Многопоточная обработка

Введение

Поток – это путь выполнения, который может проходить независимо от других таких путей.

Каждый поток запускается внутри процесса ОС, который предоставляет изолированную среду для выполнения программы. В однопоточной программе внутри изолированной среды процесса функционирует только один поток, поэтому он получает монопольный доступ к среде. В многопоточной программе внутри единственного процесса запускается множество потоков, разделяя одну и ту же среду выполнения (скажем, память). Отчасти это одна из причин, почему полезна многопоточность: например, один поток может извлекать данные в фоновом режиме, в то время как другой поток – отображать их по мере поступления. Такие данные называются разделяемым состоянием.

Создание потока

Клиентская программа (консольная, WPF, UWP или Windows Forms) запускается в единственном потоке, который создается автоматически операционной системой (“главный” поток). Здесь он и будет существовать как однопоточное приложение, если только вы не создадите дополнительные потоки (прямо или косвенно).

Создать и запустить новый поток можно за счет создания объекта Thread и вызова его метода *Start*. Простейший конструктор *Thread* принимает делегат *ThreadStart*: метод без параметров, который указывает, где должно начинаться выполнение.

```
using System;
using System.Threading;
Thread t = new Thread(WriteY); // Начать новый поток
t.Start(); // выполняющий WriteY
           // Одновременно делать что-то в главном потоке
for (int i = 0; i < 1000; i++)
  Console.Write("x");
void WriteY()
{
  for (int i = 0; i < 1000; i++)
    Console.Write("y");
}
```

Ниже показан типичный вывод:

xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...

Главный поток создает новый поток t, в котором запускает метод, многократно выводящий символ “y”. В то же самое время главный поток многократно выводит символ “x” (см. рисунок). На компьютере с одноядерным процессором ОС должна выделять каждому потоку кванты времени (обычно размером 20 мс. в среде Windows) для эмуляции параллелизма, что дает в результате повторяющиеся блоки вывода “x” и “y”. На многоядерной или многопроцессорной машине два потока могут выполняться по-настоящему параллельно (конкурируя с другими активными процессами в системе), хотя в рассматриваемом примере все равно будут получаться повторяющиеся блоки вывода “x” и “y” из-за тонкостей работы механизма, которым класс Console обрабатывает параллельные запросы.

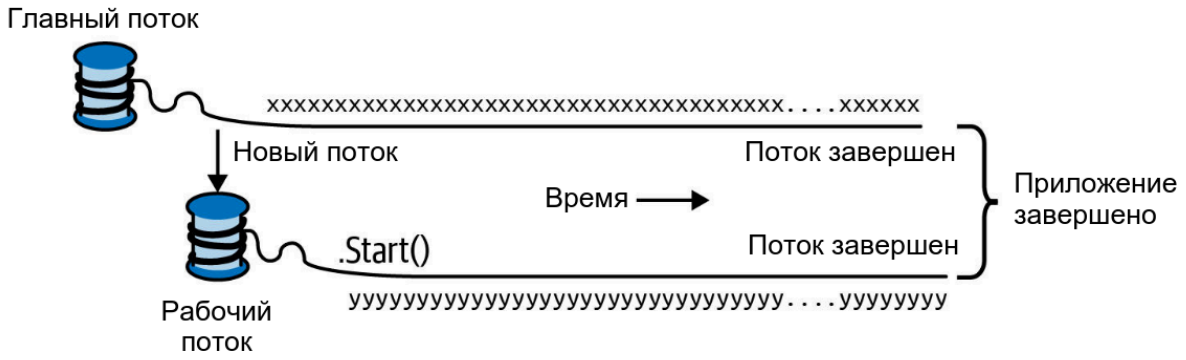

Рис. Начало нового потока

❖ Говорят, что поток *вытесняется* в точках, где его выполнение пересекается с выполнением кода в другом потоке. К этому термину часто прибегают при объяснении, почему что-то пошло не так, как было задумано!

После запуска свойство *IsAlive* потока возвращает *true* до тех пор, пока не будет достигнута точка, где поток завершается. Поток заканчивается, когда завершает выполнение делегат, переданный конструктору класса *Thread*. После завершения поток не может быть запущен повторно.

Каждый поток имеет свойство *Name*, которое можно установить для содействия отладке. Это особенно полезно в Visual Studio, т.к. имя потока отображается в окне Threads (Потоки) и в панели инструментов Debug Location (Местоположение отладки). Установить имя потока можно только один раз; попытки изменить его позже приведут к генерации исключения.

Статическое свойство *Thread.CurrentThread* возвращает поток, выполняющийся в текущее время:

```
Console.WriteLine(Thread.CurrentThread.Name);
```

Join и Sleep

С помощью вызова метода *Join* можно организовать ожидание окончания другого потока:

```
Thread t = new Thread(Go);
t.Start();
t.Join();
Console.WriteLine("Thread t has ended!"); // Поток t завершен!
void Go() { for (int i = 0; i < 1000; i++) Console.Write("y"); }
```

Код выводит на консоль символ “*y*” тысячу раз и затем сразу же строку “*Thread t has ended!*” При вызове метода *Join* можно указывать тайм-аут, выраженный в миллисекундах или в виде структуры *TimeSpan*. Тогда метод будет возвращать *true*, если поток был завершен, или *false*, если истекло время тайм-аута.

Метод *Thread.Sleep* приостанавливает текущий поток на заданный период:

```
Thread.Sleep(TimeSpan.FromHours(1)); // Ожидать 1 час
Thread.Sleep(500); // Ожидать 500 мс
```

Вызов *Thread.Sleep(0)* немедленно прекращает текущий квант времени потока, добровольно передавая контроль над центральным процессором (ЦП) другим потокам. Метод *Thread.Yield()* делает то же самое, но уступает контроль только потокам, функционирующим на том же самом процессоре.

❖ Вызов *Sleep(0)* или *Yield()* в производственном коде иногда полезен для расширенной настройки производительности. Это также великолепный диагностический инструмент для поиска проблем, связанных с безопасностью к потокам: если вставка вызова *Thread.Yield()* в любое место кода нарушает работу программы, то в ней почти наверняка присутствует ошибка.

На время ожидания *Sleep* или *Join* поток блокируется.

Блокирование

Поток считается заблокированным, если его выполнение приостановлено по некоторой причине, такой как вызов метода *Sleep* или ожидание завершения другого потока через вызов *Join*. Заблокированный поток немедленно уступает свой квант процессорного времени и далее не потребляет процессорное время, пока удовлетворяется условие блокировки. Проверить, заблокирован ли поток, можно с помощью его свойства *ThreadState:*

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

❖ Свойство *ThreadState* является перечислением флагов, комбинирующим три “уровня” данных в побитовой манере. Однако большинство значений являются избыточными, неиспользуемыми или устаревшими. Следующий расширяющий метод ограничивает *ThreadState* одним из четырех полезных значений : *Unstarted*, *Running*, *WaitSleepJoin* и *Stopped*:

```
public static ThreadState Simplify(this ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

Свойство ThreadState удобно для диагностических целей , но непригодно для синхронизации, т.к. состояние потока может измениться в промежутке между проверкой ThreadState и обработкой данной информации.

Когда поток блокируется или деблокируется, ОС производит переключение контекста. С ним связаны небольшие накладные расходы, обычно составляющие одну или две микросекунды.

Интенсивный ввод-вывод или интенсивные вычисления

Операция, которая большую часть своего времени тратит на ожидание, пока что-то произойдет, называется операцией с интенсивным вводом-выводом; примером может служить загрузка веб-страницы или вызов метода *Console.ReadLine*. (Операции с интенсивным вводом-выводом обычно включают в себя ввод или

вывод, но это не жесткое требование: вызов метода *Thread.Sleep* также считается операцией с интенсивным вводом-выводом.) И напротив, операция, которая большую часть своего времени затрачивает на выполнение вычислений с привлечением ЦП, называется операцией с интенсивными вычислениями.

Блокирование или зацикливание

Операция с интенсивным вводом-выводом работает одним из двух способов. Она либо синхронно ожидает завершения определенной операции в текущем потоке (такой как *Console.ReadLine*, *Thread.Sleep* или *Thread.Join*), либо работает асинхронно, инициируя обратный вызов, когда интересующая операция завершается спустя какое-то время (более подробно об этом позже).

Операции с интенсивным вводом-выводом, которые ожидают синхронным образом, большую часть своего времени тратят на блокирование потока. Они также могут периодически “прокручиваться” в цикле:

```
while (DateTime.Now < nextStartTime)
  Thread.Sleep(100);
```

Оставляя в стороне тот факт, что существуют более эффективные средства (вроде таймеров и сигнализирующих конструкций), еще одна возможность предусматривает зацикливание потока:

```
while (DateTime.Now < nextStartTime);
```

В общем случае это очень неэкономное расходование процессорного времени: среда CLR и ОС предполагают, что поток выполняет важные вычисления, и соответствующим образом выделяют ресурсы. В сущности, мы превращаем код, который должен быть операцией с интенсивным вводом-выводом, в операцию с интенсивными вычислениями.

❖ Относительно вопроса зацикливания или блокирования следует отметить пару нюансов. Во-первых, *очень кратковременное* зацикливание может быть эффективным, когда ожидается скорое (возможно в пределах нескольких микросекунд) удовлетворение некоторого условия, поскольку оно избегает накладных расходов и задержки, связанной с переключением контекста. Платформа .NET предлагает специальные методы и классы для содействия зацикливанию (см. информацию по ссылке SpinLock and SpinWait на странице http://albahari.com/threading/). Во-вторых, затраты на блокирование не являются нулевыми. Дело в том, что за время своего существования каждый поток связывает около 1 Мбайт памяти и служит источником текущих накладных расходов на администрирование со стороны среды CLR и ОС. По этой причине блокирование может быть ненадежным в контексте программ с интенсивным вводом-выводом, которые нуждаются в поддержке сотен или тысяч параллельных операций. Взамен такие программы должны использовать подход, основанный на обратных вызовах, что полностью освободит поток на время ожидания.

Локальное или разделяемое состояние

Среда CLR назначает каждому потоку собственный стек в памяти, так что локальные переменные хранятся отдельно. В следующем примере мы определяем метод с локальной переменной, после чего вызываем его одновременно в главном потоке и во вновь созданном потоке:

```
new Thread(Go).Start(); // Вызвать Go в новом потоке
Go(); // Вызвать Go в главном потоке
void Go()
{
  // Объявить и использовать локальную переменную cycles
  for (int cycles = 0; cycles < 5; cycles++)
    Console.Write("? * ");
}
```

В стеке каждого потока создается отдельная копия переменной *cycles*, так что вывод вполне предсказуемо содержит десять знаков вопроса.

Потоки разделяют данные, если они имеют общую ссылку на один и тот же экземпляр:

```
bool _done = false;
new Thread(Go).Start();
Go();
void Go()
{
  if (!_done)
  {
    _done = true;
    Console.WriteLine("Done");
  }
}
```

Оба потока разделяют переменную *_done*, поэтому слово “Done” выводится один раз, а не два.

Локальные переменные, захваченные лямбда-выражением, тоже могут быть разделяемыми:

```
bool done = false;
ThreadStart action = () =>
{
  if (!done)
  {
    done = true;
    Console.WriteLine("Done");
  }
};
new Thread(action).Start();
action();
```

Тем не менее, поля чаще применяются для разделения данных между потоками. В следующем примере в обоих потоках метод *Go* вызывается на том же самом экземпляре *ThreadTest*, так что они разделяют поле *_done*:

```
var tt = new ThreadTest();
new Thread(tt.Go).Start();
tt.Go();
class ThreadTest
{
  bool _done;
  public void Go()
  {
    if (!_done)
    {
      _done = true;
      Console.WriteLine("Done");
    }
  }
}
```

Статические поля предлагают еще один способ разделения данных между потоками:

```
class ThreadTest
{
  static bool done; // Статические поля разделяются между всеми
                    // потоками в том же самом домене приложения.
  static void Main()
  {
    new Thread(Go).Start();
    Go();
  }
  static void Go()
  {
    if (!done)
    {
      done = true;
      Console.WriteLine("Done");
    }
  }
}
```

Все четыре примера иллюстрируют еще одну ключевую концепцию: безопасность в отношении потоков (или наоборот – ее отсутствие). Вывод в действительности не определен: возможно (хотя и маловероятно), что слово “Done” будет выведено дважды. Однако если мы поменяем местами порядок следования операторов в методе *Go*, то вероятность двукратного вывода слова “Done” значительно возрастет:

```
static void Go()
{
  if (!done)
  {
    Console.WriteLine("Done");
    done = true;
  }
}
```

Проблема в том, что один поток может оценивать оператор *if* точно в то же самое время, когда второй поток выполняет оператор *WriteLine* – до того, как он получит шанс установить поле *_done* в *true*.

❖ Приведенный пример демонстрирует одну из многочисленных ситуаций, в которых разделяемое записываемое состояние может привести к возникновению определенной разновидности несистематических ошибок, характерных для многопоточности. В следующем разделе мы покажем, как с помощью блокировки устранить проблемы; тем не менее, по возможности лучше вообще избегать применения разделяемого состояния. Позже мы объясним, как в этом могут помочь шаблоны асинхронного программирования.

Блокировка и безопасность потоков

Исправить предыдущий пример можно, получив монопольную блокировку на период чтения и записи разделяемого поля. Для этой цели в языке C# предусмотрен оператор *lock*.

Когда два потока одновременно соперничают за блокировку (что может возникать с любым объектом ссылочного типа; в рассматриваемом случае *_locker*), один из потоков ожидает, или блокируется, до тех пор, пока блокировка не станет доступной . В таком случае гарантируется, что только один поток может войти в данный блок кода за раз, и строка *Done* будет выведена лишь однократно. Код, защищенный подобным образом (от неопределенности в многопоточном контексте), называется безопасным в отношении потоков.

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();
  static void Main()
  {
    new Thread(Go).Start();
    Go();
  }
  static void Go()
  {
    lock (_locker)
    {
      if (!_done)
      {
        Console.WriteLine("Done");
        _done = true;
      }
    }
  }
}
```

❖ Даже действие автоинкрементирования переменной не является безопасным к потокам: выражение *x++* выполняется на лежащем в основе процессоре как отдельные операции чтения, инкремента и записи. Таким образом, если два потока выполняют *x++* одновременно за пределами блокировки, то переменная *x* в итоге может быть инкрементирована один раз, а не два (или, что еще хуже, в определенных обстоятельствах переменная *x* может быть вообще *разрушена*, получив смесь битов старого и нового содержимого).

Блокировка не является панацеей для обеспечения безопасности потоков – довольно легко забыть заблокировать доступ к полю и тогда блокировка сама может создать проблемы (наподобие состояния взаимоблокировки).

Хорошим примером применения блокировки может служить доступ к разделяемому кешу внутри памяти для часто используемых объектов базы данных в приложении ASP.NET Core. Приложение такого вида очень просто заставить работать правильно без возникновения взаимоблокировки.

Передача данных потоку

Иногда требуется передать аргументы начальному методу потока. Проще всего это сделать с использованием лямбда-выражения, которое вызывает данный метод с желаемыми аргументами:

```
Thread t = new Thread(() => Print("Hello from t!"));
t.Start();
void Print(string message) => Console.WriteLine(message);
```

Такой подход позволяет передавать методу любое количество аргументов. Можно даже поместить всю реализацию в лямбда-функцию с множеством операторов:

```
new Thread(() =>
{
  Console.WriteLine("I'm running on another thread!");
  Console.WriteLine("This is so easy!");
}).Start();
```

Альтернативный (менее гибкий) прием предусматривает передачу аргумента методу *Start* класса *Thread*:

```
Thread t = new Thread(Print);
t.Start("Hello from t!");
void Print(object messageObj)
{
  string message = (string)messageObj; // Здесь необходимо
приведение
  Console.WriteLine(message);
}
```

Код работает потому, что конструктор класса Thread перегружен для приема одного из двух делегатов:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

Лямбда-выражения и захваченные переменные

Как уже должно быть понятно, лямбда-выражение является наиболее удобным и мощным способом передачи данных потоку. Однако следует соблюдать осторожность, чтобы случайно не изменить захваченные переменные после запуска потока. Например, рассмотрим следующий код:

```
for (int i = 0; i < 10; i++)
{
  new Thread(() => Console.Write(i)).Start();
}
```

Вывод будет недетерминированным! Вот типичный результат:
0223557799

Проблема в том, что на протяжении всего времени жизни цикла переменная *i* ссылается на ту же самую ячейку в памяти. Следовательно, каждый поток вызывает метод *Console.Write* с переменной, значение которой может измениться в ходе его выполнения! Решение заключается в применении временной переменной, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread(() => Console.Write(temp)).Start();
}
```

Теперь все цифры от 0 до 9 будут выводиться в точности по одному разу. (*Порядок* вывода по-прежнему не определен, т.к. потоки могут запускаться в неопределимые моменты времени.)

Переменная *temp* является локальной по отношению к каждой итерации цикла. Таким образом, каждый поток захватывает отличающуюся ячейку памяти, и проблемы не возникают. Проблему в приведенном ранее коде проще проиллюстрировать с помощью показанного далее примера:

```
string text = "t1";
Thread t1 = new Thread(() => Console.WriteLine(text));
text = "t2";
Thread t2 = new Thread(() => Console.WriteLine(text));
t1.Start(); t2.Start();
```

Поскольку оба лямбда-выражения захватывают одну и ту же переменную *text*, строка “t2” выводится дважды.

Обработка исключений

Любые блоки *try/catch/finally*, действующие во время создания потока, не играют никакой роли в потоке, когда он начинает свое выполнение. Взгляните на следующую программу:

```
try
{
  new Thread(Go).Start();
}
catch (Exception ex)
{
  // Сюда мы никогда не попадем!
  Console.WriteLine("Exception!"); // Исключение!
}
void Go()
{
```

```
  throw null;
} // Генерирует исключение NullReferenceException
```

Оператор *try/catch* здесь безрезультатен, и вновь созданный поток будет обременен необработанным исключением *NullReferenceException*. Такое поведение имеет смысл, если принять во внимание тот факт, что каждый поток обладает независимым путем выполнения.

Чтобы исправить ситуацию, обработчик событий потребуется переместить внутрь метода *Go*:

```
new Thread(Go).Start();
void Go() {
  try
  {
    throw null; //Исключение NullReferenceException будет перехвачено ниже
  }
  catch (Exception ex)
  {
    // ...
    // Обычно необходимо зарегистрировать исключение в журнале
    // и/или сигнализировать другому потоку об отсоединении
    // ...
  }
}
```

В производственных приложениях необходимо предусмотреть обработчики исключений для всех методов входа в потоки – в точности как это делается в главном потоке (обычно на более высоком уровне в стеке выполнения). Необработанное исключение приведет к прекращению работы всего приложения, да еще и с отображением безобразного диалогового окна!

❖ При написании таких блоков обработки исключений вы редко будете *игнорировать* ошибку: обычно вы зарегистрируете в журнале подробности исключения. Для клиентского приложения, возможно, вы отобразите диалоговое окно, позволяющее пользователю автоматически отправить подробные сведения веб-серверу. Затем, по всей видимости, вы решите перезапустить приложение, поскольку существует возможность того, что непредвиденное исключение оставило приложение в недопустимом состоянии.

Централизованная обработка исключений

В приложениях WPF, UWP и Windows Forms можно подписываться на “глобальные” события обработки исключений – *Application.DispatcherUnhandledException* и *Application.ThreadException*. Они инициируются после возникновения необработанного исключения в любой части программы, которая вызвана в цикле сообщений (сказанное относится ко всему коду, выполняющемуся в главном потоке, пока активен экземпляр Application). Прием полезен в качестве резервного средства для регистрации и сообщения об ошибках (хотя он неприменим для необработанных исключений , которые возникают в созданных вами потоках, не относящихся к пользовательскому интерфейсу). Обработка упомянутых событий предотвращает аварийное завершение программы, хотя впоследствии может быть принято решение о ее перезапуске во избежание потенциального разрушения состояния, к которому может привести необработанное исключение.

Потоки переднего плана или фоновые потоки

По умолчанию потоки, создаваемые явно, являются потоками переднего плана. Потоки переднего плана удерживают приложение в активном состоянии до тех пор, пока хотя бы один из них выполняется, но фоновые потоки этого не делают. После того, как все потоки переднего плана прекратят свою работу, заканчивается и приложение, а любые все еще выполняющиеся фоновые потоки будут принудительно завершены.

Выяснить либо изменить фоновое состояние потока можно с использованием его свойства *IsBackground*:

```
static void Main(string[] args)
{
  Thread worker = new Thread(() => Console.ReadLine());

  if (args.Length > 0)
    worker.IsBackground = true;
```

```
  worker.Start();
}
```

Если запустить такую программу без аргументов, тогда рабочий поток предполагает, что она находится в фоновом состоянии, и будет ожидать в операторе *ReadLine* нажатия пользователем клавиши . Тем временем главный поток завершается, но приложение остается запущенным, потому что поток переднего плана все еще активен. С другой стороны, если методу *Main* передается аргумент, то рабочему потоку назначается фоновое состояние, и программа завершается почти сразу после завершения главного потока (прекращая выполнение метода *ReadLine*).

Когда процесс прекращает работу подобным образом, любые блоки *finally* в стеке выполнения фоновых потоков пропускаются. Если программа задействует блоки *finally* (или *using*) для проведения очистки вроде удаления временных файлов, то вы можете избежать этого, явно ожидая такие фоновые потоки вплоть до завершения приложения, либо за счет присоединения к потоку, либо с помощью сигнализирующей конструкции (см. раздел “Передача сигналов” далее). В любом случае должен быть указан тайм-аут, чтобы можно было уничтожить поток, который отказывается завершаться, иначе приложение не сможет быть нормально закрыто без привлечения пользователем диспетчера задач (или команды *kill* в Unix).

Потоки переднего плана не требуют такой обработки, но вы должны позаботиться о том, чтобы избежать ошибок, которые могут привести к отказу завершения потока. Обычной причиной отказа в корректном завершении приложений является наличие активных фоновых потоков.

Приоритет потока

Свойство *Priority* потока определяет, сколько времени на выполнение получит данный поток относительно других активных потоков в ОС, со следующей шкалой значений:

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

Это становится важным, когда одновременно активно несколько потоков. Увеличение приоритета потока должно производиться осторожно, т.к. может привести к торможению других потоков. Если нужно, чтобы поток имел больший приоритет, чем потоки в других процессах, тогда потребуется также увеличить приоритет процесса с применением класса *Process* из пространства имен *System.Diagnostics*:

```
using Process p = Process.GetCurrentProcess();
p.PriorityClass = ProcessPriorityClass.High;
```

Прием может нормально работать для потоков, не относящихся к пользовательскому интерфейсу, которые выполняют минимальную работу и нуждаются в низкой задержке (т.е. возможности реагировать очень быстро). В приложениях с обильными вычислениями (особенно в тех, которые имеют пользовательский интерфейс) увеличение приоритета процесса может приводить к торможению других процессов и замедлению работы всего компьютера.

Передача сигналов

Иногда нужно, чтобы поток ожидал получения уведомления (уведомлений) от другого потока (потоков). Это называется передачей сигналов. Простейшей сигнализирующей конструкцией является класс *ManualResetEvent*. Вызов метода *WaitOne* класса *ManualResetEvent* блокирует текущий поток до тех пор, пока другой поток не “откроет” сигнал, вызвав метод *Set*. В приведенном ниже примере мы запускаем поток, который ожидает события

ManualResetEvent. Он останется заблокированным в течение двух секунд до тех пор, пока главный поток не выдаст *сигнал*:

```
var signal = new ManualResetEvent(false);
new Thread(() =>
{
  Console.WriteLine("Waiting for signal..."); // Ожидание
сигнала...
  signal.WaitOne();
  signal.Dispose();
  Console.WriteLine("Got signal!");
}).Start();
Thread.Sleep(2000);
signal.Set(); // "Открыть" сигнал
```

После вызова метода *Set* сигнал остается “открытым”; для его “закрытия” понадобится вызвать метод *Reset*. Класс *ManualResetEvent* – одна из нескольких сигнализирующих конструкций, предоставляемых средой CLR.

Многопоточность в обогащенных клиентских приложениях

В приложениях WPF, UWP и Windows Forms выполнение длительных по времени операций в главном потоке снижает отзывчивость приложения, потому что главный поток обрабатывает также цикл сообщений, который отвечает за визуализацию и поддержку событий клавиатуры и мыши.

Популярный подход предусматривает настройку “рабочих” потоков для выполнения длительных по времени операций. Код в рабочем потоке запускает длительную операцию и по ее завершении обновляет пользовательский интерфейс. Тем не менее, все обогащенные клиентские приложения поддерживают потоковую модель, в которой элементы управления пользовательского интерфейса могут быть доступны только из создавшего их потока (обычно главного потока пользовательского интерфейса). Нарушение данного правила приводит либо к непредсказуемому поведению, либо к генерации исключения.

Следовательно, когда нужно обновить пользовательский интерфейс из рабочего потока, запрос должен быть перенаправлен потоку пользовательского интерфейса (формально это называется маршализацией). Вот как выглядит низкоуровневый способ реализации такого действия (позже мы обсудим другие решения, которые на нем основаны):

- в приложении WPF вызовите метод *BeginInvoke* или *Invoke* на объекте *Dispatcher* элемента;
- в приложении UWP вызовите метод *RunAsync* или *Invoke* на объекте *Dispatcher*;
- в приложении Windows Forms вызовите метод *BeginInvoke* или *Invoke* на элементе управления.

Все упомянутые методы принимают делегат, ссылающийся на метод, который требуется запустить. Методы *BeginInvoke/RunAsync* работают путем постановки этого делегата в очередь сообщений потока пользовательского интерфейса (та же очередь, которая обрабатывает события, поступающие от клавиатуры, мыши и таймера). Метод *Invoke* делает то же самое, но затем блокируется до тех пор, пока сообщение не будет прочитано и обработано потоком пользовательского интерфейса. По указанной причине метод Invoke позволяет получить возвращаемое значение из метода. Если возвращаемое значение не требуется, то методы *BeginInvoke/RunAsync* предпочтительнее из-за того, что они не блокируют вызывающий компонент и не привносят возможность возникновения взаимоблокировки.

❖ Вы можете представлять себе, что при вызове метода Application.Run выполняется следующий псевдокод:

```
while (приложение не завершено)
{
  Ожидать появления чего-нибудь в очереди сообщений
  Что-то получено: к какому виду сообщений оно относится?
    Сообщение клавиатуры/мыши -> запустить обработчик событий
    Пользовательское сообщение BeginInvoke -> выполнить делегат
    Пользовательское сообщение Invoke -> выполнить делегат и отправить
результат
}
```

Цикл такого вида позволяет рабочему потоку подготовить делегат для выполнения в потоке пользовательского интерфейса.

В целях демонстрации предположим, что имеется окно WPF с текстовым полем по имени *txtMessage*, содержимое которого должно быть обновлено рабочим потоком после выполнения длительной задачи (эмулируемой с помощью вызова метода *Thread.Sleep*).

Ниже приведен необходимый код:

```
partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread(Work).Start();
  }
  void Work()
  {
    Thread.Sleep(5000); // Эмулировать длительно выполняющуюся задачу
    UpdateMessage("The answer");
  }

  void UpdateMessage(string message)
  {
    Action = () => txtMessage.Text = message;
    Dispatcher.BeginInvoke(action);
  }
}
```

После запуска показанного кода немедленно появляется окно. Спустя пять секунд текстовое поле обновляется. Для случая Windows Forms код будет похож, но только в нем вызывается метод *BeginInvoke* объекта *Form*:

```
void UpdateMessage(string message)
{
  Action = () => txtMessage.Text = message;
  this.BeginInvoke(action);
}
```

Допускается иметь множество потоков пользовательского интерфейса, если каждый из них владеет своим окном. Основным сценарием может служить приложение с несколькими высокоуровневыми окнами, которое часто называют приложением с *однодокументным интерфейсом* (Single Document Interface – SDI),

например, Microsoft Word. Каждое окно SDI обычно отображает себя как отдельное “приложение” в панели задач и по большей части оно функционально изолировано от других окон SDI. За счет предоставления каждому такому окну собственного потока пользовательского интерфейса окна становятся более отзывчивыми.

Контексты синхронизации

В пространстве имен *System.ComponentModel* определен абстрактный класс *SynchronizationContext*, который делает возможным обобщение маршализации потоков.

В обогащенных API-интерфейсах для мобильных и настольных приложений (UWP, WPF и Windows Forms) определены и созданы экземпляры подклассов *SynchronizationContext*, которые можно получить через статическое свойство *SynchronizationContext.Current* (при выполнении в потоке пользовательского интерфейса). Захват этого свойства позволяет позже “отправлять” сообщения элементам управления пользовательского интерфейса из рабочего потока:

```
partial class MyWindow : Window
{
  SynchronizationContext _uiSyncContext;
  public MyWindow()
  {
    InitializeComponent();
    // Захватить контекст синхронизации для текущего потока пользовательского
интерфейса:
    _uiSyncContext = SynchronizationContext.Current;
    new Thread(Work).Start();
  }
  void Work()
  {
    Thread.Sleep(5000); // Эмулировать длительно выполняющуюся задачу
    UpdateMessage("The answer");
  }
  void UpdateMessage(string message)
  {
    // Маршализировать делегат потоку пользовательского интерфейса:
    _uiSyncContext.Post(_ => txtMessage.Text = message, null);
  }
}
```

Удобство заключается в том, что один и тот же подход работает со всеми обогащенными API-интерфейсами.

Вызов метода *Post* эквивалентен вызову *BeginInvoke* на объекте *Dispatcher* или *Control*; есть также метод *Send*, который является эквивалентом *Invoke*.

Пул потоков

Всякий раз, когда запускается поток, несколько сотен микросекунд тратится на организацию таких элементов, как новый стек локальных переменных. Снизить эти накладные расходы позволяет *пул потоков*, предлагая накопитель заранее созданных многократно применяемых потоков. Организация пула потоков жизненно важна для эффективного параллельного программирования и реализации мелкомодульного параллелизма; пул потоков позволяет запускать короткие операции без накладных расходов, связанных с начальной настройкой потока.

При использовании потоков из пула следует учитывать несколько моментов:

- Невозможность установки свойства *Name* потока из пула затрудняет отладку (хотя при отладке в окне Threads среды Visual Studio к потоку можно присоединять описание).
- Потоки из пула всегда являются *фоновыми*.
- Блокирование потоков из пула может привести к снижению производительности (см. раздел “Чистота пула потоков” далее).

Приоритет потока из пула можно свободно изменять – когда поток возвратится обратно в пул, будет восстановлен его первоначальный приоритет.

Для выяснения, является ли текущий поток потоком из пула, предназначено свойство *Thread.CurrentThread.IsThreadPoolThread*.

Вход в пул потоков

Простейший способ явного запуска какого-то кода в потоке из пула предполагает применение метода *Task.Run*

```
// Класс Task находится в пространство имен System.Threading.Tasks
Task.Run(() => Console.WriteLine("Hello from the thread pool"));
```

Поскольку до выхода версии .NET Framework 4.0 задачи не существовали, общепринятой альтернативой был вызов метода *ThreadPool.QueueUserWorkItem:*

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```

Перечисленные ниже компоненты неявно используют пул потоков:

- серверы приложений ASP.NET Core и Web API;
- классы *System.Timers.Timer* и *System.Threading.Timer*;
- конструкции параллельного программирования;
- (унаследованный) класс *BackgroundWorker*;

Чистота пула потоков

Пул потоков содействует еще одной функции, которая гарантирует то, что временный излишек интенсивной вычислительной работы не приведет к превышению лимита ЦП. Превышение лимита – это условие, при котором активных потоков имеется больше, чем ядер ЦП, и операционная система вынуждена выделять потокам кванты времени. Превышение лимита наносит ущерб производительности, т.к. выделение квантов времени требует интенсивных переключений контекста и может приводить к недействительности кешей ЦП, которые стали очень важными в обеспечении производительности современных процессоров.

Среда CLR избегает превышения лимита в пуле потоков за счет постановки задач в очередь и настройки их запуска. Она начинает с запуска такого количества параллельных задач, которое соответствует числу аппаратных ядер, и затем регулирует уровень параллелизма по алгоритму поиска экстремума, непрерывно

подгоняя рабочую нагрузку в определенном направлении. Если производительность улучшается, тогда среда CLR продолжает двигаться в том же направлении (а иначе – в противоположном). В результате обеспечивается продвижение по оптимальной кривой производительности даже при наличии соперничающих процессов на компьютере.

Стратегия, реализованная в CLR, работает хорошо в случае удовлетворения следующих двух условий :

- элементы работы являются в основном кратковременными (менее 250 мс либо в идеале менее 100 мс), так что CLR имеет много возможностей для измерения и корректировки;
- в пуле не доминируют задания, которые большую часть своего времени являются заблокированными.

Блокирование ненадежно, поскольку дает среде CLR ложное представление о том, что оно загружает ЦП. Среда CLR достаточно интеллектуальна, чтобы обнаружить это и скомпенсировать (за счет внедрения дополнительных потоков в пул), хотя такое действие может сделать пул уязвимым к последующему превышению лимита. Также может быть введена задержка, потому что среда CLR регулирует скорость внедрения новых потоков, особенно на раннем этапе времени жизни приложения (тем более в клиентских ОС, где она отдает предпочтение низкому потреблению ресурсов).

Поддержание чистоты пула потоков особенно важно, когда требуется в полной мере задействовать ЦП.