**Лабораторная работа №15**

***Работа с потоками***

**Теоретические сведения**

**Понятие потока**

C# поддерживает параллельное выполнение кода через многопоточность. Поток – это независимый путь исполнения, способный выполняться одновременно с другими потоками.

Программа на C# запускается как единственный поток, автоматически создаваемый CLR и операционной системой (“главный” поток), и становится многопоточной при помощи создания дополнительных потоков.

Пример 1.

class ThreadTest

{

static void Main()

{

Thread t = new Thread(WriteY);

t.Start(); // Выполнить WriteY в новом потоке

while (true)

Console.Write("x"); // Все время печатать 'x'

}

static void WriteY()

{

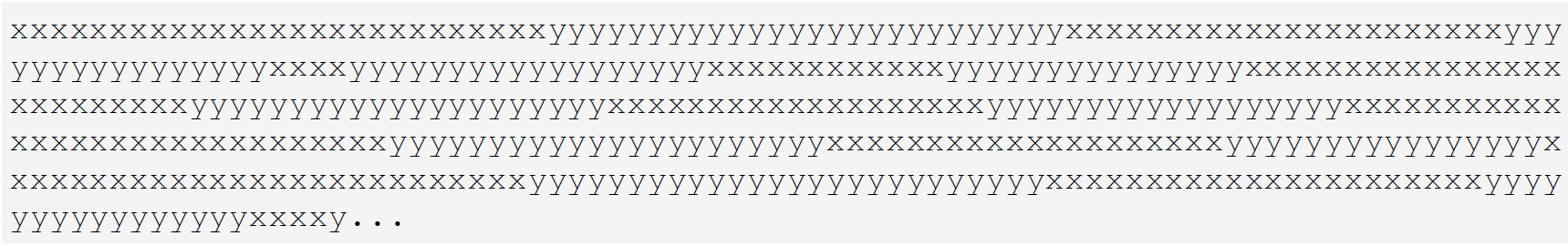
while (true)

Console.Write("y"); // Все время печатать 'y'

}

}

Вывод:



В главном потоке создается новый поток **t**, исполняющий метод, который непрерывно печатает символ ‘**y**’. Одновременно главный поток непрерывно печатает символ ‘**x**’.

CLR назначает каждому потоку свой стек, так что локальные переменные хранятся раздельно. Вместе с тем потоки разделяют данные, относящиеся к тому же экземпляру объекта, что и сами потоки. Иногда необходимо выполнить блокировку участка кода функции на время обработки данных. Выполнение такой операции состоит в получении эксклюзивной блокировки на время чтения и записи разделяемых полей. C# обеспечивает это при помощи оператора lock.

Пример 2

static void Go()

{

lock (locker)

{

if (!done)

{

Console.WriteLine("Done");

done = true;

}

}

}

Когда два потока одновременно борются за блокировку (в нашем случае объекта locker), один поток переходит к ожиданию (блокируется), пока блокировка не освобождается. В данном случае это гарантирует, что только один поток может одновременно исполнять критическую секцию кода, и "Done" будет напечатано только один раз. Код, защищенный таким образом от неопределѐнности в плане многопоточного исполнения, называется потокобезопасным.

Временная приостановка (блокирование) – основной способ координации, или синхронизации действий потоков. Ожидание эксклюзивной блокировки – это одна из причин, по которым поток может блокироваться. Другая причина – если поток приостанавливается (Sleep) на заданный промежуток времени:

Thread.Sleep(TimeSpan.FromSeconds(30)); // Блокировка на 30сек

Также поток может ожидать завершения другого потока, вызывая его метод Join:

Thread t = new Thread(Go); // Go – статический метод

t.Start();

t.Join(); // Ожидаем завершения потока

Будучи заблокированным, поток не потребляет ресурсов CPU.

Управление многопоточностью осуществляет планировщик потоков, эту функцию CLR обычно делегирует операционной системе. Планировщик потоков гарантирует, что активным потокам выделяется соответствующее время на выполнение, а потоки, ожидающие или блокированные, к примеру, на ожидании эксклюзивной блокировки, или пользовательского ввода – не потребляют времени CPU.

На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков. Это приводит к непредсказуемому поведению, как в самом первом примере, где каждая последовательность символов „X‟ и „Y‟ соответствует кванту времени, выделенному потоку. В Windows XP типичное значение кванта времени – десятки миллисекунд – выбрано как намного большее, чем затраты CPU на переключение контекста между потоками (несколько микросекунд).

Говорят, что поток вытесняется, когда его выполнение приостанавливается из-за внешних факторов типа квантования времени. В большинстве случаев поток не может контролировать, когда и где он будет вытеснен.

**Потоки и процессы**

Все потоки одного приложения логически содержатся в пределах процесса – модуля операционной системы, в котором исполняется приложение.

В некоторых аспектах потоки и процессы схожи – например, время разделяется между процессами, исполняющимися на одном компьютере, так же, как между потоками одного C#-приложения. Ключевое различие состоит в том, что процессы полностью изолированы друг от друга. Потоки разделяют память (кучу) с другими потоками этого же приложения. Благодаря этому один поток может поставлять данные в фоновом режиме, а другой – показывать эти данные по мере их поступления.

**Создание и запуск потоков**

Для создания потоков используется конструктор класса Thread, принимающий в качестве параметра делегат типа ThreadStart, указывающий метод, который нужно выполнить. Делегат ThreadStart определяется так:

public delegate void ThreadStart();

Вызов метода Start начинает выполнение потока. Поток продолжается до выхода из исполняемого метода. Вот пример, использующий полный синтаксис C# для создания делегата ThreadStart:

class ThreadTest

{

static void Main()

{

Thread t = new Thread(new ThreadStart(Go));

t.Start(); // Выполнить Go() в новом потоке.

Go(); // Одновременно запустить Go() в главном потоке.

}

static void Go() { Console.WriteLine("hello!"); }

}

Поток можно создать, используя для присваивания значений делегатам более удобный сокращенный синтаксис C#:

static void Main()

{

Thread t = new Thread(Go); // Без явного использования ThreadStart

t.Start();

...

}

static void Go() { ... }

Поток имеет свойство IsAlive, возвращающее true после вызова Start() и до завершения потока.

Поток, который закончил исполнение, не может быть начат заново.

**Именование потоков**

Поток можно поименовать, используя свойство Name. Это предоставляет большое удобство при отладке: имена потоков можно вывести в Console.WriteLine и увидеть в окне Debug – Threads в Microsoft Visual Studio. Имя потоку может быть назначено в любой момент, но только один раз – при попытке изменить его будет сгенерировано исключение.

Главному потоку приложения также можно назначить имя – в следующем примере доступ к главному потоку осуществляется через статическое свойство CurrentThread класса 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);

}

}

По умолчанию потоки создаются как основные, что означает, что приложение не будет завершено, пока один из таких потоков будет исполняться. C# также поддерживает фоновые потоки, они не продлевают жизнь приложению, а завершаются сразу же, как только все основные потоки будут завершены.

Статус потока переключается с основного на фоновый при помощи свойства IsBackground

**Приоритеты потоков**

|  |
| --- |
| Свойство Priority определяет, сколько времени на исполнение будет выделено потоку относительно других потоков того же процесса. Существует 5 градаций приоритета потока:  enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest } |

Значение приоритета становится существенным, когда одновременно исполняются несколько потоков.

Установка приоритета потока на максимум еще не означает работу в реальном времени (real-time), так как существуют еще приоритет процесса приложения.

**Важнейшие средства синхронизации**

В следующих таблицах приведена информация об инструментах .NET для координации (синхронизации) потоков:

Таблица 1. Простейшие методы блокировки

|  |  |
| --- | --- |
| Конструкция | Назначение |
| Sleep | Блокировка на указанное время |
| |  | | --- | | Join | | Ожидание окончания другого потока |

Таблица 2. Блокировочные конструкции

|  |  |  |  |
| --- | --- | --- | --- |
| Конструкция | Назначение | Доступность из других процессов | Скорость |
| lock | Гарантирует, что только один поток может получить доступ к ресурсу или секции кода. | нет | быстро |
| Mutex | Гарантирует, что только один поток может получить доступ к ресурсу или секции кода. Может использоваться для предотвращения запуска нескольких экземпляров приложения. | да | средне |
| Semaphore | Гарантирует, что не более заданного числа потоков может получить доступ к ресурсу или секции кода. | да | средне |

Таблица 3. Сигнальные конструкции

|  |  |  |  |
| --- | --- | --- | --- |
| Конструкция | Назначение | Доступность из других процессов | Скорость |
| EventWaitHandle | Позволяет потоку ожидать сигнала от другого потока. | да | средне |
| Wait and Pulse | Позволяет потоку ожидать, пока не выполнится заданное условие блокировки. | нет | средне |

Таблица 4. Не блокирующие конструкции

|  |  |  |  |
| --- | --- | --- | --- |
| Конструкция | Назначение | Доступность из других процессов | Скорость |
| Interlocked | Выполнение простых не блокирующих атомарных операций. | Да – через разделяемую память | очень быстро |
| volatile | Для безопасного не блокирующего доступа к полям. | Да – через разделяемую память | очень быстро |

**Блокировка**

Когда поток остановлен в результате использования конструкций, перечисленных в вышеприведенных таблицах, говорят, что он блокирован. Будучи блокированным, поток немедленно перестает получать время CPU, устанавливает свойство ThreadState в WaitSleepJoin и остается в таком состоянии, пока не разблокируется. Разблокировка может произойти в следующих четырех случаях:

* выполнится условие разблокировки;
* истечет таймаут операции (если он был задан);
* по прерыванию через Thread.Interrupt;
* по аварийному завершению через Thread.Abort.

Поток не считается блокированным, если его выполнение приостановлено нерекомендуемым методом Suspend.

**Sleeping**

Вызов Thread.Sleep блокирует текущий поток на указанное время (либо до прерывания):

static void Main()

{

Thread.Sleep(0); // отказаться от одного кванта времени CPU

Thread.Sleep(1000); // заснуть на 1000 миллисекунд

Thread.Sleep(TimeSpan.FromHours(1)); // заснуть на 1 час

Thread.Sleep(Timeout.Infinite); // заснуть до прерывания

}

Если быть более точным, Thread.Sleep отпускает CPU и сообщает, что потоку не должно выделяться время в указанный период. Thread.Sleep(0) отпускает CPU для выделения одного кванта времени следующему потоку в очереди на исполнение.

**Постановка задачи**

Реализовать многопоточное приложение. Для каждого из вариантов необходимо реализовать возможность задавать приоритет каждого из порожденных потоков. Все приложения должны иметь графический интерфейс.

**Варианты:**

|  |  |
| --- | --- |
| **Вариант** |  |
| **1** | Поиск указанной строки в указанном файле. Обработка одной строки в порожденном потоке. |
| **2** | Умножение матрицы на вектор. Обработка одной строки матрицы - в порожденном потоке. |
| **3** | Поиск всех простых чисел (простым называется число, которое является своим наибольшим делителем) в указанном интервале чисел, разделенном на несколько диапазонов. Обработка каждого диапазона производится в порожденном потоке. Классический алгоритм Евклида определения наибольшего общего делителя двух целых чисел (x, y) может применяться при следующих условиях:   * оба числа x и y неотрицательные; * оба числа x и y отличны от нуля.   На каждом шаге алгоритма выполняются сравнения:   * если x == y, то ответ найден; * если x < y, то y заменяется значением y - x; *  если x > y, то x заменяется значением x - y. |
| **4** | Warcraft. Заданное количество юнитов добывают золото равными порциями из одной шахты, задерживаясь в пути на случайное время, до ее истощения. Работа каждого юнита реализуется в порожденном процессе потоке. |
| **5** | Винни-Пух и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Винни-Пух потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Работа каждой пчелы реализуется в порожденном потоке. |
| **6** | Шарики. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали. При выпадении шарика за нижнюю границу допустимой области шарик исчезает. Изменение координат каждого шарика в отдельном потоке. |
| **7** | Противостояние нескольких команд. Каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Борьба каждой команды реализуется в отдельном потоке. |
| **8** | Контрольная сумма. Для нескольких файлов (разного размера) требуется вычислить контрольную сумму (сумму кодов всех символов файла). Обработка каждого файла выполняется в отдельном потоке. |
| **9** | Авиаразведка. Создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (потоков), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фиксирует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. |
| **10** | Бег с препятствиями. Создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (потоки) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. |
| **11** | Создать поток, который управляет перемещением кнопки/панели. Кнопка движется по Х. Создать 3 кнопки и устроить "тараканьи" бега среди них. |
| **12** | Параллельно отрисовывать графики функций: y = sin(x), y = 4\*x2-2\*x – 22, y = ln(x2)/x3. |
| **13** | Однорукий бандит - 3 потока, генерирующие числа от 0 до 9. По нажатию кнопки потоки останавливаются и результат анализируется. При анализе использовать следующие комбинации (три одинаковых числа, два одинаковых числа, три единицы, три семерки, две единицы, имеется четверка) |
| **14** | Вычислить 10-ю степень двойки 1 - сложением, умножением и просто возведением в степень. |
| **15** | Создать игру, где будут 2-3 барана и волк. При совпадении координат волка с бараном баран исчезает. При совпадении координат баранов появляется новый баран. Все движутся хаотически. |
| **16** | Создать два потока. Первый ищет числа Фибоначчи (каждое последующее число равно сумме двух предыдущих чисел), второй простые числа. Результат работы каждого потока сохраняются в отдельный файл. После остановки потока – программа производит анализ файлов, выводит их на экран, а так же показывает количество найденных чисел Фибоначчи и простых чисел. |
| **17** | Создать два потока. Каждый поток стремится установить бегунок компонента TrackBar в положение 25 или 75. |
| **18** | Создать два потока. Первый поток производит запись в файл случайных данных. Второй производит чтение данных из этого файла и вывод их на экран. |
| **19** | Создать приложение выполняющее построение графика функции y = 23\*x2 – 33, с шагом x=0.01. Первый поток производит расчет данных функции и добавляет их в конец массива данных. Второй поток извлекает из массива данных значения и производит выводит функции на экран. |
| **20** | Создайте приложение, в отдельном потоке вычисляющее значение π и непрерывно обновляющего его в пользовательском интерфейсе. Для расчета числа использовать следующую формулу. |
| **21** | Создать приложение выполняющее сортировку массива данных и визуального отображения процесса сортировки на экране. Первый поток производит сортировку по возрастанию, второй по убыванию. После каждого перемещения элемента результирующий производится графический вывод текущего состояния сортировки. Каждый поток работает с отдельным экземпляром массива данных. Состояние сортировки выводится в двух элементах. |
| **22** | Создать приложение выполняющее сортировку массива данных и визуального отображения процесса сортировки на экране. Первый массив производит сортировку по возрастанию, второй по убыванию, третий производит графический вывод массива на экран. Каждый поток работает с общим экземпляром массива данных. Состояние сортировки выводится в одном элементе. |
| **23** | Создать приложение выполняющее перемещение кнопки по форме. Первый поток производит перемещение кнопки согласно текущих значений приращения х, у. Второй поток производит расчет текущего значения приращения х,у. У пользователя должна быть возможность менять правило расчета х,у – выбрав из выпадающего списка определенный элемент. Выпадающий список содержит элементы – “не перемещаться”, "по прямой", "sin(x)", "cos(x)". |
| **24** | Создайте приложение, в отдельном потоке вычисляющее значение w и непрерывно обновляющего его в пользовательском интерфейсе. Для расчета числа использовать следующую формулу.  W = 1 + sin(x) – 2\*cos(x) + 4\*sin2(x) – 8\*cos2(x) +.. |
| **25** | Создать приложение содержащих два порожденных потока. Первый поток производит добавление в элемент RichTextBox новой строки, второй поток удаление последней строки. При невозможности удаления строки поток должен приостанавливаться на время необходимое первому потоку для добавления новой строки. |