ЛАБОРАТОРНАЯ РАБОТА 9

Тема: «Разработка, отладка и испытание многопоточного приложения с синхронизированными потоками»

Цель: Сформировать умения и навыки организации многопоточной обработки на основе класса Thread с синхронизированными потоками.

Время выполнения: 4 часа.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Многопоточное программирование позволяет разделить представление и обработку информации на несколько «легковесных» процессов (light-weight processes), имеющих общий доступ как к методам различных объектов приложения, так и к их полям. Многопоточность незаменима в тех случаях, когда графический интерфейс должен реагировать на действия пользователя при выполнении определенной обработки информации. Потоки могут взаимодействовать друг с другом через основной «родительский» поток, из которого они стартованы.

В качестве примера можно привести некоторый поток, отвечающий за представление информации в интерфейсе, который ожидает завершения работы другого потока, загружающего файл, и одновременно отображает некоторую анимацию или обновляет прогресс-бар. Кроме того, этот поток может остановить загружающий файл поток при нажатии кнопки «Отмена».

Когда использовать потоки

Типовое приложение с многопоточностью выполняет длительные вычисления в фоновом режиме. Главный поток продолжает выполнение, в то время как рабочий поток выполняет фоновую задачу. В приложениях Windows Forms, когда главный поток занят длительными вычислениями, он не может обрабатывать сообщения клавиатуры и мыши, и приложение перестает откликаться. По этой причине следует запускать отнимающие много времени задачи в рабочем потоке, даже если главный поток в это время демонстрирует пользователю модальный диалог с надписью *“Работаю... Пожалуйста, ждите”*, так как программа не может перейти к следующей операции, пока не закончена текущая. Такое решение гарантирует, что приложение не будет помечено операционной системой как “Не отвечающее”, соблазняя пользователя с горя прикончить процесс. Опять же, в этом случае модальный диалог может предоставить кнопку “Отмена”, так как форма продолжает получать сообщения, пока задача выполняется в фоновом потоке. Класс **BackgroundWorker** наверняка пригодится вам при реализации такой модели.

В случае приложений без UI, например, служб Windows, многопоточность имеет смысл, если выполняемая задача может занять много

времени, так как требуется ожидание ответа от другого компьютера (сервера приложений, сервера баз данных или клиента). Запуск такой задачи в отдельном рабочем потоке означает, что главный поток немедленно освобождается для других задач.

Другое применение многопоточность находит в методах, выполняющих интенсивные вычисления. Такие методы могут выполняться быстрее на многопроцессорных компьютерах, если рабочая нагрузка разнесена по нескольким потокам (количество процессоров можно получить через свойство **Environment.ProcessorCount**).

C#-приложение можно сделать многопоточным двумя способами: либо явно создавая дополнительные потоки и управляя ими, либо используя возможности неявного создания потоков .NET Framework – **BackgroundWorker,** пул потоков, потоковый таймер, Remoting-сервер, Web-службы или приложение ASP.NET. В двух последних случаях альтернативы многопоточности не существует. Однопоточный web-сервер не просто плох, он попросту невозможен! К счастью, в случае серверов приложений, не хранящих состояние (stateless), многопоточность реализуется обычно довольно просто, сложности возможны разве что в синхронизации доступа к данным в статических переменных.

Когда потоки не нужны

Многопоточность наряду с достоинствами имеет и свои недостатки. Самое главный из них – значительное увеличение сложности программ. Сложность увеличивают не дополнительные потоки сами по себе, а необходимость организации их взаимодействия. От того, насколько это взаимодействие является преднамеренным, зависит продолжительность цикла разработки, а также количество спорадически проявляющихся и трудноуловимых ошибок в программе. Таким образом, нужно либо поддерживать дизайн взаимодействия потоков простым, либо не использовать многопоточность вообще, если только вы не имеете противоестественной склонности к переписыванию и отладке кода.

Кроме того, чрезмерное использование многопоточности отнимает ресурсы и время CPU на создание потоков и переключение между потоками. В частности, когда используются операции чтения/записи на диск, более быстрым может оказаться последовательное выполнение задач в одном или двух потоках, чем одновременное их выполнение в нескольких потоках. Далее будет описана реализация ***очереди Поставщик/Потребитель***, предоставляющей такую функциональность.

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

```
class Program
{
   static int x=0;
   static void Main(string[] args)
   {
```

```
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
    public static void Count()
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            x++;
Thread.Sleep(100);
        }
    }
}
```

Здесь у нас запускаются пять потоков, которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово lock. Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

```
class Program
{
    static int x=0;
    static object locker = new object();
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
    public static void Count()
    {
        lock (locker)
```

```
        {
            x = 1;
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
                x++;
                Thread.Sleep(100);
            }
        }
    }
}
```

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

Обработка исключений

Обрамление кода создания и запуска потока блоками try/catch/finally имеет мало смысла. Посмотрите следующий пример:

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

  static void Go() { throw null; }
}
```

try/catch здесь фактически совершенно бесполезны, и **NullReferenceException** во вновь созданном потоке обработано не будет. Вы поймете почему, если вспомните, что поток имеет свой независимый путь исполнения. Решение состоит в добавлении обработки исключений непосредственно в метод потока:

```
public static void Main()
{
  new Thread(Go).Start();
}

static void Go()
{
  try
  {
```

```
    ...
    throw null;      // это исключение будет поймано ниже
    ...
  }
  catch(Exception ex)
  {
    Логирование исключения и/или сигнал другим потокам
    ...
  }
}
```

Начиная с .NET 2.0, необработанное исключение в любом потоке приводит к закрытию всего приложения, а значит игнорирование исключений – это не наш метод. Следовательно, блок **try/catch** необходим в каждом методе потока – по крайней мере, в приложениях не для собственного употребления – чтобы избежать закрытия приложения из-за необработанного исключения. Это может быть довольно обременительно, особенно для программистов Windows Forms, которые используют глобальный перехватчик исключений, как показано ниже:

```
using System;
using System.Threading;
using System.Windows.Forms;

static class Program
{
  static void Main()
  {
    Application.ThreadException += HandleError;
    Application.Run(new MainForm());
  }

  static void HandleError(object sender, ThreadExceptionEventArgs e)
  {
    Логирование исключения, завершение или продолжение работы
  }
}
```

Событие **Application.ThreadException** возникает, когда исключение генерируется в коде, который был вызван (возможно, по цепочке) из обработчика сообщения Windows (например, от клавиатуры, мыши и т.д.) – короче говоря, практически из любого кода приложения Windows Forms. Поскольку это замечательно работает, появляется чувство ложной безопасности, - что все исключения будут обработаны этим центральным обработчиком. Исключения, возникающие в рабочих потоках – хороший пример исключений, которые не ловятся в **Application.ThreadException** (код в методе **Main** – другой такой пример, включая конструктор **MainForm**, отрабатывающий до запуска цикла обработки сообщений).

.NET Framework предоставляет низкоуровневое событие для глобальной обработки исключений - **AppDomain.UnhandledException**. Это событие происходит, когда есть необработанное исключение в любом потоке и для любых типов приложений (с пользовательским интерфейсом или без него). Однако, хотя это и хороший способ регистрации необработанных исключений, он не предоставляет никакого способа предотвратить закрытие приложения или подавить сообщение .NET о необработанном исключении.

Резюме

Многопоточность позволяет приложениям разделять задачи и решать независимо каждую из них, максимально эффективно используя процессорное время. Однако многопоточность является верным выбором не всегда и порой может замедлить работу приложения. Создание и управление потоками на C# осуществляется посредством класса *System. Threading. Thread.* Безопасность потоков является важным понятием, связанным с созданием и использованием потоков. Безопасность потоков означает, что члены объекта всегда поддерживаются в действительном состоянии при одновременном использовании несколькими потоками. Важно, чтобы наряду с изучением синтаксиса многопоточности вы также поняли, когда ее применять, а именно: для повышения параллелизма, упрощения структуры и оптимального использования процессорного времени.

Индивидуальные задания для лабораторной работы 1:

1. Создайте консольное приложение, которое запускает два потока. Один поток должен выводить числа от 1 до 10 с задержкой в 1 секунду между выводами, а другой поток должен выводить буквы от ‘A’ до ‘J’ с такой же задержкой.
2. Создайте программу, в которой два потока одновременно увеличивают значение одной и той же переменной. Используйте механизм синхронизации для предотвращения гонок данных.
3. Напишите программу, которая создает несколько потоков, каждый из которых выводит на экран свое имя и идентификатор потока.
4. Напишите программу, которая создает и запускает несколько потоков с использованием класса Thread. Каждый поток должен выполнять свою задачу (например, вычисление факториала числа).
5. Создайте программу, которая одновременно читает данные из одного файла и записывает их в другой файл, используя несколько потоков
6. Напишите программу, которая использует таймер для выполнения периодических задач в отдельном потоке.
7. Напишите программу, которая демонстрирует обработку исключений в многопоточном приложении. Например, можно создать несколько потоков, каждый из которых может выбросить исключение, и реализовать механизм их обработки.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам, представленные в теоретических сведениях, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.