**ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ОЛЕСЯ ГОНЧАРА**

Кафедра електронних обчислювальних машин

**КУРСОВА РОБОТА**

з дисципліни «Системне програмне забезпечення»

Студента 3 курсу групи КІ-21-2

спеціальність «Комп’ютерна інженерія»

Лебідь О.П.

(прізвище та ініціали)

Керiвник: доцент Матвєєва Н.О.

Кількість балів

Національна шкала

Члени комісії:

(підпис)

(підпис)

(підпис)

м. Дніпро

2024р.

**РЕФЕРАТ**

Курсова робота містить 42 сторінок, 9 ілюстрацій, 7 джерел інформації та 8 лістингів.

Об’єктом дослідження є системне програмне забезпечення, а саме програма на мові C#, розроблена з використанням Microsoft Visual Studio.

Предметом дослідження – підсистема управління технологічним процесом, яка планує виконання процесів, розподіляє процесорний час, створює та руйнує процеси, забезпечує їх необхідними системними ресурсами та підтримує взаємодію між ними.

Метою даної роботи є розробка програми на мові програмування С# з використанням середовища розробки Microsoft Visual Studio.

Одержані висновки: розроблено та реалізувано програмну систему з використанням мови програмування C# та середовища Microsoft Visual Studio, яка дозволяє ефективно управляти технологічними процесами.

Ключові слова: C#, MICROSOFT VISUAL STUDIO, ПОТОКИ, ПРОЦЕСИ, СИНХРОНІЗАЦІЯ, ВІДОБРАЖЕННЯ ФАЙЛУ У ПАМ’ЯТІ, КАНАЛИ ПЕРЕДАЧІ ДАНИХ.

**Зміст**

ВСТУП5

ЗАВДАННЯ НА КУРСОВУ РОБОТУ 7

1. ТЕОРЕТИЧНА ЧАСТИНА11

1.1 Потоки11

1.1.1 Визначення потоку11

1.1.2 Створення та завершення потоку12

1.1.3 Призупинення та відновлення потоку12

1.1.4 Пріоритети потоків13

1.1.5 Визначення моменту закінчення потоку13

1.1.6 Передача аргументу потоку14

1.1.7 Властивість IsBackground14

1.2 Процеси15

1.2.1 Визначення процесу15

1.2.2 Запуск та завершення процесів16

1.3 Синхронізація16

1.3.1 Визначення синхронізації16

1.3.2 М'ютекс17

1.3.3 Семафор18

1.3.4 Події19

1.3.5 Клас Monitor і блокування20

1.3.6 Методи Wait(), Pulse() і PulseAll()20

1.3.7 Атрибут MethodImplAttribute21

1.3.8 Клас Interlocked22

1.4 Відображення файлу у пам’яті23

1.4.1 Концепція механізму відображення файлу у пам’яті23

1.4.2 Класи простору імен System.IO.MemoryMappedFiles23

1.4.3 Постійні зіставлені в пам’яті файли. Метод CreateFromFile()24

1.4.4 Непостійні зіставлені в пам’яті файли. Метод CreateNew()25

1.4.5 Метод OpenExisting()26

1.5 Канали передачі даних27

1.5.1 Способи передачі даних між процесами27

1.5.2 Зв’язки між процесами27

1.5.2 Анонімні канали28

1.5.2 Іменовані канали29

2. ПРАКТИЧНА ЧАСТИНА31

ВИСНОВКИ41

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ43  
ДОДАТКИ44

**ВСТУП**

У сучасному світі технологій, які розвиваються з неймовірною швидкістю, стає все важливішим розуміти та ефективно використовувати мови програмування С# та середовище розробки Microsoft Visual Studio. Програмування, безумовно, є однією з найцінніших навичок у цифрову епоху, і воно впливає на всі аспекти нашого життя.

Мова програмування С#, що використовується для розробки додатків на платформі .NET, є однією з найпопулярніших та найпотужніших мов програмування. Вона дозволяє створювати, оновлювати, змінювати та запускати програми в структурованому форматі. Це робить її незамінною у багатьох сферах, включаючи науку, освіту, медицину, бізнес, тощо.

Неможливо переоцінити важливість С# та Microsoft Visual Studio. Вони використовуються в усіх сферах життя, від наукових досліджень до комерційної діяльності. Вони дозволяють створювати потужні та ефективні програми, спрощуючи доступ до інформації та її обробку.

Використання Microsoft Visual Studio для розробки програм на С# дозволяє використовувати потужні інструменти цієї системи для ефективного програмування. Microsoft Visual Studio – одне з найпопулярніших середовищ розробки, яке використовується у світі, яке пропонує широкий спектр можливостей для роботи з кодом.

С# – це мова програмування для створення надійних, масштабованих та ефективних програм. Вона є ключовим елементом у розробці ефективних програм. Це допомагає зменшити помилки, тим самим забезпечуючи більшу ефективність та надійність програм. Крім того, вона сприяє логічній організації коду, що полегшує його використання та аналіз.

Розробка програм на С# з використанням Microsoft Visual Studio зменшує помилки і робить програми більш ефективними та надійними. Це також полегшує логічну організацію коду, спрощуючи його використання та аналіз. В результаті програма стає більш масштабованою, надійною та ефективною.

Метою даної роботи є розробка програми на мові програмування С# з використанням середовища розробки Microsoft Visual Studio.

**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 12: Підсистема управління технологічним процесом**

Підсистема управління технологічним процесом планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно існуючими процесами в системі, а також займається створенням і руйнуванням процесів, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами.

Основою програми є абстрактний клас Betaproc. Він реалізує абстрактну модель процесу, без конкретизації дій, які потрібно виконати, з набором змінних і методів, загальних для всіх процесів. Три процеси sinProc, FibbonProc і ProcRandom, походять від цього класу, і кожен реалізує тільки методи, які повертають тип процесу і сам метод, виконаний процесом. При цьому кожен метод Base, який містить виконуваний код, має загальний дескриптор у своєму тілі, який дозволяє лише одному процесу виконувати код, коли інші стоять у черзі та отримують час виконання пропорційно пріоритету. У момент запуску запускається таймер, який забезпечує однаковий квант часу в 3 секунди для кожного процесу.

Однак процеси не висять в пам'яті комп'ютера просто так. Клас ProcManager був створений спеціально для управління процесами. Коли процес створений, він і вся інформація про нього вноситься в масив, і відповідно до номера осередку, в яку записаний процес, йому присвоюється ідентифікатор, за яким до нього можна отримати доступ в процесі роботи. Клас ProcManager також реалізує графічне представлення в пам'яті. Кожен важливий елемент процесу відображається в спеціальній таблиці на бланку, і при зміні одного з них викликається подія, яка змінює запис в таблиці в режимі реального часу, завдяки чому можна спостерігати, як мітка «працює» проходить від одного процесу до іншого.

Панель «Процес» дозволяє вибрати процес, який хочемо розпочати. Він має три точки: числа Фібоначчі, випадкові числа та рекурсивний синус.

Область «Пріоритет» дозволяє встановити пріоритет процесу для початку. Має 5 пунктів: низькій, нижче середнього, середній, вище середнього, високий.

Кнопка «Пуск» використовується для запуску процесу з обраними параметрами.

Таблиця в центрі вікна відображає стан кожного з запущених процесів. Після додавання кожного процесу до нього автоматично додається один рядок. Має 5 полів:

1) Номер процесу – показує порядковий номер процесу

2) Тип процесу – показує дію яку цей процес виконує

3) Статус процесу – показує, запущено процес чи ні. Також показує, чи зупинено, завершено або просто відновлено процес.

4) Пріоритет процесу – показує пріоритет процесу, який був призначений йому при його створенні.

5) Відсоток процесорного часу – відображає відсоток використаного процесорного часу.

У правій частині вікна програми є поле, в якому записується результат виконання працюючого процесу в режимі реального часу.

У нижній частині вікна програми є елементи управління для роботи з вже працюючими процесами.

Кнопка "Стоп" – зупиняє обраний процес з можливістю його подальшого перезапуску.

кнопка "Відновити" – перезапускає зупинений процес.

кнопка «Готово» – зупиняє обраний процес без можливості його завершення.

Програма також має стандартний рядок стану, за допомогою якого можна згортати, розгортати або закривати робоче вікно програми.

Процеси запускаються за допомогою кнопки «Пуск», але спочатку потрібно встановити параметри процесу.

Випадкове число – процес генерує випадкове число в діапазоні від одного до ста і відображає його у вихідному полі.

Числа Фібоначчі – генерує послідовність Фібоначчі, починаючи з першого члена, і відображає їх у вихідному полі. Як тільки значення порядкових номерів перевищить тисячу, значення скидаються до перших доданків.

Рекурсивний синус – породжує значення синуса X. Спочатку X дорівнює 1, пізніше йому присвоюються обчислені значення sin(x). Значення, округлені до третього символу, відображаються в полі виводу.

Пріоритет процесу показує ймовірність початку процесу, якщо попередній процес завершив свою роботу. Наприклад, якщо запустити три процеси з однаковим пріоритетом, кількість разів їх запуску буде приблизно однаковою, а якщо запустити два процеси з високим пріоритетом і один процес з низьким пріоритетом, то процес із низьким пріоритетом виконуватиметься приблизно один із шістнадцяти разів. Однак архітектура програми передбачає, що якщо кількість запущених процесів більше одиниці, то один і той же процес не може виконуватися більше одного разу поспіль.

1. **ТЕОРЕТИЧНА ЧАСТИНА**

**1.1 Потоки**

**1.1.1 Визначення потоків**

**Потоки (Thread)** – це незалежні послідовності інструкцій у програмі, які дозволяють виконувати різні завдання паралельно. Кожен процес може містити один або кілька потоків, які заплановано для виконання операційною системою. Потоки поділяються на пріоритетні та фонові. Пріоритетний потік стає фоновим, якщо всі пріоритетні потоки в його процесі призупинені. Багатозадачність може здійснюватися на рівні процесів (для одночасного виконання кількох програм) або на рівні потоків (для паралельного виконання окремих частин однієї програми).

Потоки можуть перебувати в різних станах: виконуються, готові до виконання, призупинені, заблоковані або завершені. Для координації виконання потоків використовується синхронізація, яка забезпечує коректну взаємодію між потоками.

У будь-якій програмі завжди є принаймні один потік, який називають основним. Основний потік може створювати додаткові потоки. У мові програмування C# та середовищі .NET підтримується як багатозадачність на основі процесів, так і на основі потоків. Основним класом для багатопотокового програмування – це клас **Thread**. При запуску додатка на C#, операційна система створює процес, а середовище CLR (Common Language Runtime) створює домен додатків, у якому працює додаток. Хоча потоки можуть охоплювати межі доменів додатків, в кожен момент часу кожен потік виконується тільки в одному домені додатків. Для отримання поточного контексту, в якому виконується потік, використовується статична властивість **Thread.CurrentContext**.

Потоки є зручним уніфікованим програмним інтерфейсом для читання або запису файлів (включаючи спеціальні файли, особливо пов’язаних з пристроями), сокетів та передачі даних між процесами.

Абстракція потоку особливо важлива в мові програмування C, де він представляє собою джерело вводу та/або виводу даних (зазвичай байтів), пов’язаних з файлом, пристроєм, або іншим процесом. Робота з потоками було перенесено на багато інших мов: C++: iostream зі стандартної бібліотеки C++.

**1.1.2 Створення та завершення потоку**

Щоб створити потоки в C# використовується клас **Thread** з простору імен **System.Threading**. Методи, які використовуються як точки входу потоку, повинні повертати **void** і не приймати параметрів. Після створення об'єкта **Thread** потік починає виконання, викликаючи свій метод **Start()**. Потік можна зупинити до його нормального завершення за допомогою методу **Abort()**, який генерує виняток **ThreadAbortException**. Однак, починаючи з .NET 5, метод **Thread.Abort** позначений як застарілий. Запит на передчасне завершення потоку може бути відмінений самим потоком за допомогою методу **ResetAbort()**.

Існують інші способи управління потоками та обробки багатозадачності. Зокрема, для сучасного багатопотокового програмування рекомендується використовувати класи та методи з простору імен **System.Threading.Tasks**, такі як **Task**, які спрощують створення та керування асинхронними операціями. **Task** забезпечує більш ефективну обробку паралельних задач і забезпечує кращу підтримку для обробки винятків і скасування операцій.

**1.1.3 Призупинення та відновлення потоку**

У ранніх версіях .NET Framework була можливість призупиняти потік за допомогою методу **Thread.Suspend()** та відновлювати його за допомогою методу **Thread.Resume**. Однак через можливі проблеми з безпекою та взаємоблокуванням ці методи вважаються застарілими, і їх використання не рекомендується. Використання цих методів може спричинити призупинення потоку на тривалий період часу або призвести до неправильної роботи програми.

Замість цього рекомендується використовувати інші засоби синхронізації, такі як м'ютекси (**Mutex**) та семафори (**Semaphore**). Ці механізми дозволяють керувати доступом до критично важливих ресурсів і уникати перегонів даних між потоками. Крім того, розробники можуть використовувати інші прийоми, такі як блокування потоку за допомогою змінних умов (**Monitor.Wait()** та **Monitor.Pulse()**) або використання асинхронних підходів та обіцянок (**async/await**) для управління паралельними операціями.

**1.1.4 Пріоритети потоків**

Кожен потік має власний пріоритет, який визначає, яку частину обчислювальних ресурсів він отримує від центрального процесора (CPU). Як правило, потоки з вищим пріорітетом отримують більше часу від CPU, ніж потоки з нижчим пріоритетом. Пріоритет потоку можна змінювати за допомогою властивості **Priority** класу **Thread**.

Коли високопріоритетний потік очікує доступу до спільних ресурсів або виконання певної операції, він може заблокуватися, і замість нього може виконуватися фоновий потік. Це забезпечує ефективніше використання обчислювальних ресурсів та покращити загальну продуктивність системи.

Крім того, варто враховувати, що планування завдань на рівні операційної системи також впливає на час, який виділяється для виконання кожного потоку. Операційна система має власний механізм планування, який визначає, який потік матиме доступ до CPU в певний момент часу. Це впливає на продуктивність і відзначається динамікою та структурою самої операційної системи.

**1.1.5. Визначення моменту закінчення потоку**

У класі **Thread** є два основні методи для визначення моменту закінчення потоку:

1. Властивість **IsAlive**: Ця властивість повертає значення true, якщо потік все ще виконується, і false, якщо він завершив роботу. Зазвичай використовується для перевірки стану виконання потоку. Наприклад: **if (myThread.IsAlive) { /\* виконувати дії \*/ }**.

2. Метод **Join()**: Цей метод дозволяє поточному потоці чекати на завершення іншого потоку. Він блокує виконання поточного потоку до того моменту, поки потік, для якого він був викликаний, не завершиться. Якщо потік не було розпочато, він генерує виключення **ThreadStateException**. Крім того, в методі **Join()** можна вказати максимальний час очікування завершення потоку, щоб уникнути блокування надто тривалих операцій. Найпростіша форма цього методу виглядає так: **myThread.Join()**.

**1.1.6. Передача аргументу потоку**

Метод **Start()** використовується для передачі аргументу потоку. Об'єкт, що передається як аргумент методу **Start()**, автоматично передається методу, який виконує роль точки входу в потік. Це означає, що дані можуть передаватися в потік через параметр методу, який виконується у цьому потоці.

Для використання параметризованої форми методу **Start()**, потрібно використати конструктор класу **Thread**, який приймає делегат **ParameterizedThreadStart**. **ParameterizedThreadStart** – це делегат, який приймає аргумент типу **object**. Тобто, у методі, який є точкою входу в потік, повинен мати параметр типу **object**, через який буде передаватися аргумент.

Цей механізм дозволяє передавати дані в потік із зовнішнього середовища, що дозволяє забезпечити більш гнучку та параметризовану поведінку потоків у програмі.

**1.1.7 Властивість IsBackground**

У середовищі .NET існують два типи потоків: пріоритетні та фонові. Процес не завершиться, поки не будуть завершені всі пріоритетні потоки. Фонові потоки автоматично завершуються після закінчення всіх пріоритетних потоків або якщо головний процес завершує свою роботу, навіть якщо фоновий потік був ще не закінченим.

За замовчуванням створюваний потік є пріоритетним. Однак його можна зробити фоновим, встановивши властивість **IsBackground** класу **Thread** в значення **true**. Таким чином, такий підхід дозволяє автоматично завершити фоновий потік, якщо він єдиний залишився в процесі виконання.

Встановлення значення **false** для властивості **IsBackground** вказує на те, що потік є пріоритетним, тобто його робота не буде завершена автоматично, поки не завершаться всі інші потоки або процеси.

**1.2 Процеси**

**1.2.1 Визначення процесу**

Багатозадачність на основі потоків часто використовується в програмуванні на C#, але багатозадачність на основі процесів також може бути організована. Для цього використовується клас **Process**, визначеного в просторі імен **System.Diagnostics**, який дозволяє запускати виконання іншої програми з поточної. У Windows процес представляє собою об'єкт ядра, який має системні ресурси, необхідні для запуску додатка.

Виконання процесу починається з його первинного потоку і закінчується після завершенні всіх його потоків. Кожен процес у Windows має свій унікальний ідентифікатор, який дозволяє ідентифікувати його в системі. У деяких випадках для зміни характеристик процесу, таких як пріоритет, потрібно знати його дескриптор. Дескриптор процесу можна використовувати лише в межах поточного процесу і не може бути успадкований іншими процесами.

Загальний використання класу **Process** дозволяє взаємодіяти з процесами на операційній системі, здійснювати їх запуск, управляти їхнім життєвим циклом та отримувати інформацію про них. Це дозволяє створювати різноманітні додатки, що взаємодіють з іншими програмами та контролюють їх роботу.

**1.2.2 Запуск та завершення процесів**

Метод **Start()** класу **Process** використовується для запуску нового процесу вказаного файлу. Ім'я виконуваного файлу передається як аргумент цього методу. Після запуску нового процесу, процес, який його створив, називається батьківським, а сам створений процес – дочірнім.

Після завершення роботи дочірнього процесу, рекомендується викликати метод **Close()**, щоб звільнити ресурси, які були виділені для цього процесу. Це важливо для ефективного управління ресурсами та пам'яттю комп'ютера.

**1.3 Синхронізація**

**1.3.1 Визначення синхронізації**

**Синхронізація процесів** – це процес забезпечення взаємодії та координації між двома або більше процесами, щоб уникнути конфліктів або взаємних блокувань при роботі зі спільними ресурсами. В контексті багатопотокового програмування, синхронізація використовується для забезпечення правильного порядку виконання дій та доступу до спільних ресурсів між потоками.

Потоки, що належать до одного процесу, можуть спільно використовувати різні ресурси, такі як пам'ять та файли. Механізми синхронізації включають в себе безліч об'єктів операційної системи, які допомагають координувати доступ до цих ресурсів. Серед таких об'єктів можна виділити взаємовиключення (**Mutex**), події (**Event**), семафори (**Semaphore**), блокування (**Lock**), монітори (**Monitor**) тощо.

Кожен з цих об'єктів може перебувати у сигнальному або несигнальному стані, що дозволяє потокам перевіряти стан об'єкта та чекати на його зміну перед продовженням виконання дій. Ефективне використання механізмів синхронізації може підтримувати коректну та безпечну роботу багатопотокових програм, забезпечуючи взаємодію між потоками без виникнення конфліктів або ситуацій взаємного блокування.

**1.3.2 М'ютекс**

**М'ютекс (Mutex)** – це механізм синхронізації, який дозволяє тільки одному потоку отримувати доступ до спільного ресурсу в будь-який момент часу, запобігаючи конфліктам з іншими потоками. Це важливий інструмент для управління доступом до ресурсів у багатопотокових або багатопроцесних програмах, де одночасний доступ може призвести до некоректної роботи програми або втрати цілісності даних.

Існує два типи м'ютексів: іменований і анонімний. Іменований м'ютекс доступний для кількох процесів і може використовуватися для синхронізації доступу до ресурсів між різними програмами. Анонімний м'ютекс доступний лише в рамках одного процесу і призначений для координації потоків цього процесу.

Для отримання м'ютекса і блокування доступу до ресурсу використовується метод **WaitOne()**. Якщо м'ютекс доступний, потік отримує його і може продовжувати роботу з ресурсом. Після завершення роботи з ресурсом потік повинен звільнити м'ютекс за допомогою методу **ReleaseMutex()**. Ці методи забезпечують одночасний доступ до спільних ресурсів лише одному потоку, що дозволяє уникнути конфліктів та проблем зі спільними даними.

Крім того, м'ютекси можуть використовуватися для реалізації більш складних схем синхронізації, таких як запобігання дедлокам (взаємним блокуванням), забезпечення порядку доступу до ресурсів та контроль за виконанням критичних секцій коду.

У .NET, м'ютекси також можуть бути використані для синхронізації доступу до ресурсів у розподілених системах, де різні процеси можуть працювати на різних фізичних або віртуальних машинах. Це робить м'ютекси гнучким і потужним інструментом для розробки складних багатопотокових і багатопроцесних додатків.

**1.3.3 Семафор**

**Семафор (Semaphore)** – це механізм синхронізації, який дозволяє декільком потокам отримувати доступ до спільних ресурсів одночасно. Він використовує лічильник для контролю кількості дозволів на доступ. Якщо значення лічильника більше нуля, доступ дозволяється, в іншому випадку він заборонений. Потік, який потребує доступу, намагається отримати дозвіл від семафора. Якщо дозвіл надано, лічильник зменшується. Після того, як доступ більше не потрібний, потік звільняє дозвіл, і лічильник збільшується.

Семафори є особливо корисними для контролю доступом до групи ресурсів, таких як мережеві з'єднання або пул потоків. Їх можна використовувати для обмеження кількрсиі одночасних доступів до ресурсу, тим самим запобігаючи його перевантаженню.

У фреймворку .NET існують два основних класи семафорів: **Semaphore** та **SemaphoreSlim**.

**Semaphore** може бути іменованим, що дозволяє використовувати його для синхронізації між різними процесами. Іменовані семафори можуть бути створені з унікальним ім'ям, яке дозволяє кільком процесам спільно використовувати один і той самий семафор.

**SemaphoreSlim** – це полегшена версія семафора, оптимізована для короткого часу очікування і зниженої накладної вартості. **SemaphoreSlim** використовує менш ресурсоємні методи для контролю доступу до ресурсу і є більш ефективним у випадках, коли потрібно часто отримувати і звільняти доступ.

Обидва класи підтримують методи **Wait()** для отримання дозволу та **Release()** для звільнення дозволу, що забезпечує гнучке управління доступом до ресурсів у багатопотоковому середовищі.

**1.3.4 Події**

Події в .NET відіграють важливу роль у синхронізації потоків на системному рівні. Вони дозволяють одному потоку чекати на виникнення подій в іншому потоці, забезпечуючи ефективне управління багатопотоковими операціями. До основних класів, які забезпечують таку функціональність, належать **ManualResetEvent**, **AutoResetEvent**, **ManualResetEventSlim** та **CountdownEvent**.

Клас **ManualResetEvent** дозволяє одному або декільком потокам очікувати на події. Потік, що очікує події, викликає метод **WaitOne()**, який блокує його до тих пір, поки інший потік не викличе метод **Set()**, встановлюючи подієвий об'єкт у сигнальний стан. Після цього всі очікуючі потоки можуть продовжити своє виконання. Метод **Reset()** повертає об'єкт у несигнальний стан.

Клас **AutoResetEvent** також дозволяє одному або більше потокам чекати на подію, але після встановлення сигнального стану за допомогою методу **Set()**, подія автоматично переходить у несигнальний стан, як тільки один з очікуючих потоків отримує повідомлення про подію і продовжує своє виконання. Це дозволяє гарантувати, що тільки один потік продовжить виконання після спрацьовування події.

**ManualResetEventSlim** – це оптимізована версія **ManualResetEvent**, яка забезпечує більш ефективне управління ресурсами і працює швидше при коротких періодів очікування.

**CountdownEvent** – це клас, який дозволяє одному або кільком потокам чекати, поки певна кількість операцій, що виконуються в інших потоках, не завершиться. Кількість очікуваних операцій встановлюється при створенні об'єкта **CountdownEvent**. Метод Signal() зменшує лічильник очікуваних операцій на одиницю, а коли лічильник досягає нуля, очікуючі потоки можуть продовжити виконання.

**1.3.5 Клас Monitor і блокування**

У мові програмування C# ключове слово **lock** використовується як простий і зручний спосіб доступу до засобів синхронізації. Ці засоби визначені в класі **Monitor**, що знаходиться в просторі імен **System.Threading**.

Клас **Monitor** містить кілька методів для управління синхронізацією. Наприклад, метод **Enter()** використовується для отримання блокування об'єкта, а метод **Exit()** – для зняття блокування. Якщо об'єкт заблокований іншим потоком, метод **Enter()** блокує виклик потік до тих пір, поки об'єкт не буде доступним.

Однак методи **Enter()** і **Exit()** використовуються рідко, оскільки оператор **lock** автоматично надає еквівалентні засоби синхронізації потоків. Використання оператора **lock** спрощує процес блокування об'єкта, що робить його більш зручним для програмування на C#. Оператор **lock** забезпечує автоматичне звільнення блокування об'єкта, навіть якщо в блоці коду виникає виняток, завдяки чому код стає більш безпечним і стійким.

Крім того, в класі **Monitor** визначено метод **TryEnter()**, який повертає true, якщо потік отримує блокування для об'єкта, і false в іншому випадку. Цей метод допомагає реалізувати альтернативні варіанти синхронізації потоків, коли необхідні об'єкти тимчасово недоступні. **TryEnter()** може також приймати параметр тайм-ауту, що дозволяє вказати максимальний час очікування блокування перед тим, перш ніж повернеться значення false.

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

Клас **Monitor**, який є частиною простору імен **System.Threading** у C#, надає методи **Wait()**, **Pulse()**, і **PulseAll()**, що використовуються для складної синхронізації потоків. Ці методи можна викликати лише з заблокованого фрагмента коду.

Коли потік тимчасово блокується, він викликає метод **Wait()**, який переводить потік у стан очікування і звільняє блокування об'єкта. Це дозволяє іншим потокам отримати доступ до цього об'єкта. Після виклику **Wait()**, потік може бути активований, коли інший потік викликає метод **Pulse()** або **PulseAll()**.

Метод **Pulse()** відновлює виконання першого потоку в черзі очікування на отримання блокування. Метод **PulseAll()**, навпаки, сигналізує всім потокам, які чекають зняття блокування, дозволяючи їм продовжити виконання.

Метод **Wait()** має дві форми: одна просто очікує, поки потік не отримає повідомлення про звільнення об'єкта, а інша дозволяє вказати період очікування в мілісекундах. Якщо вказаний період часу закінчується, потік автоматично пробуджується навіть без сигналу.

Важливо відзначити, що методи **Wait()**, **Pulse()**, і **PulseAll()** повинні викликатися тільки з коду, що знаходиться всередині блоку синхронізації, наприклад, всередині блоку **lock**. Якщо ці методи викликаються поза межами синхронізованим кодом, виникає виняток **SynchronizationLockException**.

Це означає, що для успішного використання методів **Wait()**, **Pulse()**, і **PulseAll()** потрібно:

1. Забезпечити, що потік, який викликає ці методи, вже володіє блокуванням об'єкта.

2. Використовувати методи всередині блоку **lock** або іншого синхронізованого коду, щоб уникнути **SynchronizationLockException**.

**1.3.7 Атрибут MethodImplAttribute**

У мові програмування C# можна повністю синхронізувати метод за допомогою атрибута **MethodImplAttribute**. Цей атрибут надає альтернативний метод синхронізації методів порівняно з використанням оператора **lock**. Він дозволяє заблокувати весь метод, що може бути корисно у випадках, коли потрібно забезпечити доступ до методу тільки одному потоку одночасно.

Атрибут **MethodImplAttribute** визначено в просторі імен **System.Runtime.CompilerServices**. Для синхронізації методу достатньо вказати атрибут **MethodImplOptions.Synchronized**. Цей атрибут блокує весь метод для поточного екземпляра об'єкта, доступного за посиланням **this**. Для статичних методів блокується сам тип.

Важливо зазначити, що атрибут **MethodImplAttribute** має власні обмеження і властивості. Наприклад, він не підходить для використання в відкритих об'єктах або класах через свою поведінку блокування всього методу. Слід також зазначити, що така синхронізація може призвести до затримок виконання методу в разі, якщо інші потоки намагаються отримати доступ до нього одночасно.

**1.3.8 Клас Interlocked**

Клас **Interlocked** у мові програмування C# – це потужний інструмент для виконання безпечних операцій зі спільними змінними в багатопотоковому середовищі. Його використання є ефективною альтернативою іншим методам синхронізації, особливо коли потрібно виконати операції над загальними змінними, таких як збільшення, зменшення, порівняння та обмін значеннями, без необхідності ручного управління блокуванням.

Однією з головних особливостей **Interlocked** є те, що його методи гарантують атомарність операцій, тобто вони виконуються як єдина, неподільна операція, без можливості переривання потоком. Це забезпечує надійність і безпеку в багатопотокових додатках, оскільки можлівсть внесення змін під час виконання операції.

Клас **Interlocked** надає широкий спектр статичних методів для виконання різних операцій зі спільними змінними, включаючи атомарне додавання, інкрементування, декрементування, порівняння та обмін значеннями. Також доступна підтримка для 64-разрядних операцій, що робить **Interlocked** дуже потужним інструментом для обробки великих обсягів даних у багатопотоковому середовищі.

**1.4 Відображення файлу у пам’яті**

**1.4.1 Концепція механізму відображення файлу у пам’яті**

Механізм відображення вмісту файлу у віртуальну пам'ять процесу є важливим для багатьох операцій в операційних системах, особливо для динамічного завантаження бібліотек і обміну даними між процесами. Механізм заснований на створенні ОС файлів підкачки з віртуальними сторінками, які можуть бути відображені в адресний простір процесу.

У системі Windows механізм відображення файлів в адресний простір процесу дозволяє переглядати не тільки вміст файлів підкачки, але і вміст звичайних файлів. Такий файл, відображений у пам'ять процесу, називається файловим видом або просто видом файлу.

Одні й ті ж самі файли можуть бути відображені в адресний простір кількох процесів одночасно, і ОС гарантує, що вміст файлу залишається узгодженим для всіх процесів. Це дає змогу завантажувати програми та динамічно підключати бібліотеки функцій під час їх виконання.

Одним з основних застосувань цього механізму є завантаження програми на виконання в адресному просторі процесу і динамічне підключення бібліотек функцій під час виконання цієї програми. Крім того, він дозволяє здійснювати обмін даними між процесами, враховуючи, що система забезпечує узгодженість даних в файлі, який відображається в пам'ять.

**1.4.2 Класи простору імен System.IO.MemoryMappedFiles**

Простір імен **System.IO.MemoryMappedFiles** надає доступ до класів, які дозволяють працювати зі зіставленими в пам'ять файлами. Основними класами цього простору імен – **MemoryMappedFile**, **MemoryMappedFileSecurity**, **MemoryMappedViewAccessor** та **MemoryMappedViewStream**.

**MemoryMappedFile** використовується для представлення зіставленого в пам'ять файлу, що дозволяє зчитувати та записувати дані у великі файли, без необхідності читати або записувати весь файл.

**MemoryMappedFileSecurity** дозволяє встановлювати права доступу до зіставлених в пам'ять файлів, контролюючи, які операції можуть виконуватись і хто може їх виконувати.

**MemoryMappedViewAccessor** та **MemoryMappedViewStream** надають зручні способи доступу до зіставлених в пам'ять файлів з різними рівнями функціональності. Перший дозволяє працювати з файлом як з рядком байтів, забезпечуючи довільний доступ до даних. Другий надає можливість роботи з файлом як з потоком, спрощуючи послідовне читання або запис даних.

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

Постійні файли, які використовуються як зіставлені у пам'яті файли, насправді пов'язані з вихідним файлом на диску. Це означає, що дані у вихідному файлі будуть збережені на диску, коли останній процес завершує роботу з файлом. Ці файли дають можливість ефективно оброблятт з дуже великими обсягами даних.

Метод **CreateFromFile** в класі **System.IO.MemoryMappedFiles.MemoryMappedFile** використовується для створення зіставленого у пам'яті файлу на основі вже існуючого файлу на диску або файлового потоку (за допомогою класу **FileStream**).

Цей метод має декілька перевантажень:

* **CreateFromFile(String)**: створює файл, зіставлений у пам’ять, із указаного шляху до існуючого файлу на диску.
* **CreateFromFile(String, FileMode)**: створює зіставлений у пам'яті файл із заданим режимом доступу, використовуючи файл на диску.
* **CreateFromFile(String, FileMode, String)**: створює зіставлений у пам'яті файл з вказаним режимом доступу, ім'ям та файлом на диску.
* **CreateFromFile(String, FileMode, String, Int64)**: створює зіставлений у пам'яті файл з вказаним режимом доступу, ім'ям, розміром та файлом на диску.
* **CreateFromFile(String, FileMode, String, Int64, MemoryMappedFileAccess)**: створює зіставлений у пам'яті файл з вказаним режимом доступу, ім'ям, розміром, типом доступу та файлом на диску.

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

Непостійні файли є зіставленими в пам'яті файлами, які не пов'язані з фізичним файлом на диску. Коли останній процес завершує роботу з таким файлом, дані втрачаються, а файл видаляється функцією збору сміття. Ці файли особливо корисні для створення спільної пам'яті для взаємодії між процесами (IPC).

Метод **CreateNew** в класі **System.IO.MemoryMappedFiles.MemoryMappedFile** створює зіставлений у пам’яті файл з вказаною ємністю в системній пам’яті. Цей метод приймає ім'я для файлу, зіставленого в пам'яті, та максимальний розмір у байтах, який буде виділено для цього файлу.

Метод **CreateNew** має декілька перевантажень, які дозволяють створювати зіставлений у пам'яті файл з різними параметрами, зокрема:

- Режим доступу.

- Опції виділення пам'яті.

- Права безпеки.

- Можливість успадковування.

Ці перевантаження забезпечують гнучкість у створенні зіставлених у пам'яті файлів для роботи з великими обсягами даних та забезпечення ефективного обміну даними між процесами.

Основні класи простору імен **System.IO.MemoryMappedFiles** включають:

1. **MemoryMappedFile**: Представляє файл, зіставлений у пам'яті, дозволяючи програмістам працювати з великими файлами шляхом відображення їх у віртуальну пам'ять процесу.

2. **MemoryMappedFileSecurity**: Керує дозволами доступу до зіставлених у пам'яті файлів, визначаючи права доступу та перевіряючи спроби доступу.

3. **MemoryMappedViewAccessor**: Надає доступ до зіставленого у пам'яті файлу з довільним доступом, дозволяючи читати та записувати дані безпосередньо у пам'ять.

4. **MemoryMappedViewStream**: Надає потік, що дозволяє послідовний доступ до зіставленого у пам'яті файлу, полегшуючи роботу з файлами як зі звичайними потоками.

**1.4.5 Метод OpenExisting()**

Метод **OpenExisting** класу **System.IO.MemoryMappedFiles.MemoryMappedFile** використовується для отримання об'єкта вже створеного зіставленого в пам'яті файлу. Цей файл може бути постійним або тимчасовим, і цей метод дозволяє отримати доступ до нього, використовуючи задане ім'я в системній пам'яті.

Метод **OpenExisting** має декілька перевантажень, які дозволяють відкривати існуючий зіставлений в пам'яті файл з різними параметрами. Цим параметрам належать права доступу та параметри успадковування, що забезпечують гнучкість у використанні зіставлених в пам'яті файлів. Це робить можливим ефективну роботу з великими файлами та організацію обміну даними між різними процесами.

Основні перевантаження методу включають:

1. **OpenExisting(String)** – відкриває наявний зіставлений в пам'яті файл за його іменем.

2. **OpenExisting(String, MemoryMappedFileRights)** – дозволяє вказати права доступу при відкритті зіставленого в пам'яті файлу.

3. **OpenExisting(String, MemoryMappedFileRights, HandleInheritability)** – дозволяє вказати додатковіпараметри успадковування дескрипторів.

Кожне з цих перевантажень дозволяє налаштовувати відкриття файлу під конкретні потреби, що значно розширює можливості використання зіставлених в пам'яті файлів у різноманітних сценаріях.

**1.5 Канали передачі даних**

**1.5.1 Способи передачі даних між процесами**

Обмін даними між паралельними процесами вимагає спеціальних механізмів для передачі даних від одного потоку (відправника) до іншого (одержувача), особливо коли ці потоки працюють у різних процесах. Коли потоки функціонують в одному процесі, можна використовувати глобальні змінні та механізми синхронізації для обміну даними. Але якщо потоки належать до різних процесів, потрібні спеціальні засоби операційної системи.

Один із способів здійснення цього обміну – створення каналу передачі даних, який включає в себе вхідний та вихідний буфери пам`яті, потоки ядра операційної системи та загальну пам'ять, до якої мають доступ обидва потоки ядра. Користувацькі потоки отримують доступ до цих буферів через виклики функцій ядра операційної системи. Таким чином, передача даних з відправника до одержувача відбувається через послідовність взаємодіючих потоків, які обмінюються даними через спільну пам'ять.

**1.5.2 Зв’язки між процесами**

Для передачі даних між процесами необхідно встановити зв'язок між ними, який може мати фізичний або логічний характер. Під фізичним зв'язком розуміється пряме з'єднання апаратних пристроїв, тоді як логічний передбачає використання програмних механізмів. Основні види зв'язків є напівдуплексний, де передача даних можлива лише в одному напрямку, та дуплексний, де передача може відбуватися в обидва напрямки одночасно.

Топологія зв'язку визначає конфігурацію взаємозв'язків між процесами-відправниками та адресатами, може бути у форматі 1 —> 1, 1 —> N, N —> 1 або N —> М. При розробці системи обміну даними спочатку обирається топологія зв'язків та визначаються напрямки передачі даних по цих зв'язках.

Після цього в програмах реалізовуються обрані зв'язки між процесами за допомогою спеціальних функцій операційної системи, призначених для встановлення зв'язку між процесами. Система передачі даних, яка зазвичай вбудована у ядро операційної системи, надає функції для обміну даними між процесами. Тому встановлення та керування зв'язками між процесами є ключовим етапом у розробці систем обміну даними.

**1.5.3 Анонімні канали**

Анонімні канали, що забезпечують взаємодію між процеснами на локальному комп'ютері, споживають менше системних ресурсів, порівняно з іменованими каналами. Вони односпрямовані та безіменні, тому не можуть використовуватися для мережевої взаємодії. Ці канали зручно використовувати для координації взаємодії між потоками або між батьківськими та дочірніми процесами.

У .NET анонімні канали реалізуються за допомогою класів **AnonymousPipeServerStream** і **AnonymousPipeClientStream**. Вони допомагають встановити безпечні та захищені зв'язки між дочірнім та батьківськими процесами. Клієнтська сторона анонімного каналу створюється за допомогою дескриптора каналу, який передається серверною стороною каналу.

Об'єкт класу **AnonymousPipeServerStream** повинен знищити копію дескриптора клієнта за допомогою методу **DisposeLocalCopyOfClientHandle**, щоб бути сповіщеним про наявність клієнта. Конструктори класу **AnonymousPipeServerStream** та **AnonymousPipeClientStream** дозволяють ініціалізувати нові екземпляри цих класів з різними параметрами, включаючи напрямок каналу, режим успадкування, розмір буфера, захист каналу та дескриптори каналу. Таким чином, анонімні канали є важливим інструментом для організації міжпроцесної взаємодії.

**1.5.4 Іменовані канали**

Іменовані канали відіграють ключову роль у міжпроцесному зв’язку між сервером та одним або кількома клієнтами. Вони можуть бути налаштовані як однонаправлені, так і двонаправлені, дозволяючи обмінюватися повідомленнями. Ці канали дозволяють декільком клієнтам одночасно підключатися до серверного процесу, використовуючи одне ім'я каналу. Крім того, іменовані канали підтримують індивідуальні налаштування, що дозволяють підключеним процесам мати власні дозволи на віддалених серверах.

У .NET іменовані канали реалізовані за допомогою класів **NamedPipeServerStream** і **NamedPipeClientStream**. Ці канали можуть використовуватися для здійснення зв'язку між процесами як локально, так і через мережу. Те саме ім'я каналу може використовуватися декількома об'єктами **NamedPipeClientStream**. Будь-який процес може діяти як сервер іменованого каналу, як клієнт або і як сервер і як клієнт одночасно.

Конструктори класу **NamedPipeServerStream** та **NamedPipeClientStream** дозволяють ініціалізувати нові екземпляри з різними параметрами, включаючи ім'я каналу, напрямок каналу, максимальну кількість екземплярів сервера, режим передачі та опції каналу.

1. **ПРАКТИЧНА ЧАСТИНА**

Підсистема управління технологічним процесом планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно існуючими процесами в системі, а також займається створенням і руйнуванням процесів, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами.

Для початку додаємо до конструктора потрібні елементи. Ось що додано:

1. Чотири кнопки:

- startButton (Пуск)

Кнопка «Пуск» використовується для запуску процесу з обраними параметрами. Процеси запускаються за допомогою цієї кнопки, але спочатку потрібно встановити параметри процесу.

- stopButton (Стоп)

Кнопка "Стоп" – зупиняє обраний процес з можливістю його подальшого перезапуску.

- resumeButton (Відновити)

Кнопка "Відновити" – перезапускає зупинений процес.

- doneButton (Готово)

Кнопка «Готово» – зупиняє обраний процес без можливості його завершення.

2. Два випадаючих списки:

- processComboBox (Випадкове число, Числа Фібоначчі, Рекурсивний синус)

Панель «Процес» дозволяє вибрати процес, який хочемо розпочати. Він має три точки: числа Фібоначчі, випадкові числа та рекурсивний синус.

- priorityComboBox (Низький, Нижче середнього, Середній, Вище середнього, Високий)

Область «Пріоритет» дозволяє встановити пріоритет процесу для початку. Має 5 пунктів: низькій, нижче середнього, середній, вище середнього, високий.

3. Таблицю DataGridView з п'ятьма стовпцями:

- Номер процесу (Columns1)

Номер процесу – показує порядковий номер процесу.

- Тип процесу (Column2)

Тип процесу – показує дію яку цей процес виконує.

- Статус процесу (Column3)

Статус процесу – показує, запущено процес чи ні. Також показує, чи зупинено, завершено або просто відновлено процес.

- Пріоритет процесу (Column4)

Пріоритет процесу – показує пріоритет процесу, який був призначений йому при його створенні.

- Відсоток процесорного часу (Column5)

Відсоток процесорного часу – відображає відсоток використаного процесорного часу.

4. TextBox для виводу тексту (outputTextBox).

Таблиця в центрі вікна відображає стан кожного з запущених процесів. Після додавання кожного процесу до нього автоматично додається один рядок.

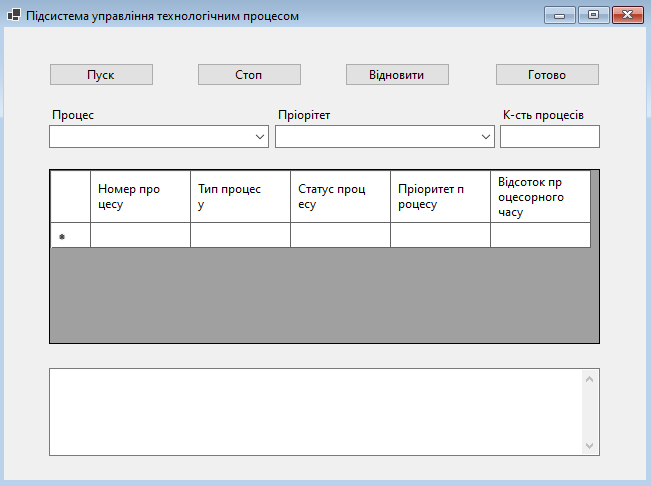
5. Три надписи (Labels):

- label1: "Процес"

- label2: "Пріорітет"

- label3: "К-сть процесів"

6. TextBox для введення кількості процесів (numProc).

  
Рис. 1 – Зовнішній вигляд додатку в конструкторі

Основою програми є абстрактний клас **Betaproc**. Він реалізує абстрактну модель процесу, без конкретизації дій, які потрібно виконати, з набором змінних і методів, загальних для всіх процесів.

Лістинг 1

Абстрактний клас Betaproc

public abstract class Betaproc

{

public abstract string ProcessType();

public abstract void Base();

}

Три процеси **sinProc**, **FibbonProc** і **ProcRandom**, походять від цього класу, і кожен реалізує тільки методи, які повертають тип процесу і сам метод, виконаний процесом.

Випадкове число – процес генерує випадкове число в діапазоні від одного до ста і відображає його у вихідному полі.

Лістинг 2

Процес ProcRandom

public class ProcRandom : Betaproc

{

private Random random = new Random();

public override string ProcessType()

{

return "Random Number";

}

public override void Base()

{

int number = random.Next(1, 101);

File.AppendAllText(filePath, $"Випадкове число: {number}\r\n");

}

}

Числа Фібоначчі – генерує послідовність Фібоначчі, починаючи з першого члена, і відображає їх у вихідному полі. Як тільки значення порядкових номерів перевищить тисячу, значення скидаються до перших доданків.

Лістинг 3

Процес FibbonProc

public class FibbonProc : Betaproc

{

private int a = 0;

private int b = 1;

public override string ProcessType()

{

return "Fibonacci Numbers";

}

public override void Base()

{

int temp = a;

if (a > 1000 || b > 1000)

{

a = 1;

b = 1;

temp = 0;

}

a = b;

b = temp + b;

File.AppendAllText(filePath, $"Фiбоначчi: {a}\r\n");

}

}

Рекурсивний синус – породжує значення синуса X. Спочатку X дорівнює 1, пізніше йому присвоюються обчислені значення sin(x). Значення, округлені до третього символу, відображаються в полі виводу.

Лістинг 4

Процес sinProc

public class sinProc : Betaproc

{

private double x = 1;

public override string ProcessType()

{

return "Recursive Sinus";

}

public override void Base()

{

x = Math.Sin(x);

File.AppendAllText(filePath, $"Рекурсивный синус: {x}\r\n");

}

}

При цьому кожен метод **Base**, який містить виконуваний код, має загальний дескриптор у своєму тілі, який дозволяє лише одному процесу виконувати код, коли інші стоять у черзі та отримують час виконання пропорційно пріоритету.

У момент запуску запускається таймер, який забезпечує однаковий квант часу в 3 секунди для кожного процесу.

Лістинг 5

Метод public ProcManager

private System.Timers.Timer timer;

public ProcManager(Form1 form)

{

this.form = form;

timer = new System.Timers.Timer();

timer.Interval = 3000;

timer.Elapsed += new ElapsedEventHandler(TimerElapsed!);

File.WriteAllText(filePath, "");

}

Однак процеси не висять в пам'яті комп'ютера просто так. Клас **ProcManager** був створений спеціально для управління процесами. Коли процес створений, він і вся інформація про нього вноситься в масив, і відповідно до номера осередку, в яку записаний процес, йому присвоюється ідентифікатор, за яким до нього можна отримати доступ в процесі роботи. Клас **ProcManager** також реалізує графічне представлення в пам'яті. Кожен важливий елемент процесу відображається в спеціальній таблиці на бланку, і при зміні одного з них викликається подія, яка змінює запис в таблиці в режимі реального часу, завдяки чому можна спостерігати, як мітка «працює» проходить від одного процесу до іншого.

Лістинг 6

Клас ProcManager

public class ProcManager

{

private List<Betaproc> processes = new List<Betaproc>();

private List<int> priorities = new List<int>();

private Dictionary<int, int> processExecutionCount = new Dictionary<int, int>();

private Mutex mutex = new Mutex();

private System.Timers.Timer timer;

private Form1 form;

private Random random = new Random();

public ProcManager(Form1 form)

{

this.form = form;

timer = new System.Timers.Timer();

timer.Interval = 3000;

timer.Elapsed += new ElapsedEventHandler(TimerElapsed!);

File.WriteAllText(filePath, "");

}

public void AddProcess(Betaproc process, int priority)

{

processes.Add(process);

priorities.Add(priority);

processExecutionCount[processes.Count - 1] = 0;

}

public void StartExecution()

{

timer.Start();

}

public void StopExecution()

{

timer.Stop();

}

private void TimerElapsed(object sender, ElapsedEventArgs e)

{

mutex.WaitOne();

if (processes.Count > 0)

{

int index = SelectProcessIndex();

processes[index].Base();

processExecutionCount[index]++;

UpdateProcessGrid(index);

}

mutex.ReleaseMutex();

form.outputTextBox.Invoke(new Action(() => form.outputTextBox.Text = File.ReadAllText(filePath)));

}

private int SelectProcessIndex()

{

int totalPriority = priorities.Sum();

int randomValue = random.Next(0, totalPriority);

int cumulativePriority = 0;

for (int i = 0; i < priorities.Count; i++)

{

cumulativePriority += priorities[i];

if (randomValue < cumulativePriority)

{

return i;

}

}

return 0;

}

private void UpdateProcessGrid(int index)

{

var row = form.processDataGridView.Rows[index];

row.Cells[2].Value = "Запущено";

row.Cells[4].Value = $"{(processExecutionCount[index] \* 100 / processExecutionCount.Values.Sum())}%";

}

public List<Betaproc> GetProcesses()

{

return processes;

}

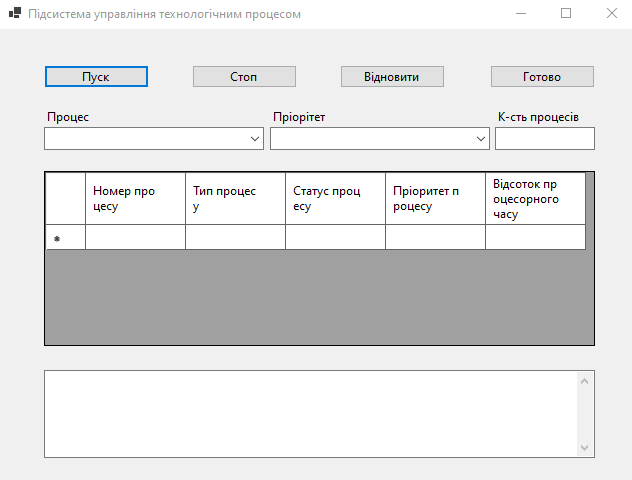
}

Пріоритет процесу показує ймовірність початку процесу, якщо попередній процес завершив свою роботу. Наприклад, якщо запустити три процеси з однаковим пріоритетом, кількість разів їх запуску буде приблизно однаковою, а якщо запустити два процеси з високим пріоритетом і один процес з низьким пріоритетом, то процес із низьким пріоритетом виконуватиметься приблизно один із шістнадцяти разів. Однак архітектура програми передбачає, що якщо кількість запущених процесів більше одиниці, то один і той же процес не може виконуватися більше одного разу поспіль.

Також у коді є передача даних між процесами, який виконується за допомогою запису даних у файл і читання цих даних з файлу. Коли кожен процес виконується, він записує свої результати у файл. Потім один з процесів, який відповідає за відображення результатів у вікні програми, зчитує цей файл і відображає дані у відповідному TextBox.

Кожен процес записує свої результати у файл за допомогою методу **File.AppendAllText(filePath, ...)** у методі **Base()** кожного класу процесу (**sinProc**, **FibbonProc**, **ProcRandom**).

Процес, який відповідає за відображення результатів, зчитує дані з файлу і відображає їх у TextBox у методі **TimerElapsed**.

****Рис. 2 –Зовнішній вигляд запущеного додатку

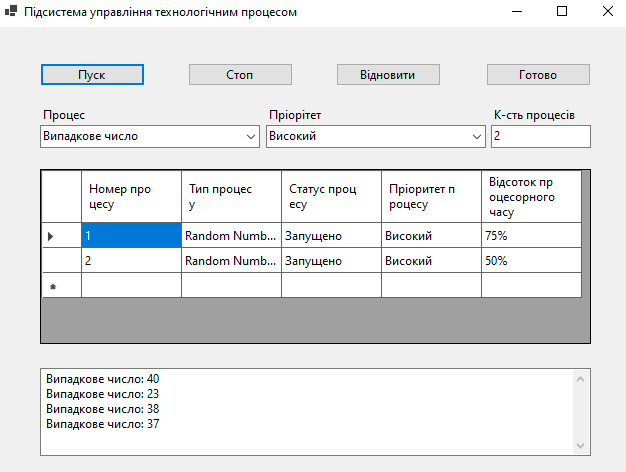
****

Рис. 3 – Запуск декількох процесів одного виду

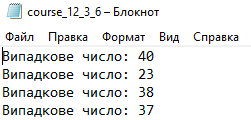
****

Рис. 4 – Відображення результату у файлі

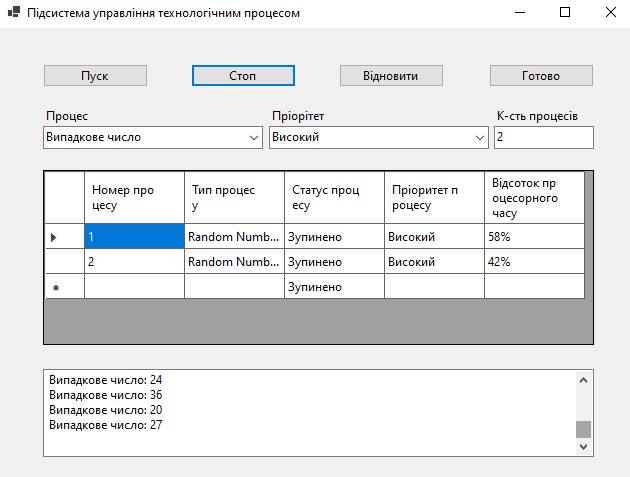


Рис. 5 – Зупинка роботи процесів

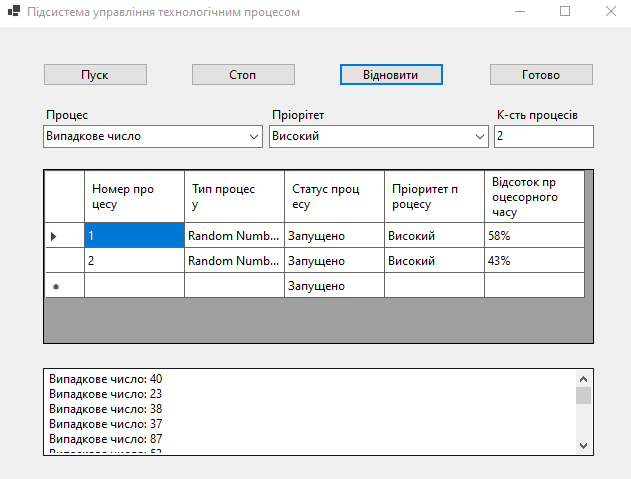


Рис. 6 – Відновлення роботи процесів

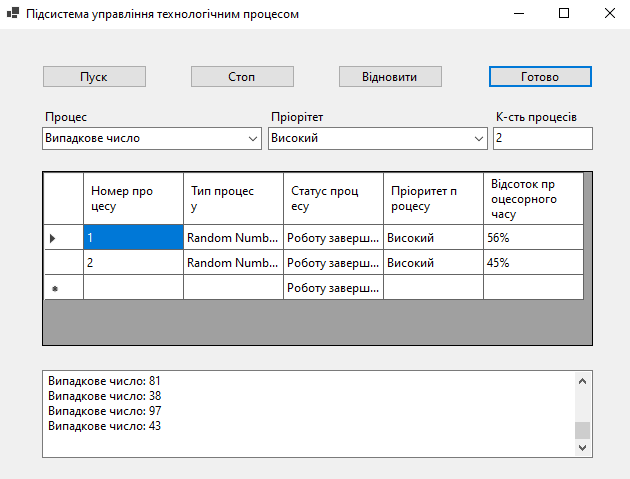


Рис. 7 – Завершення роботи процесів

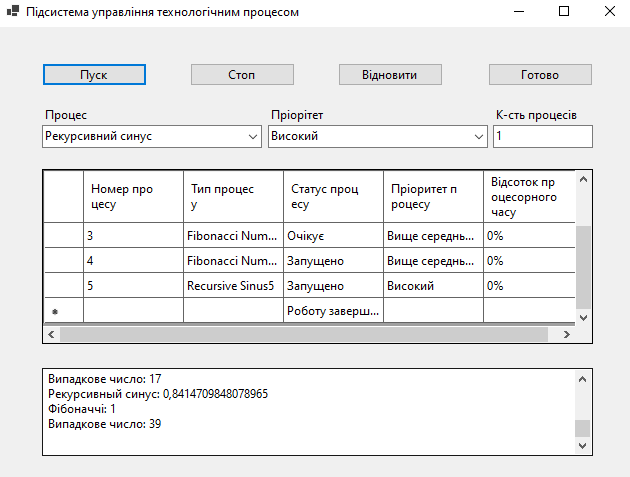


Рис. 8 – Запуск декількох процесів різного виду

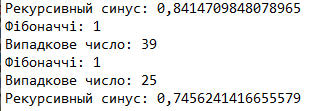


Рис. 9 – Відображення результату у файлі

# **ВИСНОВКИ**

Дана курсова робота реалізує систему управління технологічним процесом за допомогою мови програмування C# у середовищі Microsoft Visual Studio. Основна мета цієї системи полягає в ефективному розподілі процесорного часу між декількома одночасно існуючими процесами, забезпеченні їхньої взаємодії та наданні необхідних системних ресурсів.

У даній курсовій роботі було розроблено додаток для управління технологічним процесом, в якому було створено абстрактний класс Betaproc, який реалізує загальну модель процесу. Було створено три конкретних процеси SinProc, FibbonProc та ProcRandom, які успадковуються від цього класу і реалізують власні методи для виконання певних дій. Так клас ProcManager, який відповідає за управління створенням, виконанням та відображенням інформації про процеси. Графічний інтерфейс програми дозволяє вибирати тип та пріоритет процесу для запуску, а також має елементи управління для керування запущеними процесами.

Ця робота відповідає загальним вимоги до проекту, додаток був реалізован за допомогою технології Windows Forms, в ньому можна змінювати розмір вікна та продовжувати працювати без перебоїв. Програма використовує потоки та процеси для ефективного управління виконанням різних задач. Також застосовує об'єкт синхронізації для координації дій між процесами та передачі даних між ними через відображення файлів в пам'ять або канали.

У результаті було отримано наступні навички роботи:

1. Розуміння принципів управління технологічними процесами через програмне забезпечення.

2. Навички роботи з мовою програмування C# та середовищем Microsoft Visual Studio для розробки програмного забезпечення.

3. Здатність до ефективного використання потоків та процесів для розділення завдань та забезпечення паралельної обробки даних.

4. Розуміння принципів синхронізації процесів для уникнення конфліктів та забезпечення правильної взаємодії між ними.

5. Вміння розробляти графічний інтерфейс користувача для взаємодії з програмним забезпеченням.

# **СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. Матвєєва Н.О., Пономарьов І.В. Розробка багатопокових програм на

платформі .NET: навчальний посібник — Дніпро, 2022. — 169с.

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

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

4. Andrew Troelsen, Phillip Japikse. Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming. Tenth Edition - Apress, 2021. – 1353p.

5. Клири Стивен. Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование. 2-е межд. изд. — СПб.: Питер, 2020. — 272 с.

6. Rishabh Verma, Neha Shrivastava, Ravindra Akella. Parallel Programming with C# and .NET Core: Developing Multithreaded Applications Using C# and .NET Core 3.1 from Scratch (English Edition) – BPB PUBLICATIONS, 2020. – 394p.

7. What is .NET? Introduction and overview [Electronic resource] – Mode of access: URL: https://docs.microsoft.com/en-us/dotnet/core/introduction. – Last access: 2021. – Title from the screen.

**ДОДАТКИ**

ДОДАТОК А: ЛІСТИНГ Form1.cs

using System;

using System.CodeDom.Compiler;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Threading;

using System.Timers;

using System.Windows.Forms;

namespace курсовая\_3\_6

{

public partial class Form1 : Form

{

private ProcManager procManager;

private static string filePath = @"D:\course\_12\_3\_6.txt";

public abstract class Betaproc

{

public abstract string ProcessType();

public abstract void Base();

}

public class sinProc : Betaproc

{

private double x = 1;

public override string ProcessType()

{

return "Recursive Sinus";

}

public override void Base()

{

x = Math.Sin(x);

File.AppendAllText(filePath, $"Рекурсивный синус: {x}\r\n");

}

}

public class FibbonProc : Betaproc

{

private int a = 0;

private int b = 1;

public override string ProcessType()

{

return "Fibonacci Numbers";

}

public override void Base()

{

int temp = a;

if (a > 1000 || b > 1000)

{

a = 1;

b = 1;

temp = 0;

}

a = b;

b = temp + b;

File.AppendAllText(filePath, $"Фiбоначчi: {a}\r\n");

}

}

public class ProcRandom : Betaproc

{

private Random random = new Random();

public override string ProcessType()

{

return "Random Number";

}

public override void Base()

{

int number = random.Next(1, 101);

File.AppendAllText(filePath, $"Випадкове число: {number}\r\n");

}

}

public class ProcManager

{

private List<Betaproc> processes = new List<Betaproc>();

private List<int> priorities = new List<int>();

private Dictionary<int, int> processExecutionCount = new Dictionary<int, int>();

private Mutex mutex = new Mutex();

private System.Timers.Timer timer;

private Form1 form;

private Random random = new Random();

public ProcManager(Form1 form)

{

this.form = form;

timer = new System.Timers.Timer();

timer.Interval = 3000;

timer.Elapsed += new ElapsedEventHandler(TimerElapsed!);

File.WriteAllText(filePath, "");

}

public void AddProcess(Betaproc process, int priority)

{

processes.Add(process);

priorities.Add(priority);

processExecutionCount[processes.Count - 1] = 0;

}

public void StartExecution()

{

timer.Start();

}

public void StopExecution()

{

timer.Stop();

}

private void TimerElapsed(object sender, ElapsedEventArgs e)

{

mutex.WaitOne();

if (processes.Count > 0)

{

int index = SelectProcessIndex();

processes[index].Base();

processExecutionCount[index]++;

UpdateProcessGrid(index);

}

mutex.ReleaseMutex();

form.outputTextBox.Invoke(new Action(() => form.outputTextBox.Text = File.ReadAllText(filePath)));

}

private int SelectProcessIndex()

{

int totalPriority = priorities.Sum();

int randomValue = random.Next(0, totalPriority);

int cumulativePriority = 0;

for (int i = 0; i < priorities.Count; i++)

{

cumulativePriority += priorities[i];

if (randomValue < cumulativePriority)

{

return i;

}

}

return 0;

}

private void UpdateProcessGrid(int index)

{

var row = form.processDataGridView.Rows[index];

row.Cells[2].Value = "Запущено";

row.Cells[4].Value = $"{(processExecutionCount[index] \* 100 / processExecutionCount.Values.Sum())}%";

}

public List<Betaproc> GetProcesses()

{

return processes;

}

}

public Form1()

{

InitializeComponent();

procManager = new ProcManager(this);

outputTextBox.TextChanged += OutputTextBox\_TextChanged!;

}

private void OutputTextBox\_TextChanged(object sender, EventArgs e)

{

if (!outputTextBox.Focused)

{

outputTextBox.Text = File.ReadAllText(filePath);

}

}

private void startButton\_Click(object sender, EventArgs e)

{

if (processComboBox.SelectedItem == null)

{

MessageBox.Show("Будь ласка, оберіть тип процесу.");

return;

}

if (!int.TryParse(numProc.Text, out int numberOfProcesses) || numberOfProcesses <= 0)

{

MessageBox.Show("Введіть коректну кількість процесів.");

return;

}

for (int i = 0; i < numberOfProcesses; i++)

{

Betaproc process;

switch (processComboBox.SelectedItem.ToString())

{

case "Випадкове число":

process = new ProcRandom();

break;

case "Числа Фібоначчі":

process = new FibbonProc();

break;

case "Рекурсивний синус":

process = new sinProc();

break;

default:

throw new Exception("Невідомий тип процесу");

}

int priority = priorityComboBox.SelectedIndex + 1;

if (priority == 0)

{

MessageBox.Show("Будь ласка, оберіть пріоритет процесу.");

return;

}

procManager.AddProcess(process, priority);

int rowIndex = processDataGridView.Rows.Add();

DataGridViewRow row = processDataGridView.Rows[rowIndex];

row.Cells[0].Value = rowIndex + 1;

row.Cells[1].Value = $"{process.ProcessType()}{rowIndex + 1}";

row.Cells[2].Value = "Очікує";

row.Cells[3].Value = priorityComboBox.SelectedItem?.ToString() ?? "Невідомий";

row.Cells[4].Value = "0%";

procManager.StartExecution();

}

}

private void stopButton\_Click(object sender, EventArgs e)

{

procManager.StopExecution();

UpdateProcessStatus("Зупинено");

}

private void resumeButton\_Click(object sender, EventArgs e)

{

procManager.StartExecution();

UpdateProcessStatus("Запущено");

}

private void doneButton\_Click(object sender, EventArgs e)

{

procManager.StopExecution();

UpdateProcessStatus("Роботу завершено");

foreach (DataGridViewRow row in processDataGridView.SelectedRows)

{

processDataGridView.Rows.RemoveAt(row.Index);

}

}

private void UpdateProcessStatus(string status)

{

foreach (DataGridViewRow row in processDataGridView.Rows)

{

row.Cells[2].Value = status;

}

}

private void processDataGridView\_SelectionChanged(object sender, EventArgs e)

{

outputTextBox.Text = File.ReadAllText(filePath);

}

}

}

ДОДАТОК Б: ЛІСТИНГ Form1.Designer.cs

namespace курсовая\_3\_6

{

partial class Form1

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

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

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

startButton = new Button();

stopButton = new Button();

resumeButton = new Button();

doneButton = new Button();

processComboBox = new ComboBox();

priorityComboBox = new ComboBox();

processDataGridView = new DataGridView();

Columns1 = new DataGridViewTextBoxColumn();

Column2 = new DataGridViewTextBoxColumn();

Column3 = new DataGridViewTextBoxColumn();

Column4 = new DataGridViewTextBoxColumn();

Column5 = new DataGridViewTextBoxColumn();

outputTextBox = new TextBox();

label1 = new Label();

label2 = new Label();

numProc = new TextBox();

label3 = new Label();

((System.ComponentModel.ISupportInitialize)processDataGridView).BeginInit();

SuspendLayout();

//

// startButton

//

startButton.Location = new Point(45, 36);

startButton.Name = "startButton";

startButton.Size = new Size(105, 23);

startButton.TabIndex = 0;

startButton.Text = "Пуск";

startButton.UseVisualStyleBackColor = true;

startButton.Click += startButton\_Click;

//

// stopButton

//

stopButton.Location = new Point(193, 36);

stopButton.Name = "stopButton";

stopButton.Size = new Size(105, 23);

stopButton.TabIndex = 1;

stopButton.Text = "Стоп";

stopButton.UseVisualStyleBackColor = true;

stopButton.Click += stopButton\_Click;

//

// resumeButton

//

resumeButton.Location = new Point(341, 36);

resumeButton.Name = "resumeButton";

resumeButton.Size = new Size(105, 23);

resumeButton.TabIndex = 2;

resumeButton.Text = "Відновити";

resumeButton.UseVisualStyleBackColor = true;

resumeButton.Click += resumeButton\_Click;

//

// doneButton

//

doneButton.Location = new Point(491, 36);

doneButton.Name = "doneButton";

doneButton.Size = new Size(105, 23);

doneButton.TabIndex = 3;

doneButton.Text = "Готово";

doneButton.UseVisualStyleBackColor = true;

doneButton.Click += doneButton\_Click;

//

// processComboBox

//

processComboBox.ForeColor = SystemColors.WindowText;

processComboBox.FormattingEnabled = true;

processComboBox.Items.AddRange(new object[] { "Випадкове число", "Числа Фібоначчі", "Рекурсивний синус" });

processComboBox.Location = new Point(45, 98);

processComboBox.Name = "processComboBox";

processComboBox.Size = new Size(220, 23);

processComboBox.TabIndex = 4;

//

// priorityComboBox

//

priorityComboBox.ForeColor = SystemColors.WindowText;

priorityComboBox.FormattingEnabled = true;

priorityComboBox.Items.AddRange(new object[] { "Низький", "Нижче середнього", "Середній", "Вище середнього", "Високий" });

priorityComboBox.Location = new Point(271, 98);

priorityComboBox.Name = "priorityComboBox";

priorityComboBox.Size = new Size(220, 23);

priorityComboBox.TabIndex = 5;

priorityComboBox.Tag = "";

//

// processDataGridView

//

processDataGridView.ColumnHeadersHeightSizeMode = DataGridViewColumnHeadersHeightSizeMode.AutoSize;

processDataGridView.Columns.AddRange(new DataGridViewColumn[] { Columns1, Column2, Column3, Column4, Column5 });

processDataGridView.Location = new Point(45, 142);

processDataGridView.Name = "processDataGridView";

processDataGridView.Size = new Size(551, 175);

processDataGridView.TabIndex = 6;

processDataGridView.SelectionChanged += processDataGridView\_SelectionChanged;

//

// Columns1

//

Columns1.HeaderText = "Номер процесу";

Columns1.Name = "Columns1";

//

// Column2

//

Column2.HeaderText = "Тип процесу";

Column2.Name = "Column2";

//

// Column3

//

Column3.HeaderText = "Статус процесу";

Column3.Name = "Column3";

//

// Column4

//

Column4.HeaderText = "Пріоритет процесу";

Column4.Name = "Column4";

//

// Column5

//

Column5.HeaderText = "Відсоток процесорного часу";

Column5.Name = "Column5";

//

// outputTextBox

//

outputTextBox.Location = new Point(45, 341);

outputTextBox.Multiline = true;

outputTextBox.Name = "outputTextBox";

outputTextBox.ScrollBars = ScrollBars.Vertical;

outputTextBox.Size = new Size(551, 88);

outputTextBox.TabIndex = 7;

//

// label1

//

label1.AutoSize = true;

label1.Location = new Point(45, 80);

label1.Name = "label1";

label1.Size = new Size(49, 15);

label1.TabIndex = 8;

label1.Text = "Процес";

//

// label2

//

label2.AutoSize = true;

label2.Location = new Point(271, 80);

label2.Name = "label2";

label2.Size = new Size(59, 15);

label2.TabIndex = 9;

label2.Text = "Пріорітет";

//

// numProc

//

numProc.Location = new Point(496, 98);

numProc.Name = "numProc";

numProc.Size = new Size(100, 23);

numProc.TabIndex = 10;

//

// label3

//

label3.AutoSize = true;

label3.Location = new Point(496, 80);

label3.Name = "label3";

label3.Size = new Size(88, 15);

label3.TabIndex = 11;

label3.Text = "К-сть процесів";

//

// Form1

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

ClientSize = new Size(636, 453);

Controls.Add(label3);

Controls.Add(numProc);

Controls.Add(label2);

Controls.Add(label1);

Controls.Add(outputTextBox);

Controls.Add(processDataGridView);

Controls.Add(priorityComboBox);

Controls.Add(processComboBox);

Controls.Add(doneButton);

Controls.Add(resumeButton);

Controls.Add(stopButton);

Controls.Add(startButton);

Name = "Form1";

Text = "Підсистема управління технологічним процесом";

((System.ComponentModel.ISupportInitialize)processDataGridView).EndInit();

ResumeLayout(false);

PerformLayout();

}

#endregion

private Button startButton;

private Button stopButton;

private Button resumeButton;

private Button doneButton;

private DataGridViewTextBoxColumn Columns1;

private DataGridViewTextBoxColumn Column2;

private DataGridViewTextBoxColumn Column3;

private DataGridViewTextBoxColumn Column4;

private DataGridViewTextBoxColumn Column5;

public ComboBox processComboBox;

public ComboBox priorityComboBox;

public DataGridView processDataGridView;

public TextBox outputTextBox;

private Label label1;

private Label label2;

private TextBox numProc;

private Label label3;

}

}