1.2. Порівняння потоків з процесами

Потоки відрізняються від традиційних процесів багатозадачних операційних систем, в тому що процеси:

- на загал, незалежні;
- дублюють значну частину інформації про стан;
- мають окремий адресний простір;
- взаємодіють тільки через системні міжпроцесорні механізми комунікацій.

Потоки всередині процесу, з іншої сторони, розподіляють інформацію про стан процесу, і прямо доступаються до спільної пам'яті та інших ресурсів. Переключення контексту між потоками процесу на загал швидше, ніж переключення контексту між процесами. Описуючи ситуацію такі системи, як Windows NT та OS/2, кажуть, що мають «дешеві» потоки та «дорогі» процеси; в інших операційних системах ситуація не дуже відмінна.

1.3. Багатопотоковість

Багатопотоковість[2] (англ. *multi-threading*), — властивість операційної системи або застосунку, яка полягає в тому, що процес, породжений в операційній системі, може складатися з кількох нитей, що виконуються паралельно, або навіть одночасно на багатопроцесорних системах. При виконанні деяких завдань таке розділення може досягти ефективнішого використання ресурсів комп'ютера.

Головною метою багатопотоковості є квазі-багатозадачність на рівні одного виконуваного процесу, тобто всі потоки виконуються в адресному просторі процесу. Окрім цього, всі потоки процесу мають не тільки спільний адресний простір, але і спільні дескриптори файлів. Процес, що виконується, має як мінімум однин (головний) потік.

Головні переваги в багатопотоковості:

- Спрощення програми в деяких випадках, за рахунок використання загального адресного простору;
- Менші відносно процесу часові витрати на створення ниті і взаємодію між нитями;
- Підвищення продуктивності процесу за рахунок розпаралелювання процесорних обчислень і операцій вводу/виводу.

1.4. Взаємодія потоків

У багатонитевому середовищі часто виникають проблеми, зв'язані з використанням паралельними виконуваними птоками одних і тих же даних або пристроїв. Для вирішення подібних проблем використовуються такі методи взаємодії потоків, як взаємовиключення (м'ютекси), семафори, критичні секції і події.

- Взаємовиключення (mutex, м'ютекс) — це об'єкт синхронізації, який встановлюється в особливий сигнальний стан, коли не зайнятий жодним потоком. Тільки один поток володіє цим об'єктом у будь-який момент часу, звідси і назва таких об'єктів (від англійського mutually exclusive access — взаємно виключний доступ) — одночасний доступ до загального ресурсу виключається. Після всіх необхідних дій м'ютекс звільняється потоком, надаючи іншим потокам доступ до загального ресурсу.
- Семафори - є доступні ресурси, які можуть займатися кількома потоками в один і той же час, поки обсяг ресурсів не спустіє. Тоді додаткові потоки повинні чекати, поки необхідна кількість ресурсів не буде знову доступна. Семафори дуже ефективні, оскільки вони дозволяють одночасний доступ до ресурсів.
- Події - корисні в тих випадках, коли необхідно послати повідомлення потоку, що відбулося певна подія. Наприклад, при асинхронних операціях вводу/виводу з одного пристрою, система встановлює подію в сигнальний стан, коли закінчується якась з цих операцій. Один поток може використовувати кілька

різних подій в декількох операціях, що перекриваються, а потім чекати приходу сигналу від будь-якого з них.

- Критичні секції забезпечують синхронізацію подібно м'ютексам за винятком того, що об'єкти, що представляють критичні секції, доступні в межах одного процесу.

Події, м'ютекси і семафори також можна використовувати в однопроцесному застосунку, проте критичні секції забезпечують швидший і ефективніший механізм синхронізації взаємного виключення.

1.5. Засоби роботи з потоками в мові C#

C # підтримує паралельне виконання коду через багатопоточність.

Програма на C# запускається як єдиний потік, автоматично створюваний CLR(Common Language Runtime - загальномовне виконуюче середовище) і операційною системою ("головний" потік), і стає багатопотоковою за допомогою створення додаткових потоків. Ось простий приклад і його висновок:

```
class ThreadTest
{
   static void Main()
    {
        Thread thread = new Thread(Print);
        thread.Start();       // Виконати Print() у новому потоці
        while (true)
        {
                Console.Write("Hi!"); // Друкувати в косоль «Hi!»
        }
     }

   static void Print()
   {
        while (true)
        {
                Console.Write("Hello, world!");
                // Друкувати в косоль «Hello, world!»
        }
```

```
        }
    }
```

У головному потоці створюється новий потік thread, виконуючий метод, який безперервно друкує «Hi!». Одночасно головний потік безперервно друкує «Hello, world!». CLR призначає кожному потоку свій стек, так що локальні змінні зберігаються окремо.

Управління багатопоточністю здійснює планувальник потоків, цю функцію CLR зазвичай делегує операційній системі[4]. Планувальник потоків гарантує, що активним потокам виділяється відповідний час на виконання, а потоки, які очікують або блоковані, наприклад, на очікуванні ексклюзивної блокування, або користувача введення - не споживають часу CPU.

На однопроцесорних комп'ютерах планувальник потоків використовує квантування часу - швидке перемикання між виконанням кожного з активних потоків. Це призводить до непередбачуваного поведінки, як в першому прикладі, де кожна послідовність «Hi!» і «Hello, world!» відповідає кванту часу, виділеного потоку. У

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

Кажуть, що потік витісняється, коли його виконання призупиняється через зовнішніх факторів типу квантування часу. У більшості випадків потік не може контролювати, коли і де він буде витіснений.

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

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

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

```
class ThreadTest

{
    static void Main(){
        Thread t = new Thread(new ThreadStart(Run));
        t.Start();   // Виконати Run() в новому потоці
        Run ();        // Одночасно запустити Run() у головному потоці
}
static void Run ()
{
    Console.WriteLine("Hello, world!");
}
```

У цьому прикладі потік виконує метод Run () одночасно з головним потоком. Результат - два майже одночасних «Hello, world!»

Потік має властивість IsAlive, що повертає true після виклику Start () і до завершення потоку.

Потік, який закінчив виконання, не може бути початий знову.

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

Головному потоку також можна призначити ім'я - в наступному прикладі доступ до головного потоку здійснюється через статичну властивість CurrentThread класу Thread:

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

```
        worker.Name = "worker";
        worker.Start();
        Go();
    }

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

Властивість Priority визначає, скільки часу на виконання буде виділено потоку щодо інших потоків того ж процесу. Існує 5 градацій пріоритету потоку:

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

Значення пріоритету стає істотним, коли одночасно виконуються декілька потоків.

Установка пріоритету потоку на максимум ще не означає роботу в реальному часі (real-time), так як існують ще пріоритет процесу додатки. Щоб працювати в реальному часі, потрібно використовувати клас Process з простору імен System.Diagnostics для підняття пріоритету процесу:

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

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

Якщо real-time програма має користувальницький інтерфейс, може бути не бажано піднімати пріоритет його процесу, так як оновлення екрану буде з'їдати занадто багато часу CPU - гальмуючи весь комп'ютер, особливо якщо UI досить складний. Зменшення пріоритету головного потоку в поєднанні з підвищенням

пріоритету процесу гарантує, що real-time потік не буде витіснятися перемальовуванням екрану, але не рятує від гальм весь комп'ютер, тому що операційна система все ще буде виділяти багато часу CPU всьому процесу в цілому. Ідеальне рішення полягає в тому, щоб тримати роботу в реальному часі і користувальницький інтерфейс в різних процесах (з різними пріоритетами), що підтримують зв'язок через Remoting або shared memory.

1.6. Висновки до розділу 1

У даному розділі розглядалася багатопотоковість в мові програмування C. Був виконаний аналіз багатопотоковості в даній мові програмування. Короткий огляд даної мови багатопотокового програмування можна виконати за такими основними параметрами:

1. Створення потоку. Первинний (головний) потік створюється автоматично , решта - за допомогою класу Thread, якому в якості параметрів передається делегат типу ThreadStart, який вказує метод, який потрібно виконати. Виклик методу Start починає виконання потоку.

2. Засоби синхронізації потоків. У C # для синхронізації використовуються: lock, Mutex, Semaphore, EventWaitHandle, Wait and Pulse, Interlocked, volatile (для безпечного не блокує доступу до полів).

В C# існує стандартний набір функцій для повноцінної роботи багатопоточних систем. При виборі конкретного засобу програмування потрібно керуватися специфічними завданнями, які необхідно вирішити в даній задачі.