()
        {
            InitializeComponent();
        }

        private void buttonAsk_Click(object sender, EventArgs
e)
        {
            using (ServiceReference1.MagicBallClient ball =
                   new ServiceReference1.MagicBallClient())
            {
                string
answer=ball.GetAnswer(textBoxQuestion.Text);
                MessageBox.Show(answer, "Ответ");
                textBoxQuestion.Clear();
            }
```

```
        }
    }
}
```

Індивідуальні завдання

Відповідно до варіанту та обраного рівня складності виконати наступні завдання.

Рівень завдань		
Початковий	Базовий	Високий
Відкомпілювати та протестувати аудиторні завдання	Завдання № 1	Завдання № 2

Завдання № 1

Розробити WCF-службу у вигляді DLL-бібліотеки, яка виконує певну задачу згідно з індивідуальним варіантом. Розмістити WCF-службу у серверному консольному додатку. Створити клієнтський консольний додаток, який використовує функціонал WCF-служби та передає для обчислення введений користувачем масив.

№ варіанта	Завдання
1	Обчислити суму всіх елементів масиву
2	Визначити кількість парних елементів масиву
3	Обчислити добуток усіх елементів масиву
4	Визначити середнє геометричне значення елементів масиву
5	Визначити кількість елементів масиву, які менші 10
6	Визначити кількість елементів масиву, які більші 12
7	Визначити кількість від’ємних елементів масиву
8	Визначити середнє арифметичне значення елементів масиву
9	Визначити кількість елементів, які менші середнього арифметичного значення цього масиву

№ варіанта	Завдання
10	Визначити перший елемент в масиві, який ділиться націло на 5 або на 3
11	Визначити кількість елементів масиву, які більші 25 і менші 50
12	Визначити кількість додатних елементів масиву
13	Визначити кількість додатних непарних чисел в масиві
14	Обчислити суму непарних елементів масиву
15	Визначити кількість парних елементів масиву, які менші 15
16	Обчислити добуток всіх елементів масиву, які націло діляться на 3
17	Визначити середнє геометричне значення елементів масиву
18	Визначити кількість елементів масиву, які менші 53
19	Визначити кількість елементів масиву, які більші 28
20	Визначити кількість від’ємних елементів масиву, які менші 30
21	Визначити середнє арифметичне значення додатних елементів масиву
22	Визначити кількість елементів, які більші середнього арифметичного значення цього масиву
23	Визначити перший елемент в масиві, який ділиться націло на 6 або на 7
24	Визначити кількість елементів масиву, які більші 5 і менші 20
25	Визначити кількість від’ємних елементів масиву, які більші 70
26	Визначити кількість додатних непарних чисел в масиві, які менші 50
27	Визначити кількість парних елементів масиву, які менші 15
28	Обчислити добуток всіх елементів масиву, які націло діляться на 17

№ варіанта	Завдання
29	Визначити середнє геометричне значення елементів масиву
30	Визначити кількість елементів масиву, які менші 14

Завдання № 2

Розробити WCF-службу у вигляді DLL-бібліотеки, яка виконує певну задачу згідно з індивідуальним варіантом. Розмістити WCF-службу у серверному консольному додатку. Створити клієнтський додаток типу Windows Forms, який використовує функціонал WCF-служби.

№ варіанта	Завдання
1	Виконати коригування тексту, видаляючи всі символи #, {}, $. Проте якщо є запис вартості у вигляді 100$, то такі відомості залишити без змін
2	Виконати коригування тексту, видаляючи зайві знаки пробілів як всередині речення, так і на початку та вкінці
3	Виконати коригування тексту, вставляючи символи пробілу для правильного розділення слів та знаків пунктуації у реченнях. Якщо пробіл вже є, то коригувати не потрібно.
4	Виконати коригування тексту, видаляючи з нього всі знаки дефіс та нулі у числах, які не є значимими, наприклад 15,620
5	Виконати коригування тексту, додаючи до назв валют RUR, UAH, USD їх розшифрування в дужках
6	Визначити кількість слів, які повторюються в тексті більше 3 раз і хоча б два з них знаходяться разом
7	Виконати коригування тексту таким чином, щоб всі числа були виділені знаками //

№ варіанта	Завдання
8	Виконати коригування тексту, видаляючи з нього всі зайві коми
9	Виконати коригування тексту таким чином, щоб усі великі літери були замінені на їх номер в алфавіті
10	Виконати коригування тексту, видаляючи з нього всі зайві крапки
11	Виконати коригування тексту таким чином, щоб всі слова починалися з великої літери, а всі інші літери були тільки малими
12	Виконати коригування тексту, видаляючи всі числа, які є номерами телефонів
13	Виконати коригування тексту таким чином, щоб усі цифри в тексті були виділені пробілами
14	Виконати коригування тексту, видаляючи з нього всі зайві крапки, проте крапки і трикрапки в кінці речень видаляти не потрібно.
15	Виконати коригування тексту, видаляючи з нього всю інформацію, записану у дужках, дужки видалити також

Контрольні питання

1. Що таке Windows Communication Foundation?
2. Які можливі прив’язки для транспортного рівня WCF?
3. Які атрибути потрібно використовувати для надання доступу до методів інтерфейсу засобами WCF?
4. Перерахуйте обчислювальні системи за класифікацією Фліна.
5. Обґрунтуйте закон Амдала для розподілених обчислювальних систем.
6. Що таке адреса у межах технології WCF?
7. Порівняйте можливості технології WCF зі звичайними сокетами.
8. Коротко охарактеризуйте можливості бібліотеки MPI.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Клири Стивен. Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование : 2-е межд. изд. СПб. : Питер, 2020. 272 с.
2. Троелсен Э., Джепикс Ф. Язык программирования C# 7, платформы .NET и .NET Core : 8-е изд., пер. с англ. СПб. : Диалектика, 2018. 1328 с.
3. Федотов И. Е. Параллельное программирование. Модели и приемы. М. : Солон-Пресс, 2017. 390 с.
4. Малявко А. А. Параллельное программирование на основе технологий OpenMP, MPI, CUDA. М. : Юрайт, 2016. 116 с.
5. Борзунов С. В., Кургалин С. Д., Флегель А. В. Практикум по параллельному программированию. СПб. : БХВ-Петербург, 2016. 239 с.
6. Луцків А. М., Лупенко С. А., Пасічник В. В. Паралельні та розподілені обчислення : навч. посіб. Львів : Магнолія, 2015. 566 с.
7. Богачев К. Ю. Основы параллельного программирования. М. : Бином, 2014. 342 с.
8. Тормасов А. Г. Параллельное программирование многопоточных систем с разделяемой памятью. М. : Физматкнига, 2014. 208 с.
9. Уильямс Э. Д. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. М. : ДМК Пресс, 2014. 672 с.
10. Келвин Лин, Лоуренс Снайдер. Принципы параллельного программирования : учеб. пособ. М. : МГУ, 2013. 388 с.
11. Ovais Mehboob Ahmed Khan. C# 7 and .NET Core 2.0 High Performance: Build highly performant, multi-threaded, and concurrent applications using C# 7 and .NET Core 2.0. – Birmingham : Packt Publishing, 2018. 302 p.
12. Thomas Sterling, Matthew Anderson, Maciej Brodowicz. High Performance Computing: Modern Systems and Practices. USA : Morgan Kaufmann, 2017. 718 p.

13. Quinn J. Mihcael. Parallel Programming in C with MPI And OpenMP. McGraw Hill Education India Pvt Ltd, 2017. 544 p.

14. Posch Maya. Mastering C++ Multithreading: Write robust, concurrent, and parallel applications. Birmingham : Packt Publishing, 2017. 244 p.

15. Frank Nielsen. Introduction to HPC with MPI for Data Science (Undergraduate Topics in Computer Science). Berlin : Springer, 2016. 282 p.

16. David B. Kirk, Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach : 3rd ed. USA : Morgan Kaufmann, 2016. 576 p.

17. Боресков А. В., Харламов А. А., Марковский Н. Д., Микушин Д. Н., Мортиков Е. В., Мыльцев А. А., Сахарных Н. А., Фролов В. А. Параллельные вычисления на GPU. Архитектура и программная модель CUDA. М. : МГУ, 2015. 336 с.

18. Ringler Rodney. C# Multithreaded and Parallel Programming. Packt Publishing, 2014. 345 p.

19. John Cheng, Max Grossman, Ty McKercher. Professional CUDA C Programming. Birmingham : Wrox, 2014. 486 p.

20. Эхтер Ш., Роберте Дж. Многоядерное программирование. СПб. : Питер, 2010. 316 с.

ДОДАТОК А
Правила оформлення звіту

1. Перед виконанням лабораторної роботи необхідно уважно вивчити теоретичну частину до неї.

2. Створити документ Microsoft Word.

2.1. Встановити параметри сторінки: відступ зліва – 2 см; відступ справа – 1,5 см; відступ знизу – 2 см; відступ зверху – 1,5 см.

2.2. Розмір шрифту – 14, міжрядковий інтервал – одинарний.

2.3. Нумерація звіту повинна бути наскрізною, починаючи з титульного аркуша. Номер сторінки зазначають посередині верхньої частини аркуша над текстом. На титульному аркушу номер сторінки не ставлять, але рахують.

2.4. У звіті можуть бути наведені переліки, перед якими ставлять двокрапку. Перед кожною позицією переліку ставлять риску (–) або рядкову літеру з дужкою. Для подальшої деталізації переліків використовують арабські цифри з дужкою, наприклад:

а) . . . . . .

б) . . . . . .

1) . . . . .

2) . . . . .

в) . . . . . .

2.5. У тексті звіту не дозволяється: вживати звороти розмовної мови; вживати застарілі та жаргонні терміни і вислови; вживати скорочені слова, крім встановлених стандартами скорочень.

2.6. Якщо в тексті наводиться ряд числових значень в однакових одиницях, то позначення одиниці виміру зазначають тільки після останнього числового значення, наприклад: 1, 2, 3 м; або від 5 до 10 мм. Одиниці вимірювання від числових величин відокремлюють нерозривним пробілом (Ctrl+Shift+Space).

2.7. Числові значення величин треба відокремлювати від десяткової частини комою, наприклад: 7,5; 8,75; 10,00. У необхідних випадках треба використовувати математичне округлення, наприклад: вірно...розмір файлу складає 20 МБ; невірно … розмір файлу складає 20,036 МБ.

2.8. Послідовність розміщення матеріалу у звіті повинна бути наступною:

− титульний аркуш (додаток А);

− зміст;

− лабораторні роботи:

- номер лабораторної роботи (стиль *Заголовок1)*;
- тема та мета роботи;
- анотовані теоретичні відомості, які застосовуються при розв'язанні індивідуальних завдань;
- умова завдання та його розв'язок;
- матеріали самостійної роботи студента згідно виданого завдання (матеріали повинні бути оброблені та систематизовані, не допускається прямого копіювання з джерел інформації);
- висновок до лабораторної роботи;
- список використаних джерел.

− загальний висновок до лабораторного курсу.

2.9. Оформлення ілюстрацій.

2.9.1. Усі ілюстрації у звіті у вигляді креслень, ескізів, схем, графіків, діаграм, фотографій та ін. називаються рисунками.

2.9.2. Рисунки можуть бути виконані олівцем, пастою, тушшю, фломастером та розташовані на окремих аркушах або безпосередньо в тексті записки, якщо рисунки невеликі.

2.9.3. Рисунки нумеруються в межах кожної лабораторної роботи двома цифрами – номером роботи і порядковим номером рисунку в роботі, розділеними крапкою.

2.9.4. Кожний рисунок повинен мати найменування. Слово «Рисунок», його номер та найменування розміщують під рисунком та записують таким чином:

Рисунок 1.3 – Функціональна схема пристрою

2.9.5. Після номеру ставиться тире (–), а після найменування крапка не ставиться.

2.9.6. На усі рисунки повинні бути посилання у тексті роботи, наприклад: ... наведено на рисунку 2.6.

2.9.7. Графіки повинні мати координатні осі та координатну сітку. На координатних осях необхідно наносити числові значення змінних

величин; найменування фізичної величини, яка пишеться текстом паралельно відповідній осі, та через кому позначають одиницю виміру фізичної величини. Напис розміщують поза полем графіка, у кінці напису крапка не ставиться.

2.10. Оформлення таблиць.

2.10.1. Таблиці нумерують у межах кожної роботи арабськими цифрами, розділеними крапкою, та розташовують над таблицею ліворуч. Кожна таблиця повинна мати назву, яку пишуть над таблицею. Перед назвою таблиці пишуть слово «Таблиця» і її номер, який складається з номера розділу і порядкового номера таблиці в межах розділу. Номер таблиці від назви виділяють тире, наприклад:

Таблиця 4.1 – Технічні характеристики модему ADE-4400

2.10.2. Якщо висота таблиці перевищує одну сторінку, її продовження переноситься на наступну сторінку. При цьому лінію, що обмежує першу частину таблиці знизу, не проводять, а над продовженням таблиці на наступній сторінці пишуть «Продовження таблиці 4.1». При переносі таблиці допускається її головку замінювати номерами граф, відповідно до їх номерів у першій частині таблиці.

2.10.3. На всі таблиці повинні бути посилання у тексті записки, наприклад: ... наведено в таблиці 4.1.

2.11. Оформлення формул.

2.11.1. Формули і математичні рівняння подаються у тексті окремим рядком і розташовуються на його середині. Переносити формулу на наступний рядок дозволяється тільки по знаках операцій, який повторюють на початку наступного рядка.

2.11.2. Формули нумерують у межах розділу арабськими цифрами. Номер складається з номера розділу та порядкового номера формули, розділених крапкою. Номер формули записують у круглих дужках на рівні праворуч формули. Посилання на формули у тексті записки дають у дужках, наприклад: ... у формулі (2.1).

2.11.3. Пояснення символів і числових коефіцієнтів, які входять у формулу, необхідно подавати безпосередньо під формулою. Пояснення кожного символу треба давати з нового рядка, причому перший рядок пояснення повинен починатися зі слова «де» без двокрапки після нього.

2.12. Оформлення списку використаних джерел.

2.12.1. Посилання на джерело наводиться у вигляді порядкового номера джерела, взятого в квадратні дужки. Якщо необхідно посилатися одночасно на декілька джерел, їх номери зазначають через кому чи тире, наприклад: [12]; [1,4,7]; [5-9]; [2 с. 4]; [3 таблиця 2.1].

2.12.2. Перелік літературних джерел розміщують у порядку їх згадування у роботі (найзручніший спосіб) або в алфавітному порядку.

2.12.3. Бібліографічний опис джерела в переліку має відповідати вимогам ДСТУ ГОСТ 7.1:2006 «Система стандартів з інформації, бібліотечної та видавничої справи. Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання». Бібліографічний опис дається мовою оригіналу. Прізвища авторів від їх ініціалів відокремлюють нерозривним пробілом (Ctrl+Shift+Space).

2.12.4. Дозволяється також у якості джерел інформації використовувати ресурси глобальної мережі інтернет, проте тільки офіційні сайти виробників обладнання, інтернет-магазинів для складання кошторису тощо. Заборонено включати до переліку використаних джерел такі сайти як www.google.com, www.yandex.ru, www.yahoo.com та ін., які є загальними пошуковими сервісами, а також www.wikipedia.org, www.ukrreferat.com, www.referat.ru та ін., де інформація може мати неперевірений характер і додаватися на сайт неспеціалістами.

3. До захисту поточної лабораторної роботи допускаються студенти, які надали звіт в електронному варіанті і захистили попередні лабораторні роботи.

4. Кожен студент захищає лабораторну роботу індивідуально.

5. На захисті викладач може задати будь-яке питання, що стосується теоретичної частини, і будь-яке завдання іншої бригади або підсумкове.

6. На останньому занятті викладачу подається роздрукований звіт з лабораторних робіт та його електронний варіант.

7. До екзамену допускаються студенти, які захистили всі лабораторні роботи.

ЗМІСТ

Методичні вказівки
до виконання лабораторних робіт з дисципліни
«Паралельні та розподілені обчислення»
для студентів спеціальності
123 «Комп'ютерна інженерія»
усіх форм навчання

УКЛАДАЧІ: Музика Іван Олегович
Кузнєцов Денис Іванович

Реєстраційний № ___

Підписано до друку _____ 2021 р.
Формат _____А5_____
Обсяг _____138_____ стор.
Тираж _____ прим.

Видавничий центр
Криворізького національного університету,
вул. Віталія Матусевича, 11, м. Кривий Ріг