**Синхронізації потоків на основі повідомлень**

Ціль роботи

– навчитись виконувати паралельні обчислення з використанням декількох потоків та засобів синхронізації на основі повідомлень.

Теоретичні відомості

### Стан потоку

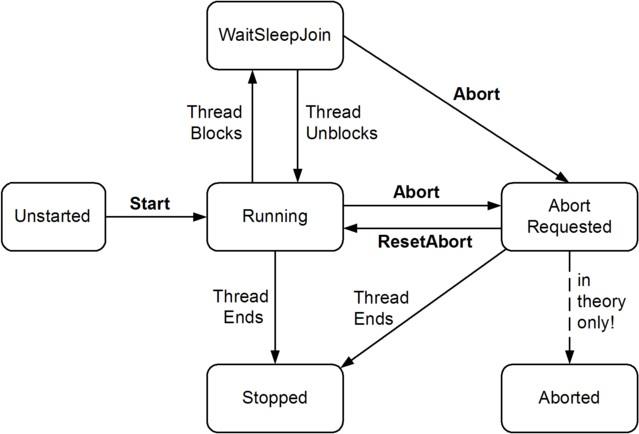


Рис.3.1. Граф станів потоку

Отримати інформацію про стан потоку можна за допомогою його властивості ThreadState. Рис. 3.1 демонструє «рівні» переліку (перерахування) ThreadState.

ThreadState спроектований не досить зручно - це комбінація трьох рівнів станів з використанням бітових прапорців:

* running/blocking/aborting (див. рис. 3.1);
* Background/foreground (ThreadState.Background);
* призупинення з використанням нерекомендованого методу
* Suspend (ThreadState.SuspendRequested і ThreadState.Suspended).

Значення перерахування в межах кожного рівня є взаємовиключними.

У результаті ThreadState – бітова комбінація з членів кожного рівня:

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

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

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

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

і замість цього потрібно перевіряти наявність ThreadState.Running шляхом виключення або в якості альтернативи, використовувати властивість IsAlive.

Властивість IsAlive, однак, може вам не підійти, так як повертає true, якщо потік блокований або припинений (false повертається тільки до того, як потік почався, і після того, як він завершиться).

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

### Wait Handles

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

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

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

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

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

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

### AutoResetEvent

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

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

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

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

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

EventWaitHandle wh = new AutoResetEvent(false);

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

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

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

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

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

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

class BasicWaitHandle {

static EventWaitHandle wh = new AutoResetEvent(false);

static void Main() {

new Thread(Waiter).Start();

Thread.Sleep(1000); // почекати ...

wh.Set();

}

static void Waiter() {

Console.WriteLine("Очікування...");

wh.WaitOne(); // очікування сигналу

Console.WriteLine("Отримали сигнал");

}

}

### Створення міжпроцесних EventWaitHandle

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

EventWaitHandle wh =

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

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

### Сценарій «Отримай і виконуй»

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

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

class AcknowledgedWaitHandle {

static EventWaitHandle ready = new AutoResetEvent(false);

static EventWaitHandle go = new AutoResetEvent(false);

static volatile string task;

static void Main() {

new Thread(Work).Start();

// Сигналізуємо робочому потоку 5 разів

for (int i = 1; i <= 5; i++) {

ready.WaitOne(); // Чекаємо на готовність робочого потоку

task = "a".PadRight(i, ‘h’); // Призначаємо задачу

go.Set(); // Повідомляємо робочому потоку, що можна починати

}

// Повідомляємо про необхідність завершення робочого потоку

// використовуючи null-значення

ready.WaitOne();

task = null;

go.Set();

}

static void Work() {

while (true) {

ready.Set(); // повідомляємо про готовність

go.WaitOne(); // Очікуємо сигнал почати

if (task == null)

return; // Завершуємо

Console.WriteLine(task);

}

}

}

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

### Черга «Постачальник/Споживач (Producer/Consumer)»

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

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

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

using System;

using System.Threading;

using System.Collections.Generic;

class ProducerConsumerQueue : IDisposable {

EventWaitHandle wh = new AutoResetEvent(false);

Thread worker;

object locker = new object();

Queue<string> tasks = new Queue<string>();

public ProducerConsumerQueue(). {

worker = new Thread(Work);

worker.Start();

}

public void EnqueueTask(string task) {

lock (locker)

tasks.Enqueue(task);

wh.Set();

}

public void Dispose() {

EnqueueTask(null); // Сигнал Споживачу про завершення

worker.Join(); // Очікування завершення

wh.Close(); // Звільнення ресурсів

}

void Work() {

while (true) {

string task = null;

lock (locker) {

if (tasks.Count > 0) {

task = tasks.Dequeue();

if (task == null)

return;

}

}

if (task != null) {

Console.WriteLine("Виконується задача: " + task);

Thread.Sleep(1000); // симуляція роботи...

}

else

wh.WaitOne(); // Більше задач немає, очікуємо сигнал...

}

}

}

Код тестування черги:

class Test {

static void Main() {

using(ProducerConsumerQueue q = new ProducerConsumerQueue()) {

q.EnqueueTask("Привіт!");

for (int i = 0; i < 10; i++)

q.EnqueueTask("Повідомлення " + i);

q.EnqueueTask("Bye!");

}

// Вихід з using приводить до виклику Dispose, який ставить в чергу

// null-задачу і очікує поки Споживач не завершить своє виконання

}

}

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

### ManualResetEvent

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

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

### WaitAny, WaitAll і SignalAndWait

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

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

WaitHandle.SignalAndWait(wh1, wh2);

в той час як другий потік - навпаки:

WaitHandle.SignalAndWait(wh2, wh1);

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

Робоче завдання

1. Реалізувати індивідуальне завдання.
2. Дослідити алгоритм виконання програми на мові С#.
3. Задокументувати у звіті по виконаній роботі висновки за результатами досліджень і підготувати відповіді на контрольні питання.

Методичні вказівки

*Завдання № 1*

Написати паралельну програму, яка обчислює значення виразу

*F* = *x*1*x*2 + (*x*3 + *x*1) *x*4.

Кожна арифметична операція має бути виконана в окремому потоці. Змінні ініціалізуються в тому ж потоці, в якому вони вперше використовуються. Ініціалізувати змінні наступними значеннями: *x*1 = 10, *x*2 = 20, *x*3 = 30, *x*4 = 40.

*Розв’язання завдання 1*

using System;

using System.Threading;

namespace Lab3\_1 {

class Program {

static EventWaitHandle wh1 = new AutoResetEvent(false),

wh2 = new AutoResetEvent(false),

wh3 = new AutoResetEvent(false);

static private int x1, x2, x3, x4;

static int A, B;

static void Main(string[] args) {

var T0 = new Thread(Func0);

var T1 = new Thread(Func1);

var T2 = new Thread(Func2);

var T3 = new Thread(Func3);

T0.Start();

T1.Start();

T2.Start();

T3.Start();

T3.Join();

Console.ReadKey();

}

static void Func0() {

x1 = 10;

x2 = 20;

A = x1\*x2;

wh1.Set();

}

static void Func1() {

x3 = 30;

B = x3 + x1;

wh2.Set();

}

static void Func2() {

x4 = 40;

wh2.WaitOne();

B \*= x4;

wh3.Set();

}

static void Func3() {

wh1.WaitOne();

wh3.WaitOne();

Console.WriteLine("F = {0}", A + B);

}

}

}

Індивідуальні завдання.

Необхідно розв’язати два завдання. У кожному із завдань обрати свій варіант.

*Завдання 1.*

Написати паралельну програму, яка обчислює значення виразу F за варіантом. Кожна арифметична операція має бути виконана в окремому потоці. Змінні ініціалізуються в тому ж потоці, в якому вони вперше використовуються. Ініціалізувати змінні наступними значеннями: *x*1 = 1, *x*2 = 2, *x*3 = 3, *x*4 = 4, *x*5 = 5, *x*6 = 6.

| № варіанта | Вираз F |
| --- | --- |
| 1 | *x*1 *x*2 + *x*3 + *x*4 *x*5 + *x*6 |
| 2 | *x*1+ *x*2 *x*3 + *x*4 + *x*5 + *x*6 |
| 3 | *x*1 *x*2 + *x*3*x*4 + *x*5 + *x*6 |
| 4 | *x*1+ *x*2 + *x*3 + (*x*4 + *x*5)\*x6 |
| 5 | (*x*1 + *x*2)(*x*3 + *x*4) + *x*5*x*6 |
| 6 | (*x*1+ *x*2 + *x*3)(*x*4 + *x*5 + *x*6) |
| 7 | (*x*1+ *x*2)(*x*3 + *x*4) + *x*5 + *x*6 |
| 8 | (*x*1+ *x*2)(*x*3 + *x*4)*x*5 + *x*6 |
| 9 | (*x*1+ *x*2)(*x*3 + *x*4)(*x*5 + *x*6) |
| 10 | (*x*1+ *x*2)(*x*3 + *x*4 + *x*5 + *x*6) |
| 11 | (*x*1 *x*2+ *x*3)*x*4 + *x*5 + *x*6 |
| 12 | (*x*1+ *x*2) + *x*3(*x*4+ *x*5 + *x*6) |
| 13 | *x*1*x*2 + (*x*3 + *x*4 + *x*5) *x*6 |
| 14 | (*x*1+ *x*2 *x*3 + *x*4 + *x*5) *x*6 |
| 15 | *x*1(*x*2 + *x*3 + *x*4 + *x*5) *x*6 |
| 16 | (*x*1 *x*2+ *x*3)( *x*4 + *x*5 + *x*6) |
| 17 | *x*1+(( *x*2 + *x*3)*x*4+ *x*5) *x*6 |
| 18 | *x*1 *x*2 *x*3 + *x*4 *x*5x6 |
| 19 | *x*1 *x*2 + ( *x*3 + *x*4 + *x*5)*x*6 |
| 20 | (*x*1 *x*2 + *x*3)(*x*4 + *x*5) *x*6 |
| 21 | (*x*1+ *x*2) *x*3 *x*4 + *x*5 *x*6 |
| 22 | (*x*1+ *x*2 + *x*3)*x*4(*x*5 + *x*6) |
| 23 | (*x*1 *x*2 +( *x*3 + *x*4) + *x*5) *x*6 |
| 24 | *x*1+ ( *x*2 + *x*3) + (*x*4 + *x*5) *x*6 |
| 25 | (*x*1+ *x*2) + (*x*3*x*4+ *x*5*x*6) |
| 26 | *x*1 *x*2 + ( *x*3 + *x*4 + *x*5)*x*6 |
| 27 | (*x*1+ *x*2 *x*3 + *x*4 + *x*5)*x*6 |
| 28 | *x*1( *x*2 + *x*3 + *x*4+ *x*5)*x*6 |
| 29 | (*x*1 *x*2 + *x*3)( *x*4 + *x*5+ *x*6) |
| 30 | *x*1 + (( *x*2 + *x*3) *x*4 + *x*5)*x*6 |

*Завдання 2.*

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

1. Кількість робочих потоків P, якими виконують паралельні обчислення, має задаватися користувачем;
2. Потрібно побудувати графік залежності часу виконання програми від розмірності матриць (N = 103, 104, 105 …) при однопоточному режимі та кількості потоків, яка відповідає кількості логічних ядер персонального комп’ютера.

| № варіанта | Вираз, для обчислення |
| --- | --- |
| 1 | (M1+M2)\*(M3+M4)+M5\*M6 |
| 2 | (M1+M2+M3)\*(M4+M5+M6) |
| 3 | (M1+M2)\*(M3+M4)+M5+M6 |
| 4 | (M1+M2)\*(M3+M4)\*M5+M6 |
| 5 | (M1+M2)\*(M3+M4)\*(M5+M6) |
| 6 | (M1+M2)\*(M3+M4+M5+M6) |
| 7 | (M1\*M2+M3)\*M4+M5+M6 |
| 8 | (M1+M2)+M3\*(M4+M5+M6) |
| 9 | M1\*M2+(M3+M4+M5)\*M6 |
| 10 | (M1+M2\*M3+M4+M5)\*M6 |
| 11 | M1\*(M2+M3+M4+M5)\*M6 |
| 12 | (M1\*M2+M3)\*(M4+M5+M6) |
| 13 | M1+((M2+M3)\*M4+M5)\*M6 |
| 14 | M1\*M2\*M3+M4\*M5\*M6 |
| 15 | M1\*M2+(M3+M4+M5)\*M6 |
| 16 | (M1\*M2+M3)\*(M4+M5)\*M6 |
| 17 | (M1+M2)\*M3\*M4+M5\*M6 |
| 18 | (M1+M2+M3)\*M4\*(M5+M6) |
| 19 | (M1\*M2+(M3+M4)+M5)\*M6 |
| 20 | M1+(M2+M3)+(M4+M5)\*M6 |
| 21 | (M1+M2)+(M3\*M4+M5\*M6) |
| 22 | M1\*M2+(M3+M4+M5)\*M6 |
| 23 | (M1+M2\*M3+M4+M5)\*M6 |
| 24 | M1\*(M2+M3+M4+M5)\*M6 |
| 25 | (M1\*M2+M3)\*(M4+M5+M6) |
| 26 | M1+((M2+M3)\*M4+M5)\*M6 |
| 27 | M1\*M2+M3+M4+M5\*M6 |
| 28 | M1+M2\*M3+M4+M5+M6 |
| 29 | M1\*M2+M3\*M4+M5+M6 |
| 30 | M1+M2+M3+(M4+M5)\*M6 |

Контрольні питання

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