**ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ОЛЕСЯ ГОНЧАРА**

Кафедра електронних обчислювальних машин

**КУРСОВА РОБОТА**

з дисципліни «Системне програмне забезпечення»

Студента IIІ курсу групи КІ-21-2

спеціальності 123 – «Комп’ютерна інженерія»

Шейко Ростислав Олександрович

Керівник: доц. Матвєєва Н. О.

Кількість балів

Національна шкала

Члени комісії:

доц. Герасимов В.В.

доц. Матвєєва Н.О.

ас. Морозов О.С.

м. Дніпро

2024

**ЗМІСТ**

[**ВСТУП 3**](#_Toc167263591)

[**ЗАВДАННЯ НА КУРСОВУ РОБОТУ 4**](#_Toc167263592)

[**2. ТЕОРИТИЧНА ЧАСТИНА 5**](#_Toc167263593)

[**2.1. Потоки 5**](#_Toc167263594)

[**2.1.1. Визначення потоку 5**](#_Toc167263595)

[**2.1.2. Створення та завершення потоку 8**](#_Toc167263596)

[**2.1.3. Призупинення та відновлення потоку 10**](#_Toc167263597)

[**2.1.4. Пріоритети потоків 10**](#_Toc167263598)

[**2.1.5. Визначення моменту закінчення потоку 11**](#_Toc167263599)

[**2.1.6. Властивість IsBackground 12**](#_Toc167263600)

[**2.2. Процеси 12**](#_Toc167263601)

[**2.2.1 Визначення процесу 12**](#_Toc167263602)

[**2.2.2. Запуск та завершення процесів 13**](#_Toc167263603)

[**2.3. Синхронізація 14**](#_Toc167263604)

[**2.3.1. Визначення синхронізації 14**](#_Toc167263605)

[**3. ПРАКТИЧНА ЧАСТИНА 19**](#_Toc167263606)

[**3.1. Структура та вигляд додатку 19**](#_Toc167263607)

[**3.2. Функціонал додатку 19**](#_Toc167263608)

[**3.3. Тестування 21**](#_Toc167263609)

[**ВИСНОВКИ 25**](#_Toc167263610)

[**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 26**](#_Toc167263611)

[**ДОДАТОК А 27**](#_Toc167263612)

# ВСТУП

У сучасному світі розробка програмного забезпечення стає все більш складною і потребує використання різноманітних технологій для досягнення оптимальної продуктивності та ефективності.

Мета цієї курсової роботи полягає у створенні проекту на мові програмування C#, який забезпечує можливість змінювати розміри вікна без втрати функціональності, використовуючи потоки, процеси та механізми синхронізації.

Актуальність даної курсової роботи полягає у необхідності розуміння та ефективного використання різноманітних аспектів системного програмного забезпечення в реальних проектах.

Завдання курсової роботи включають в себе розробку проекту на мові програмування C#, який дозволяє змінювати розміри вікна та продовжувати при цьому працювати, використання потоків та об'єктів синхронізації для забезпечення коректної роботи програми, а також реалізацію функціональності, що дозволяє користувачеві встановлювати кількість потоків.

Виконання цих завдань дозволить отримати практичні навички у розробці ефективного програмного забезпечення, використанні можливостей багатопоточності та синхронізації для оптимізації обробки даних. Це підготує до вирішення реальних інженерних завдань у сучасних обчислювальних середовищах, де вимоги до продуктивності та надійності є надзвичайно високими.

# ЗАВДАННЯ НА КУРСОВУ РОБОТУ

Загальні вимоги до проектів.

1. Проект на мові програмування C# Forms.

2. Програма повинна дозволяти змінювати розміри вікна і продовжувати при цьому працювати.

3. Використовувати потоки, процеси і обов'язково якийсь об'єкт синхронізації (функції очікування, події, м'ютекси, семафори), передача даних між процесами виконати через відображення файлів в пам'ять або канали.

4. Програма повинна дозволяти ставити кількість потоків (процесів).

5. Варіанти за списком в журналі.

*Варіант 19. Завдання Виробник-Споживач*

Паралельні потоки здійснюють циклічні виклики функцій.

Результати роботи розміщуються кожним потоком в загальну чергу кінцевої довжини. Третій потік бере результати роботи з черги і виводить їх на екран.

Черга повинна бути реалізована з використанням двох об'єктів семафорної синхронізації (для потоків 1 і 2, і для потоку 3). Коли черга заповнена, потоки 1 і 2 повинні бути призупинені функцією очікування. Якщо в черзі немає результатів, потік 3 також повинен бути призупинений функцією очікування.

Розділ результатів у черзі коду має бути відформатований за допомогою об'єкта синхронізації монітора.

# 2. ТЕОРИТИЧНА ЧАСТИНА

## **2.1. Потоки**

### **2.1.1.** **Визначення потоку**

Потік (thread) - це незалежна послідовність інструкцій у програмі. Потоки дозволяють програмі одночасно виконувати різні завдання. Наприклад, під час введення коду мови програмування C# у вікні редактора Visual Studio окремий фоновий потік проводить аналіз на наявність синтаксичних помилок. Аналогічно працює засіб перевірки орфографії в Microsoft Word: один потік очікує введення даних користувачем, тоді як інший фоново здійснює аналіз. Третій потік може записувати дані у тимчасовий файл, а четвертий - завантажувати додаткові дані з Інтернету [2].

У серверному додатку один потік завжди очікує запити від клієнтів (потік-слухач). Отримавши запит, він негайно передає його окремому робочому потоку, який продовжує взаємодію з клієнтом. Потік-слухач повертається до очікування нових запитів від наступних клієнтів.

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

Потік має пріоритет, лічильник команд, який вказує на місце в програмі, де відбувається обробка, та стек, в якому зберігаються локальні змінні потоку. Стек у кожного потоку свій, але пам'ять для програмного коду та купа поділяються між усіма потоками, які функціонують усередині одного процесу [2].

Потоки в межах одного процесу можуть швидко взаємодіяти між собою, оскільки всі вони звертаються до однієї і тієї ж віртуальної пам'яті. Однак це ускладнює ситуацію, оскільки безліч потоків можуть змінювати одну і ту ж область пам'яті.

Розрізняють два види багатозадачності:

- на основі процесів;

- на основі потоків.

У багатозадачності на основі процесів програма є найменшою одиницею коду, яку координує планувальник завдань ОС. На комп'ютері можуть одночасно виконуватися кілька програм, наприклад, електронні таблиці, текстовий редактор та браузер.

Потік є координованою одиницею виконуваного коду. У багатозадачності на основі потоків процес може мати більше одного потоку, що дозволяє одночасно вирішувати декілька завдань у межах однієї програми. Наприклад, текст у редакторі може форматуватися одночасно з його друком, якщо ці дії виконуються в різних потоках.

Відмінності багатозадачності на основі процесів і потоків:

- багатозадачність на основі процесів призначена для паралельного виконання програм;

- багатозадачність на основі потоків - для паралельного виконання окремих частин однієї програми.

Потік може перебувати в одному з кількох станів:

- виконується;

- готовий до виконання, як тільки отримає час і ресурси ЦП;

- призупинений, але може бути відновлений;

- заблокований в очікуванні ресурсів;

- завершений, коли його виконання закінчено і не може бути відновлено.

У середовищі .NET є два типи потоків:

- пріоритетний;

- фоновий.

За замовчуванням новостворений потік є пріоритетним, але його можна зробити фоновим. Пріоритетний потік відрізняється від фонового тим, що фоновий потік автоматично завершується, якщо всі пріоритетні потоки в процесі зупинені.

Для координації виконання потоків використовується синхронізація. У мові програмування C# для цього передбачена окрема підсистема.

Кожен процес складається хоча б з одного потоку, який називається основним, оскільки саме з нього починається виконання програми. З основного потоку можна створювати інші потоки.

У C# і .NET підтримуються обидва види багатозадачності: на основі процесів і потоків. Засобами C# можна створювати як процеси, так і потоки, та керувати ними. Класи, які підтримують багатопотокове програмування, визначені в просторі імен System.Threading. Найважливішим з них є клас Thread, який представляє окремий потік.

Щоб отримати посилання на поточний потік, необхідно викликати статичну властивість Thread.CurrentThread:

Thread currThread = Thread.CurrentThread;

При запуску додатку на C# операційна система створює процес, а середовище CLR створює всередині цього процесу логічний контейнер - домен додатків (AppDomain), в якому працює додаток. У .NET немає прямої відповідності "один до одного" між доменами додатків і потоками. Домен додатків може мати кілька потоків, що виконуються одночасно, і потоки можуть перетинати межі доменів додатків за рішенням планувальника Windows та CLR [2].

Хоча потоки можуть перетинати межі доменів додатків, у кожен конкретний момент часу потік може виконуватися тільки в одному домені додатків. Щоб отримати доступ до домену додатків, в якому працює поточний потік, використовується метод Thread.GetDomain():

AppDomain ad = Thread.GetDomain();

Кожен потік може бути переміщений в певний контекст, і він може переміщатися в межах нового контексту за примхою CLR. Для отримання поточного контексту потоку використовується статична властивість Thread.CurrentContext, яка повертає об'єкт System.Runtime.Remoting.Contexts.Context:

Context ctx = Thread.CurrentContext;

### **Створення та завершення потоку**

Для створення потоку необхідно отримати екземпляр об'єкта типу `Thread`, визначеного в просторі імен `System.Threading`. Ось найпростіша форма конструктора класу `Thread`:

public Thread (ThreadStart запуск)

де `запуск` — це ім'я методу, що використовується для початку виконання потоку, а `ThreadStart` — це делегат, визначений у .NET:

public delegate void ThreadStart()

Отже, метод, що вказується як точка входу до потоку, повинен повертати тип `void` і не приймати аргументів.

Новостворений потік не почне виконуватися, поки не буде викликаний його метод `Start()`, визначений у класі `Thread`. Після запуску потік буде виконуватися, доки не завершиться метод, на який вказує `запуск`. Після завершення цього методу потік автоматично зупиняється. Спроба повторного виклику методу `Start()` для вже запущеного потоку призведе до винятку `ThreadStateException` [2].

Іноді потік потрібно перервати до його нормального завершення. Наприклад, може виникнути необхідність зупинити потік, який вийшов з-під контролю. Для цього використовується метод `Thread.Abort()`. Ось найпростіша форма цього методу:

public void Abort()

Метод `Abort()` створює умови для генерування винятку `ThreadAbortException` у потоці, для якого він був викликаний. Цей виняток призводить до переривання потоку і може бути перехоплений у коді програми, але в такому випадку він автоматично генерується ще раз для завершення потоку. Метод `Abort()` не завжди зупиняє потік негайно, тому після його виклику варто викликати метод `Join()` для гарантованої зупинки потоку. У рідкісних випадках метод `Abort()` може не зупинити потік, наприклад, якщо блок `finally` входить у нескінченний цикл [2].

З початку .NET 5 метод `Thread.Abort` визнаний застарілим. Виклик цього методу попереджає компілятора, а під час виконання викликає виняток `PlatformNotSupportedException`.

Запит на передчасне переривання можна скасувати всередині потоку. Для цього потрібно перехопити виняток `ThreadAbortException` і викликати метод `ResetAbort()`. Це дозволяє уникнути повторного генерування винятку після завершення обробника винятку, що перериває потік. Ось форма оголошення методу `ResetAbort()`:

public static void ResetAbort()

Виклик методу `ResetAbort()` може завершитися невдачею, якщо потік не має належного режиму надійного скасування передчасного переривання [2].

### **Призупинення та відновлення потоку**

У ранніх версіях середовища .NET Framework потік можна було призупинити за допомогою методу `Thread.Suspend()` і відновити за допомогою методу `Thread.Resume()`. Проте, обидва ці методи тепер вважаються застарілими і не рекомендуються до використання. Це пояснюється тим, що використання методу `Suspend()` є небезпечним, оскільки він може призупинити потік, який утримує блокування. Це може перешкоджати зняттю блокування і призводити до взаємоблокування. Використання обох методів може викликати серйозні проблеми на системному рівні. Тому для припинення та відновлення потоку слід застосовувати інші засоби синхронізації, такі як м'ютекси та семафори [2].

### **Пріоритети потоків**

Кожен потік має свій пріоритет, який впливає на частоту доступу потоку до процесора (ЦП). Зазвичай фонові потоки отримують доступ до ЦП рідше, ніж високопріоритетні потоки. Це означає, що фоновий потік отримує менше часу ЦП у порівнянні з високопріоритетними потоками протягом певного періоду. Час ЦП, який виділяється потоку, значно впливає на його виконання і взаємодію з іншими потоками, що працюють у системі одночасно [2].

Варто враховувати, що на частоту доступу потоку до ЦП впливають і інші фактори, окрім пріоритету. Наприклад, якщо високопріоритетний потік очікує доступу до ресурсу (наприклад, введення з клавіатури), він блокується, і в цей час виконується фоновий потік. У такій ситуації фоновий потік може отримувати більше часу ЦП, ніж високопріоритетний потік, протягом певного періоду. Крім того, планування завдань операційною системою також впливає на час ЦП, що виділяється для потоку.

Коли створений потік починає виконуватися, йому надається пріоритет за замовчуванням. Пріоритет потоку можна змінити за допомогою властивості `Priority`, яка є членом класу `Thread`. Ось загальна форма цієї властивості:

public ThreadPriority Priority { get; set; }

де `ThreadPriority` представляє перерахування, що визначає можливі значення пріоритетів для потоку.

### **Визначення моменту закінчення потоку**

У класі `Thread` є два способи визначити момент завершення потоку. Один із них – це доступна тільки для читання властивість `IsAlive`, визначена наступним чином:

public bool IsAlive { get; }

Властивість `IsAlive` повертає логічне значення `true`, якщо потік, до якого вона звертається, ще виконується.

Інший спосіб відстеження моменту завершення потоку полягає у виклику методу `Join()`. Ось його найпростіша форма:

public void Join()

Метод `Join()` очікує, доки потік, для якого він був викликаний, не завершиться. Назва методу відображає його призначення: очікування до тих пір, поки потік, який викликає `Join()`, не приєднається до завершеного потоку. Якщо викликати метод `Join()` для потоку, який ще не розпочався, буде згенеровано виняток `ThreadStateException`. В інших варіантах методу `Join()` можна вказати максимальний період часу для очікування завершення конкретного потоку [2].

### **Властивість IsBackground**

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

*public bool IsBackground {get; set; }*

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

## **2.2. Процеси**

### **2.2.1 Визначення процесу**

При програмуванні на C# зазвичай організовується багатозадачність на основі потоків. Однак, коли це доцільно, можна реалізувати багатозадачність на основі процесів. У цьому випадку, замість запуску нового потоку в межах однієї програми, одна програма починає виконання іншої. Це здійснюється за допомогою класу `Process`, визначеного в просторі імен `System.Diagnostics`.

У Windows процес розуміється як об'єкт ядра, що володіє системними ресурсами, які використовуються виконуваним додатком. Таким чином, процесом у Windows є додаток, що виконується. Виконання кожного процесу починається з первинного потоку. Під час виконання процес може створювати додаткові потоки. Виконання процесу завершується, коли завершуються всі його потоки. Кожен процес в операційній системі Windows володіє такими ресурсами:

- Віртуальним адресним простором.

- Робочим набором сторінок у реальній пам'яті.

- Маркером доступу, що містить інформацію для системи безпеки.

- Таблицею для зберігання дескрипторів об'єктів ядра.

Окрім дескриптора, кожен процес у Windows має свій унікальний ідентифікатор, який використовується службовими програмами для відстеження роботи процесів [2,10].

Іноді процесу потрібно знати свій дескриптор, щоб змінити свої характеристики, наприклад, змінити свій пріоритет. Псевдодескриптор процесу відрізняється від справжнього дескриптора тим, що він може використовуватися лише поточним процесом і не може бути успадкованим іншими процесами. Псевдодескриптор процесу не потребує закриття після використання. З псевдодескриптором процесу можна отримати справжній дескриптор процесу.

### **Запуск та завершення процесів**

Простий спосіб запустити новий процес можливий за допомогою методу `Start()`, що належить класу `Process`. Ось одна з найпростіших форм цього методу:

public static Process Start (string ім'я\_файлу)

де `ім'я\_файлу` вказує на конкретне ім'я файлу, який має бути виконаний або пов'язаний з виконуваним файлом.

Процес, який створює новий процес, називається "батьківським процесом" відносно створюваного процесу. Новий процес, який створюється іншим процесом, називається "дочірнім процесом" відносно батьківського процесу [2].

Після завершення створеного процесу рекомендується викликати метод `Close()`, щоб звільнити пам'ять, виділену для цього процесу. Ось форма оголошення методу `Close()`:

public void Close()

## **2.3. Синхронізація**

### **Визначення синхронізації**

Синхронізація процесів визначає фіксований порядок їх виконання, щоб уникнути конкуренції або взаємного блокування при роботі зі спільними ресурсами. У програмуванні розглядаються паралельні процеси як програми, які виконуються процесором [2].

Всі потоки, які належать одному процесу, ділять деякі спільні ресурси, такі як адресний простір оперативної пам'яті або відкриті файли. Оскільки ці ресурси належать всьому процесу, кожен потік може використовувати їх без обмежень. Проте, якщо один потік ще не завершив роботу зі спільним ресурсом, а система переключилася на інший потік, який також використовує цей ресурс, результат роботи цих потоків може сильно відрізнятися від очікуваного. Ці конфлікти можуть виникнути і між потоками, які належать різним процесам. Виникає ця проблема завжди, коли два або більше потоки використовують один спільний ресурс [3].

Саме тому потрібен механізм, який дозволяє потокам узгоджувати свою роботу зі спільними ресурсами. Цей механізм відомий як механізм синхронізації потоків.

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

Існує декілька об'єктів синхронізації, найважливіші з них:

- взаємовиключення (mutex),

- подія (event),

- семафор (semaphore),

- блокування (lock),

- монітор (monitor).

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

Будь-який об'єкт синхронізації може перебувати в так званому сигнальному стані, який має різний зміст для різних типів об'єктів. Потоки можуть перевіряти поточний стан об'єкта і / або чекати на його зміну для узгодження своїх дій. Операційна система гарантує, що коли потік працює з об'єктами синхронізації, його виконання не буде перерване, доки він не завершить дію.

**2.3.2. М'ютекс**

М'ютекс, або взаємовиключення, є засобом синхронізації, який гарантує, що доступ до об'єкта може мати лише один потік одночасно. Монітор - це розширення м'ютексу, яке може мати ім'я або бути безіменним. Якщо м'ютекс має ім'я, це означає, що він є глобальним і може бути доступним з різних процесів. Якщо ж немає імені, це анонімний м'ютекс, до якого можна отримати доступ лише в межах одного процесу.

М'ютекс може мати префікс Global\ або Local\, що вказує на глобальний або локальний простір імен. Локальні м'ютекси придатні для синхронізації між процесами з батьківським або дочірнім зв'язком в рамках того ж самого сеансу.

Наприклад, якщо є потреба у спільному доступі до системного журналу з декількох процесів, м'ютекс відмінно підходить для синхронізації, дозволяючи тільки одному процесу записувати дані у журнал у будь-який момент.

У класі `System.Threading.Mutex` існують кілька конструкторів, серед яких два найбільш уживаних. Перший конструктор створює м'ютекс, що спочатку не належить жодному потоку. Другий конструктор може надати м'ютекс одразу власнику.

Для отримання м'ютексу потрібно викликати метод `WaitOne()`, що призводить до призупинки виконання поточного потоку до моменту отримання м'ютексу. Після використання м'ютексу його необхідно звільнити методом `ReleaseMutex()`. Таким чином, одночасний доступ до загального ресурсу обмежується тільки одним потоком.

Крім того, м'ютекси допомагають уникнути конфліктів доступу до даних, що можуть призвести до непередбачуваних результатів або навіть викликати аварійні ситуації програми. Вони забезпечують механізм блокування, який дозволяє потоку чекати, поки не буде доступний м'ютекс, перед тим як продовжити виконання.

Це не тільки підвищує надійність програми, а й робить її ефективнішою за рахунок правильного використання ресурсів. Таким чином, м'ютекси стають важливим елементом при розробці програмного забезпечення, особливо в тих випадках, коли потрібно керувати доступом до спільних даних між потоками.

* + 1. **Семафор**

Семафори в мові програмування C# є потужним інструментом для управління паралельним виконанням програми та доступом до обмежених ресурсів.

Уявімо ситуацію, де маємо обмежену кількість дозволів на доступ до певного ресурсу, наприклад, до бази даних або до обмеженої кількості потоків у мережевій програмі. Семафор дозволяє нам контролювати цей доступ, обмежуючи кількість потоків, які можуть отримати доступ одночасно.

У мові C# семафори реалізовані за допомогою класу `Semaphore` в просторі імен `System.Threading`. Їх можна створити з вказанням максимальної кількості дозволів, які можуть бути видані одночасно. Наприклад, `Semaphore semaphore = new Semaphore(3, 3);` створить семафор, який дозволить одночасно отримати доступ до ресурсу трьом потокам.

Для того щоб потік міг взяти семафор, він використовує метод `WaitOne()`. Якщо дозвіл доступу доступний, потік продовжить виконання, інакше він буде зупинений, поки не звільниться дозвіл. Після закінчення роботи з ресурсом потік повинен звільнити семафор методом `Release()`, щоб інші потоки могли взяти його.

Semaphore semaphore = new Semaphore(3, 3); // створення семафору з трьома дозволами

// Потік спробує взяти дозвіл на доступ

semaphore.WaitOne();

// Робота з обмеженим ресурсом

// Після завершення роботи потрібно звільнити семафор

semaphore.Release();

Семафори також можуть бути використані для розв'язання багатьох інших задач паралельного програмування в C#, що робить їх важливим інструментом для розробників.

# 3. ПРАКТИЧНА ЧАСТИНА

## **3.1. Структура та вигляд додатку**

Для початку необхідно описати структуру самої програми. Вона складається з Windows Forms додатку. Цей додаток на платформі Windows Forms призначений для роботи з потоками і вивід інформації щодо об’єктів.

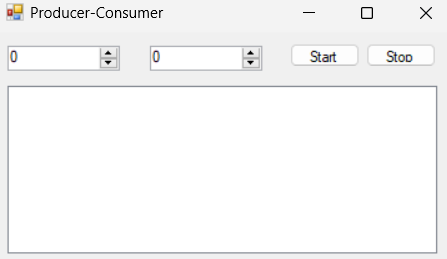


Рис.1 Вигляд додатку Windows Forms

## **3.2. Функціонал додатку**

Дослідження зосереджується на аналізі програмного коду, що реалізує виробничо-споживчу систему з використанням мови програмування C# та механізмів потоків. Програма призначена для демонстрації шаблону "Виробник-Споживач" і включає в себе взаємодію між потоками виробників і споживачів, забезпечену семафорами та потокобезпечною чергою.

Основні аспекти програми полягають у наступному:

1. Архітектурний контекст: Програма реалізована у вигляді класу Form1, що унаслідований від класу Form, що надає засоби для створення графічного інтерфейсу. У цьому контексті ініціалізуються об'єкти синхронізації та файл, відображений в пам'ять, а також визначені методи для керування потоками виробників і споживачів.

2. Потокова модель: Виробники і споживачі реалізовані у вигляді окремих потоків, які виконують відповідні завдання згідно з логікою програми. Потоки виробників створюють нові продукти та додають їх до черги, тоді як потоки споживачів вилучають продукти з черги для обробки.

3. Механізми синхронізації: Синхронізація доступу до черги реалізована за допомогою семафорів. Семафори emptySemaphore та fullSemaphore відповідають за контроль кількості вільних та заповнених місць у черзі відповідно. Потокобезпечна черга queue використовується для зберігання продуктів та забезпечення безпечного доступу до них з різних потоків.

4. Взаємодія з файлом, відображеним в пам'ять: Програма використовує файл, відображений в пам'ять, для зберігання ідентифікаторів продуктів, що обмінюються між потоками. Цей механізм дозволяє забезпечити синхронізований доступ до даних і зберігання їх у віртуальній пам'яті.

Даний аналіз програмного коду демонструє використання ефективних механізмів синхронізації для забезпечення коректної взаємодії між потоками в контексті виробничо-споживчої системи.

Код наведено в додатку «А» і «Б».

## **3.3. Тестування**

Розпочнемо наше тестування з використанням вихідних умов: один потік для виробника та один для споживача, що дозволить нам провести детальний аналіз їх взаємодії та ефективності системи у цілому.

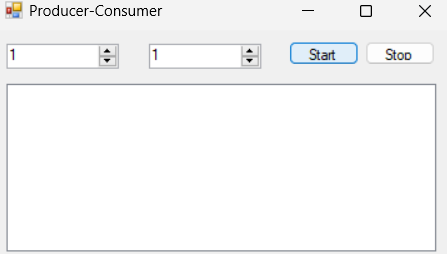


Рис 2. Одинакові дані

Наступний етап тестування: введемо додаткових два виробників до нашої системи, щоб ретельно проаналізувати їх вплив на процес виробництва та споживання продукції. Цей етап тестування спрямований на виявлення можливих відмінностей у роботі системи та оцінку їхнього впливу на загальний хід виконання завдань.

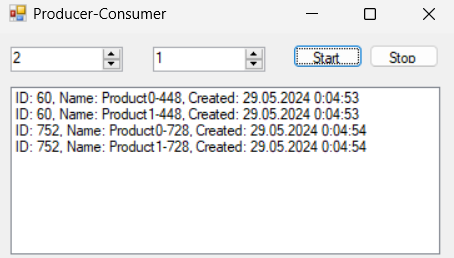


Рис 3. Виробників більше

Для ретельнішого аналізу впливу збільшеної кількості споживачів на нашу систему ми проведемо експеримент, додавши до неї додаткових споживачів. Цей крок дослідження призначений для оцінки ефективності системи в умовах збільшеного обсягу споживання продукції та виявлення можливих відмінностей у її роботі.

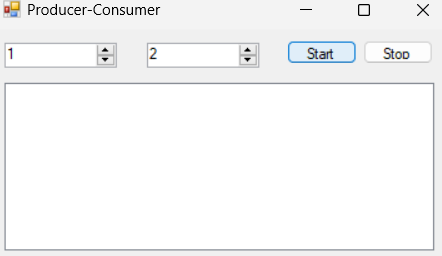


Рис 4. Споживачів більше

# ВИСНОВКИ

Під час виконання курсової роботи було досягнуто мети, що полягала у розробці проекту на мові програмування C#, що ілюструє можливості роботи з потоками, процесами та механізмами синхронізації. Проект включає створення програми, яка здатна змінювати розміри вікна без втрати функціональності, а також здійснює пошук значень байтів у файлах, в тому числі для великих файлів, за допомогою багатопотоковості.

Результати роботи включають наступне:

1. Реалізація основної функціональності програми: Програма успішно змінює розміри вікна та продовжує коректно функціонувати.

2. Використання потоків і процесів: Розроблено рішення, що дозволяє користувачеві встановлювати кількість потоків, що забезпечує гнучкість та адаптивність програми до різних умов експлуатації.

3. Використання семафорів для синхронізації: З метою забезпечення коректної взаємодії між потоками застосовувалися семафори, що дозволило уникнути конфліктів та забезпечити стабільну роботу програми.

4. Використання відображення у пам'яті: Для оптимізації пошуку значення байта у файлі використовувалося відображення у пам'яті, що дозволило ефективно працювати з файлами навіть у випадку великих обсягів даних та використання багатопоточності. Проведено аналіз продуктивності, що підтвердив значне покращення швидкодії при використанні даного механізму.

Дана робота підкреслила важливість використання сучасних методів та технологій системного програмного забезпечення для досягнення високої продуктивності та надійності програмних систем. Отримані знання та навички будуть корисними для подальшого професійного розвитку та вирішення складних задач у реальних проектах. Виконання цього завдання дозволило поглибити розуміння принципів багатопотоковості, синхронізації, а також успішно застосувати ці знання на практиці.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

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

2. Шилдт Г. Полное руководство по С# 4.0.: Пер. с англ. - М.: ООО «И. Д. Вильямc», 2012. - 1047 с.

3. Побегайло А. «Системное программирование в Windows». - СПб.: БХВ-Петербург, 2006. - 1056 с.

4. Overview of synchronization primitives [Електроний ресурс] – Режим доступу. – URL: https://learn.microsoft.com/en-us/dotnet/standard/threading/ overview-of-synchronization-primitives. – (Дата звертання: 20.05.2024).

5.

[Stopwatch Class (System.Diagnostics) [Електроний ресурс] – Режим доступу. – URL: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics .stopwatch ?view=net-8.0 – (Дата звертання: 21.05.2024).](Stopwatch Class (System.Diagnostics) [Електроний ресурс] – Режим доступу. – URL: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics .stopwatch ?view=net-8.0 – (Дата звертання: 21.05.2024). )

[6. Mutex Class (System.Threading) [Електроний ресурс] – Режим доступу. – URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading. mutex?view=net-8.0 – (Дата звертання: 19.05.2024).](6. Mutex Class (System.Threading) [Електроний ресурс] – Режим доступу. – URL:  https://learn.microsoft.com/en-us/dotnet/api/system.threading. mutex?view=net-8.0 – (Дата звертання: 19.05.2024). )

<7. Thread Class (System.Threading) [Електроний ресурс] – Режим доступу. – URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading. thread?view=net-8.0 – (Дата звертання: 18.05.2024).>

# ДОДАТОК А

Файл “Form1.cs”

using System; // Простір імен для основних типів даних і методів

using System.Collections.Concurrent; // Простір імен для потокобезпечних колекцій, таких як ConcurrentQueue

using System.Diagnostics; // Простір імен для діагностики і відстеження

using System.IO.MemoryMappedFiles; // Простір імен для роботи з файлами, відображеними в пам'ять

using System.Threading; // Простір імен для роботи з потоками і синхронізацією

using System.Windows.Forms; // Простір імен для створення Windows Forms додатків

namespace ProducerConsumerApp // Оголошення простору імен для додатку

{

public partial class Form1 : Form // Клас Form1 наслідує клас Form для створення форми

{

private int producerCount = 2; // Кількість потоків-виробників

private int consumerCount = 1; // Кількість потоків-споживачів

private Thread[] producerThreads; // Масив потоків виробників

private Thread consumerThread; // Потік споживача

private SemaphoreSlim emptySemaphore; // Семафор для контролю порожніх місць у черзі

private SemaphoreSlim fullSemaphore; // Семафор для контролю заповнених місць у черзі

private ConcurrentQueue<Product> queue; // Потокобезпечна черга для продуктів

private MemoryMappedFile mmf; // Файл, відображений в пам'ять

private MemoryMappedViewAccessor accessor; // Доступ до відображеного в пам'ять файлу

private CancellationTokenSource cancellationTokenSource; // Токен для скасування роботи потоків

public Form1() // Конструктор класу

{

InitializeComponent(); // Ініціалізація компонентів форми

InitializeSynchronizationObjects(); // Ініціалізація об'єктів синхронізації

InitializeMemoryMappedFile(); // Ініціалізація файлу, відображеного в пам'ять

}

private void InitializeSynchronizationObjects() // Метод для ініціалізації об'єктів синхронізації

{

emptySemaphore = new SemaphoreSlim(10); // Створення семафора з початковим і максимальним значенням 10

fullSemaphore = new SemaphoreSlim(0); // Створення семафора з початковим значенням 0

queue = new ConcurrentQueue<Product>(); // Створення потокобезпечної черги

}

private void InitializeMemoryMappedFile() // Метод для ініціалізації файлу, відображеного в пам'ять

{

mmf = MemoryMappedFile.CreateOrOpen("ProducerConsumerMMF", 1024); // Створення або відкриття файлу розміром 1024 байти

accessor = mmf.CreateViewAccessor(); // Створення доступу до файлу

}

private void StartProducers() // Метод для запуску потоків-виробників

{

producerThreads = new Thread[producerCount]; // Ініціалізація масиву потоків виробників

for (int i = 0; i < producerCount; i++) // Цикл для створення і запуску потоків виробників

{

int producerId = i; // Збереження ідентифікатора виробника

producerThreads[i] = new Thread(() => Producer(producerId, cancellationTokenSource.Token)); // Створення нового потоку для виробника

producerThreads[i].Start(); // Запуск потоку виробника

}

}

private void StartConsumer() // Метод для запуску потоку-споживача

{

consumerThread = new Thread(() => Consumer(cancellationTokenSource.Token)); // Створення нового потоку для споживача

consumerThread.Start(); // Запуск потоку споживача

}

private void Producer(int producerId, CancellationToken token) // Метод, що виконується потоком-виробником

{

Random random = new Random(); // Створення об'єкта для генерації випадкових чисел

try

{

while (!token.IsCancellationRequested) // Цикл, що виконується поки не запитано скасування

{

var product = new Product // Створення нового продукту

{

Id = random.Next(1000), // Встановлення випадкового ідентифікатора

Name = $"Product{producerId}-{random.Next(1000)}", // Встановлення імені продукту

CreationTime = DateTime.Now // Встановлення часу створення продукту

};

emptySemaphore.Wait(token); // Очікування наявності вільного місця у черзі з врахуванням токену скасування

lock (queue) // Блокування черги для безпечного доступу з різних потоків

{

queue.Enqueue(product); // Додавання продукту до черги

Console.WriteLine($"Producer {producerId} produced: {product}"); // Виведення повідомлення у консоль

BeginInvoke(new Action(() =>

{

listBox1.Items.Add(product); // Додавання продукту до ListBox на формі

}));

}

fullSemaphore.Release(); // Збільшення лічильника заповнених місць

Thread.Sleep(1000); // Затримка для симуляції роботи виробника

}

}

catch (OperationCanceledException) // Обробка виключення при скасуванні операції

{

// Коректне завершення роботи

}

}

private void Consumer(CancellationToken token) // Метод, що виконується потоком-споживачем

{

try

{

while (!token.IsCancellationRequested) // Цикл, що виконується поки не запитано скасування

{

fullSemaphore.Wait(token); // Очікування наявності заповненого місця у черзі з врахуванням токену скасування

if (queue.TryDequeue(out Product product)) // Спроба взяти продукт з черги

{

lock (accessor) // Блокування доступу до файлу, відображеного в пам'ять

{

// Зберігаємо лише ID продукту у MemoryMappedFile для простоти

accessor.Write(0, product.Id); // Запис ідентифікатора продукту у файл

}

Invoke(new Action(() =>

{

listBox1.Items.RemoveAt(0); // Видалення першого елементу з ListBox на формі

Console.WriteLine($"Consumer consumed: {product}"); // Виведення повідомлення у консоль

}));

emptySemaphore.Release(); // Збільшення лічильника вільних місць

}

Thread.Sleep(1000); // Затримка для симуляції роботи споживача

}

}

catch (OperationCanceledException) // Обробка виключення при скасуванні операції

{

// Коректне завершення роботи

}

}

private void startButton\_Click(object sender, EventArgs e) // Обробник натискання кнопки "Start"

{

producerCount = (int)producerCountNumericUpDown.Value; // Отримання кількості виробників з NumericUpDown

consumerCount = (int)consumerCountNumericUpDown.Value; // Отримання кількості споживачів з NumericUpDown

cancellationTokenSource = new CancellationTokenSource(); // Створення нового токену для скасування

InitializeSynchronizationObjects(); // Ініціалізація об'єктів синхронізації

StartProducers(); // Запуск потоків виробників

StartConsumer(); // Запуск потоку споживача

}

private void stopButton\_Click(object sender, EventArgs e) // Обробник натискання кнопки "Stop"

{

cancellationTokenSource.Cancel(); // Запит на скасування роботи потоків

foreach (var thread in producerThreads) // Перебір всіх потоків виробників

{

thread?.Join(); // Очікування завершення роботи потоку

}

consumerThread?.Join(); // Очікування завершення роботи потоку споживача

InitializeSynchronizationObjects(); // Оновлення об'єктів синхронізації після зупинки

}

}

public class Product // Клас для представлення продукту

{

public int Id { get; set; } // Ідентифікатор продукту

public string Name { get; set; } // Ім'я продукту

public DateTime CreationTime { get; set; } // Час створення продукту

public override string ToString() // Перевизначення методу ToString для зручного виведення продукту

{

return $"ID: {Id}, Name: {Name}, Created: {CreationTime}"; // Повернення рядка з інформацією про продукт

}

}

}

# ДОДАТОК Б

Файл “Form1.Designer.cs”

namespace ProducerConsumerApp

{

partial class Form1

{

private System.ComponentModel.IContainer components = null;

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

private void InitializeComponent()

{

this.producerCountNumericUpDown = new System.Windows.Forms.NumericUpDown();

this.consumerCountNumericUpDown = new System.Windows.Forms.NumericUpDown();

this.listBox1 = new System.Windows.Forms.ListBox();

this.button1 = new System.Windows.Forms.Button();

this.button2 = new System.Windows.Forms.Button();

((System.ComponentModel.ISupportInitialize)(this.producerCountNumericUpDown)).BeginInit();

((System.ComponentModel.ISupportInitialize)(this.consumerCountNumericUpDown)).BeginInit();

this.SuspendLayout();

//

// producerCountNumericUpDown

//

this.producerCountNumericUpDown.Location = new System.Drawing.Point(13, 13);

this.producerCountNumericUpDown.Name = "producerCountNumericUpDown";

this.producerCountNumericUpDown.Size = new System.Drawing.Size(120, 22);

this.producerCountNumericUpDown.TabIndex = 0;

//

// consumerCountNumericUpDown

//

this.consumerCountNumericUpDown.Location = new System.Drawing.Point(165, 13);

this.consumerCountNumericUpDown.Name = "consumerCountNumericUpDown";

this.consumerCountNumericUpDown.Size = new System.Drawing.Size(120, 22);

this.consumerCountNumericUpDown.TabIndex = 1;

//

// listBox1

//

this.listBox1.FormattingEnabled = true;

this.listBox1.ItemHeight = 16;

this.listBox1.Location = new System.Drawing.Point(13, 53);

this.listBox1.Name = "listBox1";

this.listBox1.Size = new System.Drawing.Size(458, 164);

this.listBox1.TabIndex = 2;

//

// button1

//

this.button1.Location = new System.Drawing.Point(314, 11);

this.button1.Name = "button1";

this.button1.Size = new System.Drawing.Size(75, 23);

this.button1.TabIndex = 3;

this.button1.Text = "Start";

this.button1.UseVisualStyleBackColor = true;

this.button1.Click += new System.EventHandler(this.startButton\_Click);

//

// button2

//

this.button2.Location = new System.Drawing.Point(396, 11);

this.button2.Name = "button2";

this.button2.Size = new System.Drawing.Size(75, 23);

this.button2.TabIndex = 4;

this.button2.Text = "Stop";

this.button2.UseVisualStyleBackColor = true;

this.button2.Click += new System.EventHandler(this.stopButton\_Click);

//

// Form1

//

this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.ClientSize = new System.Drawing.Size(492, 230);

this.Controls.Add(this.button2);

this.Controls.Add(this.button1);

this.Controls.Add(this.listBox1);

this.Controls.Add(this.consumerCountNumericUpDown);

this.Controls.Add(this.producerCountNumericUpDown);

this.Margin = new System.Windows.Forms.Padding(4);

this.Name = "Form1";

this.Text = "Producer-Consumer";

((System.ComponentModel.ISupportInitialize)(this.producerCountNumericUpDown)).EndInit();

((System.ComponentModel.ISupportInitialize)(this.consumerCountNumericUpDown)).EndInit();

this.ResumeLayout(false);

}

private System.Windows.Forms.NumericUpDown producerCountNumericUpDown;

private System.Windows.Forms.NumericUpDown consumerCountNumericUpDown;

private System.Windows.Forms.ListBox listBox1;

private System.Windows.Forms.Button button1;

private System.Windows.Forms.Button button2;

}

}