**Засоби синхронізації потоків в C#**

Ціль роботи

– організація синхронізованої роботи потоків засобами С#. Навчитись виконати паралельні обчислення з використанням декількох потоків та засобів синхронізації (семафорів, м’ютексів, моніторів).

Теоретичні відомості

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

Таблиця 2.1 – Найпростіші методи блокування

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

Таблиця 2.2 – Конструкції блокування

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

Таблиця 2.3 – Сигнальні конструкції

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

Таблиця 2.4 – Конструкції синхронізації без блокування

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

### Блокування

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

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

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

### Sleeping або Spinning

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

static void Main() {

Thread.Sleep(0); // відмовитися від одного кванта часу CPU

Thread.Sleep(1000); // заснути на 1000 мілісекунд Thread.Sleep(TimeSpan.FromHours(1)); // заснути на годину Thread.Sleep(Timeout.Infinite); // заснути до переривання

}

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

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

Клас Thread також надає метод SpinWait, який не відмовляється від часу CPU, а навпаки, завантажує процесор в циклі на задану кількість ітерацій. 50 ітерацій еквівалентні паузі приблизно в мікросекунду, хоча це залежить від швидкості і завантаження CPU. Технічно SpinWait – неблокуючий метод: ThreadState такого потоку не встановлюється в WaitSleepJoin і потік не може бути перерваний з іншого потоку.

SpinWait рідко використовується, його головне застосування це очікування ресурсу, який повинен звільниться дуже скоро (протягом декількох мікросекунд) без виклику Sleep і витрат процесорного часу на перемикання потоку. Однак ця методика вигідна тільки на багатопроцесорних комп’ютерах, на однопроцесорному комп’ютері у ресурсу немає ніякого шансу звільнитися. Часті або тривалі виклики SpinWait даремно витрачують час CPU.

### Блокування проти очікування у циклі

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

while (!proceed) {

…

};

або

while (DateTime.Now < nextStartTime) {

…

};

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

Іноді використовується гібрид блокування та очікування у циклі:

while (!proceed)

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

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

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

### Очікування завершення потоку

Потік можна заблокувати до завершення іншого потоку викликом методу Join:

class JoinDemo {

static void Main() {

Thread t = new Thread(delegate() {

Console.ReadLine();

});

t.Start();

t.Join();

Console.WriteLine("Thread t’s ReadLine complete!");

}

}

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

Thread.Sleep(1000);

Thread.CurrentThread.Join(1000);

### Блокування і потокова безпека

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

class ThreadUnsafe {

static int val1, val2;

static void Go() {

if (val2 != 0)

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

}

}

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

Ось як за допомогою блокування можна вирішити цю проблему:

class ThreadSafe {

static object locker = new object(); static int val1, val2;

static void Go() {

lock (locker) {

if (val2 != 0)

Console.WriteLine(val1 / val2);

val2 = 0;

}

}

}

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

Потік, заблокований на час очікування звільнення блокування, має властивість ThreadState, встановлену в WaitSleepJoin. Пізніше буде показано, як потік, заблокований в такому стані, може бути примусово звільнений з іншого потоку викликом методів Interrupt або Abort. Це досить потужна можливість, що використовується зазвичай для завершення робочого потоку.

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

Monitor.Enter(locker);

try {

if (val2 != 0)

Console.WriteLine(val1 / val2);

val2 = 0;

}

finally { Monitor.Exit(locker); }

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

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

### Вибір об’єкта синхронізації

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

class ThreadSafe() {

List <string> list = new List <string>();

void Test() {

lock (list) {

list.Add("Item 1");

...

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

lock (this){

...

}

або

lock (typeof(Widget))

...

}

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

### Вкладені блокування

Потік може неодноразово блокувати один і той же об’єкт багаторазовими викликами Monitor.Enter або вкладеними lock-ами. Об’єкт буде звільнений, коли буде виконано відповідну кількість разів Monitor.Exit або відбудеться вихід із самої зовнішньої конструкції lock. Тому припустима природна семантика, коли один метод викликає інший наступним чином:

static object x = new object();

static void Main(){

lock (x) {

Console.WriteLine(“I have the lock”);

Nest();

Console.WriteLine(“I still have the lock”);

}

}

static void Nest() {

lock (x) {

...

}

}

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

### Коли потрібно блокувати

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

class ThreadUnsafe {

static int x;

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

}

А ось їх потокобезпечні варіанти:

class ThreadUnsafe{

static object locker = new object();

static int x;

static void Increment() {

lock (locker) x++;

}

static void Assign() {

lock (locker) x = 123;

}

}

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

### Блокування і атомарність

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

lock (locker) {

if (x != 0) y /= x;

}

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

### Продуктивність при блокуванні

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

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

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

### Mutex

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

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

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

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

class OneAtATimePlease {

static Mutex mutex = new Mutex(false, "oreilly.com OneAtATimeDemo");

static void Main() {

// Очікуємо на отримання м'ютексу 5 сек – якщо вже є то запущений

// екземпляр додатку - завершується.

if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false)) {

Console.WriteLine("В системі запущено інший екземпляр програми!"

return;

}

try {

Console.WriteLine("Працюємо - натисніть Enter для виходу...");

Console.ReadLine();

}

finally { mutex.ReleaseMutex(); }

}

}

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

### Semaphore

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

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

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

class SemaphoreTest {

static Semaphore s = new Semaphore(3, 3); // Available=3; Capacity=3

static void Main() {

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

new Thread(Go).Start();

}

static void Go() {

while (true) {

s.WaitOne();

// Тільки 3 потоки можуть бути тут одночасно

Thread.Sleep(100);

s.Release();

}

}

}

### Потокова безпека

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

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

* розробка з урахуванням повної потокової безпеки може бути дуже трудомісткою, особливо якщо тип має багато полів (кожне поле потенційно може брати участь в багатопотоковій взаємодії);
* потокова безпека може позначитися на продуктивності (незалежно від того, чи використовується реально багатопоточність);
* потокобезпечний тип не обов’язково робить всю програму потокобезпечною, а подальша робота щодо її забезпечення може зробити потокобезпечність типу надлишковою.

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

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

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

### Потокобезпечність і типи. NET Framework

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

class ThreadSafe() {

static List <string> list = new List <string>();

static void Main() {

new Thread(AddItems).Start();

new Thread(AddItems).Start();

}

static void AddItems() {

for (int i = 0; i < 100; i++)

lock (list)

Add("Item " + list.Count);

string[] items;

lock (list)

items = list.ToArray();

foreach (string s in items)

Console.WriteLine(s);

}

}

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

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

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

if (!myList.Contains(newItem))

myList.Add(newItem);

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

myList.Clear();

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

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

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

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

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

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

### Interrupt і Abort

Заблокований потік може бути передчасно розблоковано двома шляхами:

* за допомогою Thread.Interrupt;
* за допомогою Thread.Abort.

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

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

class Program {

static void Main() {

Thread t = new Thread(delegate() {

try {

Thread.Sleep(Timeout.Infinite);

} catch(ThreadInterruptedException) {

Console.Write("Error! ");

}

Console.WriteLine("Woken!");

});

t.Start();

t.Interrupt();

}

}

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

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

if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)

worker.Interrupt();

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

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

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

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

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*

Програмно реалізувати схему, коли один потік, чекає іншого як показано в таблиці. Після досягнення кінця вектора потік закінчує своє виконання.

| **Т0** | **Т1** |
| --- | --- |
| Затримка на 2 секунди.  Сигнал в Т1 (SignT1). | Очікувати Т0 (WaitT0).  Вивести на екран «OK». |

*Розв’язання завдання 1*

На рисунку 2.1 представлено схему взаємодії потоків відповідно до завдання.

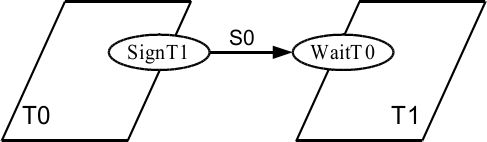


Рис. 2.1. Схема взаємодії потоків

У якості сигналу SignT1 використовується вивільнення семафора.

Один із варіантів реалізації програми подано нижче:

namespace Lab2 {

class Program {

static Semaphore S0 = new Semaphore(0, 1);

static void Main(string[] args) {

Thread T0 = new Thread(Func0);

Thread T1 = new Thread(Func1);

T0.Start();

T1.Start();

T0.Join();

T1.Join();

}

static void Func0() {

Thread.Sleep(2000);

S0.Release(); // звільнити семафор

}

static void Func1() {

S0.WaitOne(); // блокувати потік

Console.WriteLine("OK");

}

}

}

*Завдання № 2*

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

| **Т0** | **Т1** | **Т2** |
| --- | --- | --- |
| Затримка на 2 с.  Сигнал в Т1 (SignT1).  Сигнал в Т2 (SignT2). | Очікувати Т0 (WaitT0).  Вивести на екран  «OK1». | Очікувати Т0 (WaitT0).  Вивести на екран  «OK2». |

*Розв’язання завдання 2*

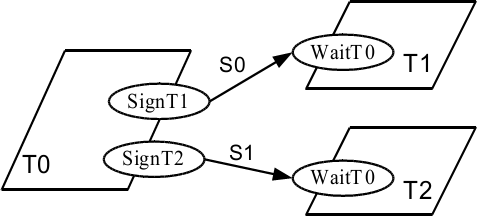
На рисунку 2.2 представлено схему взаємодії потоків відповідно до завдання.

Рис. 2.2. Схема взаємодії потоків

Для вирішення даного завдання використаємо два окремі семафори.

class Program {

static Semaphore S0 = new Semaphore(0, 1);

static Semaphore S1 = new Semaphore(0, 1);

static void Main(string[] args) {

Thread T0 = new Thread(Func0);

Thread T1 = new Thread(Func1);

Thread T2 = new Thread(Func2);

T0.Start();

T1.Start();

T2.Start();

T0.Join();

T1.Join();

T2.Join();

}

static void Func0() {

Thread.Sleep(2000);

S0.Release();

S1.Release();

}

static void Func1() {

S0.WaitOne();

Console.WriteLine("OK1");

}

static void Func2() {

S1.WaitOne();

Console.WriteLine("OK2");

}

}

*Завдання № 3*

Попереднє завдання реалізувати з використанням множинних семафорів.

*Розв’язання завдання 3*

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

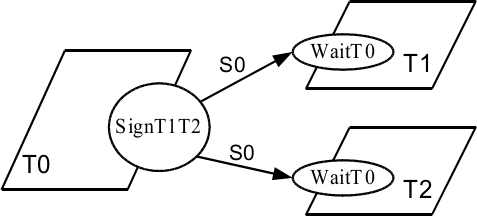


Рис. 2.3. Схема взаємодії потоків для множинного семафора

class Program {

static Semaphore S0 = new Semaphore(0, 2);

static void Main(string[] args){

Thread T0 = new Thread(Func0);

Thread T1 = new Thread(Func1);

Thread T2 = new Thread(Func2);

T0.Start();

T1.Start();

T2.Start();

T0.Join();

T1.Join();

T2.Join();

}

static void Func0(){

Thread.Sleep(2000);

S0.Release(2);

}

static void Func1(){

S0.WaitOne();

Console.WriteLine("OK1");

}

static void Func2(){

S0.WaitOne();

Console.WriteLine("OK2");

}

}

*Завдання № 4*

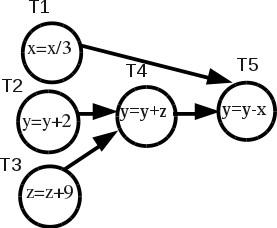
Задано граф передування (рисунок 2.4). Використовуючи семафори, виконати синхронізацію потоків. Змінні x, y та z є глобальними для всіх потоків та проініціалізовані наступним чином: x = 4, y = 2, z = 5.

Рис. 2.4. Граф передування

*Розв’язання завдання 4*

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

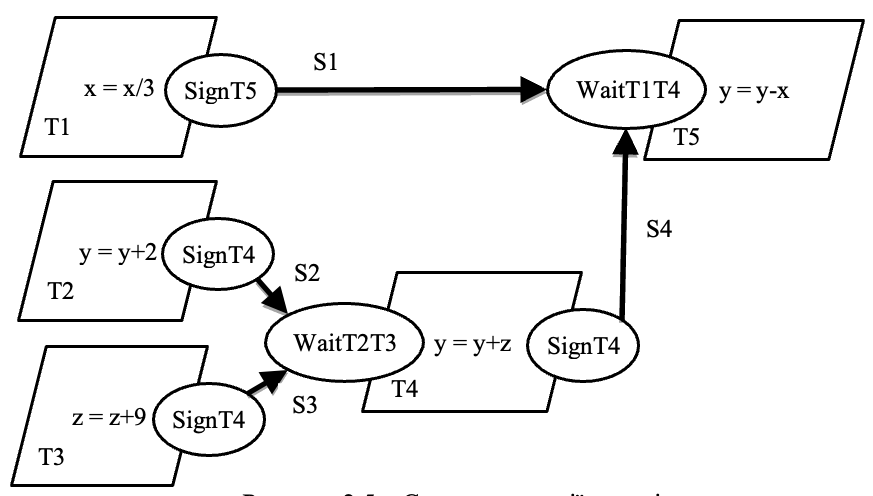


Рис. 2.5. Схема взаємодії потоків

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

class Program{

static double x = 4, y = 2, z = 5;

static Semaphore S1 = new Semaphore(0, 1);

static Semaphore S2 = new Semaphore(0, 1);

static Semaphore S3 = new Semaphore(0, 1);

static Semaphore S4 = new Semaphore(0, 1);

static void Main(string[] args) {

Thread T1 = new Thread(() => {

x /= 3; S1.Release();

});

Thread T2 = new Thread(() => {

y += 2;

S2.Release();

});

Thread T3 = new Thread(() => {

z += 9;

S3.Release();

});

Thread T4 = new Thread(() => {

S2.WaitOne();

S3.WaitOne();

y += z;

S4.Release();

});

Thread T5 = new Thread(() => {

S1.WaitOne();

S4.WaitOne(); y -= x;

});

Console.WriteLine("x = {0}; y = {1}; z = {2}", x, y, z);

T1.Start();

T2.Start();

T3.Start();

T4.Start();

T5.Start();

T1.Join();

T2.Join();

T3.Join();

T4.Join();

T5.Join();

Console.WriteLine("x = {0}; y = {1}; z = {2}", x, y, z);

Console.ReadKey();

}

}

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

class Program {

static double x = 4, y = 2, z = 5;

static Thread T1, T2, T3, T4, T5;

static void Main(string[] args) {

Console.WriteLine("x = {0}; y = {1}; z = {2}", x, y, z);

T1 = new Thread(() => { x /= 3; });

T1.Start();

T2 = new Thread(() => { y += 2; });

T2.Start();

T3 = new Thread(() => { z += 9; });

T3.Start();

T4 = new Thread(() => {

T2.Join();

T3.Join();

y += z;

});

T4.Start();

T5 = new Thread(() => {

T1.Join();

T4.Join();

y -= x;

});

T5.Start();

T5.Join();

Console.WriteLine("x = {0}; y = {1}; z = {2}", x, y, z);

Console.ReadKey();

}

}

Індивідуальні завдання.

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

| **№ варіанта** | **Граф роботи потоків** |
| --- | --- |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |

Контрольні питання

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