УДК 004.051

А.В. Зотов, А.А. Бугаенко, С.В. Теплинский, Ю.В. Потапов
Донецкий национальный технический университет г. Донецк
Кафедра компьютерной инженерии
E-mail: andrey.bugaenko.1992@gmail.com
lexas_zotov@mail.ru
urapotap@gmail.com

ПОТОКИ

Аннотация

Зотов А.В., Бугаенко А.АПотапов Ю.В., Теплинский С.В. Потоки. *В докладе рассмотрены вопросы программирования задач с использованием потоков и влияние потоков на быстродействие программы.*

Ключевые слова: *многопоточность, интерфейс, поток, пространство имен, задержка.*

Введение. Для применения многопоточности существует несколько причин. Предположим, в приложении предпринимается обращение к какому-то серверу в сети, которое может занять определенное время. Вряд ли захочется, чтобы пользовательский интерфейс из-за этого блокировался, и пользователю пришлось просто дожидаться момента, когда от сервера вернется ответ. Пользователь может выполнять в это время какие-то другие действия или вообще отменить отправленный серверу запрос. В таких ситуациях применение многопоточности приносит пользу.

Для всех видов активности, требующих ожидания, например, из-за получения доступа к файлу, базе данных или сети, может запускаться новый поток, позволяющий выполнять в это же время другие задачи.

Многопоточность может помочь, даже если есть одни только насыщенные в плане обработки задачи. Многочисленные потоки одного и того же процесса могут одновременно выполняться разными ЦП или, что чаще встречается в наши дни, разными ядрами одного многоядерного ЦП.

Разумеется, необходимо знать особенности одновременного выполнения множества потоков. Из-за того, что они выполняются в одно и то же время, при получении ими доступа к одним и тем же данным могут возникать проблемы. Чтобы этого не происходило, должны быть реализованы механизмы синхронизации.

Управление многопоточностью осуществляет планировщик потоков. Эту функцию исполнительная среда CLR (при использовании многопоточности в приложениях.NET) 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 (по материалам [2,4]).

На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков. В Windows типичное значение кванта времени – десятки миллисекунд – выбрано как намного большее, чем затраты CPU на переключение контекста между потоками (несколько микросекунд).

На многопроцессорных компьютерах многопоточность реализована как смесь квантования времени и подлинного параллелизма, когда разные потоки выполняют код на разных CPU. Необходимость квантования времени все равно остается, так как операционная система должна обслуживать как свои собственные потоки, так и потоки других приложений (по материалам [2]).

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.

Исследование работы потоков. Для исследования механики работы потоков и построения приложения использовалась платформа .NET и язык C#. Программа на C# запускается как единственный поток, автоматически создаваемый CLR и операционной системой («главный» поток), и становится многопоточной при помощи создания дополнительных потоков. Эти потоки часто называются рабочими потоками.

Поддержка многопоточности в .NET реализована в пространстве имен System.Threading (по материалам [3]), которое содержит классы и интерфейсы, которые дают возможность программировать в многопоточном режиме.

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

Многопоточность на основе потоков позволяет ускорить работу программы за счет минимизации задержек, таких как задержки ввода/вывода, обращение к внешним устройствам и т.д.

Приведем пример программы обработки массива с использованием потоков. Описание класса потока приведено на рис. 1.

```
1 publicintcount;
2 publicThreadThrd;
3 publicMyThread(stringname)
4   {
5     count = 0;
6     Thrd = newThread(this.Run);
7     Thrd.Name = name;
8     Thrd.Start();
9   }
```

Рисунок 1 – Описание класса потока

Листинг программы, использующей потоки, приведен на рис. 2.

```
1  if (Thrd.Name == "Поток #1")
2  {
3    for (int i = 0; i < 10; i++)
4      for (int j = 0; j < 1000000; j++)
5      {
6        arr[i, j] = count + j;
7        if ((i == 7)&&(j==7000))
8        {
9          Console.WriteLine("Задержка ожидания ввода с       клавиатуры в 1 потоке\n");
10       }
11     }
12   count++;
13   for (int i = 0; i < 10; i++)
14   for (int j = 1000000; j <=2 ; j--)
15   {
16     if (arr[i, j] >arr[i, j - 1])
17     {
18       buff = arr[i, j - 1];
19       arr[i, j - 1] = arr[i, j];
20       arr[i, j] = buff;
21     }
22   }
23 }
```

Рисунок 2 – Листинг программы с использованием потоков

Во время выполнения программы потока, где обрабатывается массив большого размера, основная программа ожидает ввод с клавиатуры. Время выполнения программы представлено в табл. 1.

Листинг программы, обрабатывающей массив большого размера, без использования потоков приведен на рис. 3.

```
31   for (int i = 0; i < 10; i++)
32   for (int j = 1000000; j <=2 ; j--)
33   {
34     if (arr[i, j] >arr[i, j - 1])
35     {
36       buff = arr[i, j - 1];
37       arr[i, j - 1] = arr[i, j];
38       arr[i, j] = buff;
39     }
40   }
```

Рисунок 3 – Листинг программы без обработки потоков

Таблица 1 - Результаты работы программы

	Начало работы	Конец работы
С использованием потоков	17:30:02	17:30:05
Без использования потоков	17:35:02	17:35:07

Из результатов работы видно, что задержка оказывает большое влияние на работу программы. В тоже время при использовании потоков программа продолжает работу, несмотря на ожидание ввода.

Сетевой чат. Одним из примеров применения потоков может быть программа, реализующая сетевой чат. Ее основой являются описанные в пространстве имен System.Net объекты TcpListener, реализующий роль сервера, и TcpClient, который реализует подключение к серверу и передачу информации по локальной сети (по материалам [1]). В языке C# эти объекты реализованы отдельными потоками. Таким образом, во время, пока передается или ожидается сообщение, пользователь имеет полный доступ к интерфейсу.

Оба класса имеют метод Start(), который выполняет создание потока и запуск в нем объекта TcpListener или TcpClient соответственно. Создание сервера реализует код, представленный на рис. 4.

```
1 server = new TcpListener(IPAddress.Any,port);
2 server.Start();
```

Рисунок 4 – Код программы создания сервера

При создании клиента нужно дополнительно создать поток сетевого ввода-вывода (рис. 5).

```
6 tcpClient.Connect(w.IP, port);
7 ns = tcpClient.GetStream();
```

Рисунок 5 – Код программы создания клиента

Закрытие клиента программы производится с помощью кода, приведенного на рисунке 6.

```
11  Thread.Sleep(1000);
12  ns.Close();
13  if (tcpClient != null)
14  {
15      tcpClient.Close();
16  }
17
```

Рисунок 6 - Код программы закрытия клиента

Сначала задается время ожидания для потока клиента, превышающее таймаут, установленный на сервере. Вследствие этого сервер отключает клиента, и его можно закрывать. Закрытие происходит в два этапа: сначала закрывается поток сетевого ввода-вывода, а затем – сам поток клиента.

Пример закрытия сервера показан на рис. 7.

```
19  if (count > 0)
20   {
21      SendToClients("Сервер отключился!", 512);
22   }
23  if (server != null)
24   {
25      server.Stop();
26      server = null;
27      stop = true;
28      for (inti = 0; i< MAXCLIENTS; i++)
29      {
30          if (clients[i] != null)
31          {
32              clients[i].Close();
33          }
34      }
35  }
```

Рисунок 7 – Код программы закрытия сервера

В первую очередь, сервер уведомляет всех подключенных к нему клиентов о завершении своей работы. После этого завершается поток сервера, а затем завершаются потоки сетевого ввода-вывода, соответствующие подключенным клиентам.

Таким образом, на стороне сервера, приложение имеет поток сервера интерфейса и по одному потоку сетевого ввода-вывода на каждого подключенного клиента. На стороне клиента приложение имеет поток интерфейса, поток клиента и поток сетевого ввода-вывода.

Вывод. Авторами была разработана программа «Сетевой чат» с использованием потоков для создания сервера и клиента, приема и передачи сообщений, как клиенту, так и серверу, подключения и отключения, как клиентов, так и сервера.

Таким образом, из проведенных практических исследований многопоточности можно сделать следующие выводы:

- использование нескольких потоков в одной задаче позволяет ограниченно увеличить быстродействие, так как потоки могут выполняться на нескольких вычислительных ядрах процессора одновременно. На компьютерах с одним вычислительным ядром многопоточность не дает выигрыша во времени, а наоборот, увеличивает время выполнения задачи за счет постоянного переключения между потоками;

- использование отдельных потоков для интерфейса и вычислительной или какой-либо другой задачи позволяет обеспечить стабильную отзывчивость интерфейса и предотвратить его «зависание» при выполнении длительных задач или ожидании, а также обеспечить возможность отмены назначенного задания во время его выполнения либо отображения получаемых результатов в процессе выполнения задачи.

Список литературы

1. Albahari J. C# 3.0 in a Nutshell, 3rd Edition A Desktop Quick Reference / B. Albahari. - O'Reilly Media, 2007. – 864 c.

2. Финогенов К. Г. Win32. Основы программирования. - 2-е изд., испр. и дополн / К. Г. Финогенов. - М.: ДИАЛОГ- МИФИ, 2006. - 416 с.

3. Microsoft Delevoper Network. – Mode of access: http://msdn.microsoft.com

4. Ерохин А. C# и платформа .NET. – Режим доступа: http://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php