ПАРАЛЛЕЛИЗМ И АСИНХРОННОСТЬ В C#

Тема: Многопоточная обработка

Гибадуллин Р.Ф.
rfgibadullin@kai.ru

Презентация подготовлена на основе источника:
Албахари Д. C# 9.0. Справочник. Полное описание языка.: Пер. с англ. – СпБ.: ООО «Диалектика», 2021. – С. 627-690. (См. главу 14. Параллелизм и асинхронность.)

Введение в курс «Параллелизм и асинхронность»

Ниже приведены самые распространенные сценарии применения параллелизма.

- **Написание отзывчивых пользовательских интерфейсов** — в приложениях WPF, мобильных и Windows Forms длительные задачи должны запускаться параллельно с кодом UI.
- **Обеспечение одновременной обработки запросов** — клиентские запросы на сервер должны обрабатываться параллельно для масштабируемости. В ASP.NET Core это делается автоматически.
- **Параллельное программирование** — код с интенсивными вычислениями выполняется быстрее на многоядерных компьютерах, если нагрузка распределяется между ядрами.
- **Упреждающее выполнение** — на многоядерных машинах можно улучшить производительность, предсказывая и выполняя действия заранее.
- Общий механизм одновременного выполнения кода называется *многопоточностью*. Многопоточность поддерживается CLR и ОС, являясь фундаментальной концепцией параллелизма. Важно понимать основы многопоточной обработки и влияние потоков на *разделяемое состояние*.

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.

- **Однопоточная программа** — внутри изолированной среды процесса функционирует только один поток, поэтому он получает монопольный доступ к среде.
- **Многопоточная программа** — 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***.

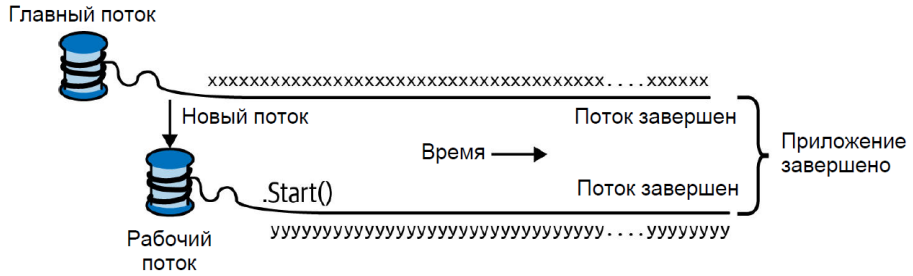
Создание потока

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.

```
// Начать новый поток, выполняющий WriteY()
Thread t = new Thread(WriteY);
t.Start();
// 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"); }
```

Типичный вывод: xxxxxxxyyyyyyyyyyyyyxxx
xxxxxxxxxxyyyyyyyyyxxxxxxxyyyy...



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".

- **На одноядерном процессоре** — ОС выделяет каждому потоку кванты времени (обычно 20 мс. в Windows) для эмуляции параллелизма, что дает повторяющиеся блоки вывода "x" и "y".
- **На многоядерной машине** — два потока могут выполняться по-настоящему параллельно (конкурируя с другими активными процессами).
- Говорят, что поток ***вытесняется*** в точках, где его выполнение пересекается с выполнением кода в другом потоке.
- После запуска свойство **IsAlive** потока возвращает **true** до тех пор, пока не будет достигнута точка завершения. После завершения поток не может быть запущен повторно.
- Каждый поток имеет свойство **Name** для содействия отладке. Установить имя потока можно только один раз.
- Статическое свойство **Thread.CurrentThread** возвращает поток, выполняющийся в текущее время.

Join и Sleep

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

```
Thread t = new Thread(Go);  t.Start();  t.Join();
```

- При вызове **Join** можно указывать тайм-аут в миллисекундах или в виде **TimeSpan**. Метод возвращает **true**, если поток завершен, или **false**, если истекло время тайм-аута.
- Метод **Thread.Sleep** приостанавливает текущий поток на заданный период:

  ```
  Thread.Sleep(TimeSpan.FromHours(1));    Thread.Sleep(500);
  ```

  - **Thread.Sleep(0)** — немедленно прекращает текущий квант времени потока, добровольно передавая контроль над ЦП другим потокам.
  - **Thread.Yield()** — делает то же самое, но уступает контроль только потокам на том же самом процессоре.
- На время ожидания **Sleep** или **Join** поток ***блокируется*** — он немедленно уступает свой квант процессорного времени и далее не потребляет процессорное время, пока удовлетворяется условие блокировки.

Блокирование

Поток считается ***заблокированным***, если его выполнение приостановлено по некоторой причине, такой как вызов метода **Sleep** или ожидание завершения другого потока через **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 свойства **ThreadState**:

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

- Когда поток блокируется или деблокируется, ОС производит *переключение контекста*. С ним связаны небольшие накладные расходы, обычно *одна–две микросекунды*.

ThreadState

Свойство **ThreadState** является перечислением флагов, комбинирующим три «уровня» данных в побитовой манере. Большинство значений избыточны или устаревшие.

- 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.

Интенсивный ввод-вывод или интенсивные вычисления

Операция с ***интенсивным вводом-выводом*** (I/O-bound) — большую часть времени тратит на ожидание. Примеры: загрузка веб-страницы, **Console.ReadLine**, **Thread.Sleep**.

Операция с ***интенсивными вычислениями*** (CPU-bound) — большую часть времени затрачивает на вычисления с привлечением ЦП.

- Операция с интенсивным вводом-выводом работает одним из двух способов:
  - **Синхронно** ожидает завершения операции в текущем потоке (**Console.ReadLine**, **Thread.Sleep**, **Thread.Join**).
  - **Асинхронно** — инициирует обратный вызов, когда операция завершается спустя какое-то время.
- Синхронные I/O-операции большую часть времени тратят на ***блокирование*** потока.

Блокирование или зацикливание

Альтернатива блокированию — ***зацикливание*** (spinning) потока:

```
while (DateTime.Now < nextStartTime);  // зацикливание - неэкономно!
```

- ❖ Это очень неэкономное расходование ЦП: среда CLR и ОС предполагают, что поток выполняет важные вычисления. По сути, код *I/O-bound* превращается в *CPU-boun*d.

Нюансы:

- ➢ **Кратковременное зацикливание** может быть эффективным, когда условие удовлетворяется за микросекунды — избегает накладных расходов переключения контекста. .NET предлагает **SpinLock** и **SpinWait**.
- ➢ **Затраты на блокирование не нулевые**: каждый поток связывает ~1 МБ памяти. Для программ с сотнями параллельных I/O-операций лучше использовать подход на основе обратных вызовов (асинхронные шаблоны).

Локальное или разделяемое состояние

Среда CLR назначает каждому потоку собственный ***стек*** в памяти, так что локальные переменные хранятся отдельно.

```
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 предсказуемо содержит десять знаков вопроса.
- Потоки ***разделяют*** данные, если они имеют общую ссылку на один и тот же экземпляр.

Оба потока разделяют переменную **_done**, поэтому слово “Done” выводится один раз. Однако двукратный вывод возможен (хотя и маловероятен) — это проблема потокобезопасности:

```
bool _done = false;
new Thread (Go).Start();
Go();

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

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();
```

Поля экземпляра — метод вызывается на одном экземпляре из разных потоков:

```
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 — разделяются между всеми потоками в домене приложения:

```
class ThreadTest {
  static bool _done;  // разделяется между всеми потоками
  static void Main() { new Thread (Go).Start(); Go(); }
  static void Go() {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Блокировка и безопасность потоков

Исправить пример можно, получив ***монопольную блокировку*** на период чтения и записи разделяемого поля. Для этого в C# предусмотрен оператор **lock**:

```
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; }
    }
  }
}
```

Оператор **lock** — ***синтаксический сахар*** для **Monitor.Enter** / **Monitor.Exit**:

```
lock (_locker) { ... }

// Эквивалентно:
Monitor.Enter(_locker);
try { ... }
finally { Monitor.Exit(_locker); }
```

- Внутри CLR захват блокировки реализован через атомарную инструкцию **CAS** (Compare-And-Swap, **cmpxchg** на x86). Если CAS не удался — кратковременное зацикливание (spinning), затем эскалация до объекта ядра ОС.
- Когда два потока соперничают за блокировку, один *ожидает*, пока она не станет доступной. Гарантируется, что только один поток может войти в блок кода за раз.
- Код, защищённый таким образом, называется ***безопасным в отношении потоков*** (thread-safe).

Инструкция cmpxchg (Compare and Exchange)

Синтаксис: **lock cmpxchg [dest], src** — атомарная инструкция x86:

```
lock cmpxchg DWORD PTR [dest], src

Алгоритм (атомарно):
1. Сравнить EAX с [dest]
2. Если равны:  [dest] = src,  ZF = 1  (успех)
3. Если нет:    EAX = [dest],  ZF = 0  (неудача)
```

- **ZF** (Zero Flag) — бит в регистре флагов **EFLAGS**. Устанавливается в 1, если результат сравнения равен нулю (значения совпали).
- Префикс **lock** блокирует шину памяти, обеспечивая атомарность на многоядерных процессорах.
- В C# доступна через **Interlocked.CompareExchange**:

```
Interlocked.CompareExchange(ref location, newValue, expected);
// Компилируется в: lock cmpxchg
```

- Применение в **lock**: CLR кладёт в EAX «свободно», а в src — ID потока. CAS успешен → блокировка захвачена. Неудача → spinning → ядро ОС.

Передача данных потоку

Проще всего передать аргументы начальному методу потока с помощью ***лямбда-выражения***:

```
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 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) приём — передача аргумента методу **Start**:

```
Thread t = new Thread (Print);
t.Start ("Hello from t!");
```

При передаче через **Start** метод потока принимает **object**, что требует приведения:

```
void Print (object messageObj)
{
  string message = (string) messageObj; // Приведение типа
  Console.WriteLine (message);
}
```

- Код работает, т.к. конструктор **Thread** перегружен для приёма одного из двух делегатов:

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

- Ограничение **ParameterizedThreadStart**: принимает только **один аргумент** типа **object**, не обеспечивает типобезопасность.

Сравнение подходов:

- **Лямбда-выражение** — гибко, любое количество типизированных аргументов, вся логика встраивается.
- **Thread.Start(object)** — менее гибко, один аргумент типа **object**, требует приведения.

Следует соблюдать осторожность, чтобы случайно не изменить ***захваченные переменные*** после запуска потока:

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

- Вывод будет **недетерминированным**! Типичный результат: **0223557799**
- Проблема: на протяжении всего цикла переменная **i** ссылается на *ту же самую* ячейку памяти. Каждый поток вызывает **Console.Write** с переменной, значение которой может измениться!
- Решение — применение ***временной переменной***:

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

- Теперь все цифры от 0 до 9 выводятся **в точности по одному разу**. Переменная **temp** локальна для каждой итерации, поэтому каждый поток захватывает отличающуюся ячейку памяти.

Проблему с захваченными переменными проще проиллюстрировать так:

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

- Оба лямбда-выражения захватывают ***одну и ту же*** переменную **text**. К моменту вызова **Start()** значение уже изменено на **"t2"**, поэтому строка "t2" выводится **дважды**.

Ключевые выводы:

- Лямбда-выражение — наиболее удобный и мощный способ передачи данных потоку.
- Не изменяйте захваченные переменные после запуска потока!
- Используйте **локальную копию** (**int temp = i**) для безопасного захвата в циклах.

Обработка исключений

Блоки **try/catch/finally**, действующие во время создания потока, ***не играют никакой роли*** в потоке, когда он начинает выполнение:

```
try {
  new Thread (Go).Start();
}
catch (Exception ex) {
  // Сюда мы никогда не попадём!
}
void Go() { throw null; } // NullReferenceException
```

- Причина: каждый поток имеет **собственный стек вызовов**. Механизм раскрутки исключений (*exception unwinding*) ищет подходящий **catch** только вверх по текущему стеку — он не может перепрыгнуть в стек другого потока:

Обработчик необходимо переместить ***внутрь метода потока***:

```
new Thread (Go).Start();
void Go() {
  try {
    ...
    throw null; // NullReferenceException перехвачено ниже
    ...
  }
  catch (Exception ex) {
    // Зарегистрировать исключение в журнале
    // и/или сигнализировать другому потоку
  }
}
```

- В производственных приложениях необходимо предусмотреть обработчики исключений для **всех методов входа в потоки** — в точности как в главном потоке.
- Необработанное исключение приведёт к **прекращению работы всего приложения**!
- Обычно в **catch** регистрируют исключение в журнале и/или перезапускают приложение.

Централизованная обработка исключений

В приложениях WPF, UWP и Windows Forms можно подписаться на «глобальные» события обработки исключений:

```
// WPF:
Application.DispatcherUnhandledException += ...;

// Windows Forms:
Application.ThreadException += ...;
```

- Эти события инициируются после возникновения необработанного исключения в любой части программы, вызванной в ***цикле сообщений*** (весь код в главном потоке, пока активен **Application**).
- Полезны как **резервное средство** для регистрации и сообщения об ошибках.
- Ограничение: **неприменимы** для необработанных исключений в созданных вами потоках, не относящихся к UI.
- Обработка этих событий предотвращает аварийное завершение, но впоследствии может потребоваться **перезапуск приложения** во избежание разрушения состояния.