12 Дәріс Тізбектерді автоматты синхрондау

12.1 Тізбектерді синхрондау контекстінің түсінігі

Қандай да бір үдерістің тізбектерін синхрондау компьютердің максималды жүктелуін қамтамасыз етуге бағытталған. Көп жағдайда тізбектердің жұмысын синхрондау тізбектердің параллельді жұмысын ұйымдастырумен теңестіріледі. Бұл олай емес, есептеу үрдісін ұйымдастырудың кейбір ортақ мезеттері болса да, мысалы, тізбектерді ұйымдастыру. Тізбектер бір деректерге жүгінгенде, мысалы, деректер базасымен жұмыс істегенде, тізбектердің синхрондауы айрықша маңызды болады. Тізбектердің параллельді жұмысы, мысалы, тізбектері мыңдаған сұранысты өңдейтін веб-сервердің жұмысы, ортақ деректерді қолдануды көздемейді.

Егерде объектіні құрса, ал ол қандай да бір деректермен әрекеттері бар жады облысы (объект контексі), және оған (жады облысына) деген қатынауды блоктаса, онда кезекті тізбектің жұмысы аяқталғанға дейін тізбектерді автоматты синхрондауды (нақты айтқанда блоктауын) ұйымдастаруға болады

C# тілінде ContextBoundObject арнайы класы бар, оны мұраға алсан, әдістер мен қасиеттердің белгілі жиынтығы бар объектілердің контекстерін құруға болады. Объектінің контекстіне қатынауды автоматты түрде синхрондау (блоктау) үшін осындай объектіні жариялаудың алдында Synchronization атрибутын көрсету қажет, сонда осындай объектінің әрекет ету облысы синхрондау контексті деп аталады.Мысалы:

```
[Synchronization]
 public class Niti : ContextBoundObject
        { . . .}
```

Осы объектінің әрекет ету облысы синхрондау контексті болып табылады. Синхрондау контекстін құрып, осы объектінің ресурстарына бірнеше жұмыс істеп тұрған тізбектің қатынауын (блоктауын) басқаруға болады- сонымен үдерісте бірнеше тізбек жұмыс істеуі мүмкін, бірақ уақыттың нақты мезетінде объектінің ресурстарына тек бір тізбекке рұқсат етілген. Сонда, синхрондау контекстімен жұмыс істейтін тізбек жұмысын аяқтағанға дейін, басқа тізбек қатынауға рұқсат алмайды. Бұл жайт бір мезетте бъектімен бірнеше жұмыс істеп тұрған тізбектің жұмыс жасауына жол бермейді.

12.2 Бірнеше тізбекпен синхрондау контекстін пайдалану мысалы

Оқу мысалы ретінде for циклының басқарушы айнымалысының мәндердің консолдық терезеге шығаруға мүмкіндік беретін әдіспен бір мезетте төрт тізбектің жұмыс істеуін қарастырамыз.

Бағдарлама коды:

```
using System;
using System.Threading;
using System.Runtime.Remoting.Contexts;

namespace ConsoleApplication1
{
[Synchronization]
```

```
 public class Niti : ContextBoundObject
 {
  public void Cikl(object ob)
  {
   int k = (int)ob;
   Console.Write("Работает нить {0}  ",k);
   for (int i = k * 10; i <= k * 10 + 9; i++)
   {
    Console.Write("  " + i);
    Thread.Sleep(100);
   }
   Console.WriteLine();
  }
 }
 class Program
 {
  static void Main(string[] args)
  {
   int i=0;
   Niti ob_1 = new Niti();
   i++;
   Console.WriteLine("Запущена нить {0}  ", i);
   Thread th1 = new Thread(new
ParameterizedThreadStart(ob_1.Cikl));
   th1.Start(i);
   i++;
   Console.WriteLine("Запущена нить {0}  ", i);
   Thread th2 = new Thread(new
ParameterizedThreadStart(ob_1.Cikl));
   th2.Start(i);
   i++;
   Console.WriteLine("Запущена нить {0}  ", i);
   Thread th3 = new Thread(new
ParameterizedThreadStart(ob_1.Cikl));
   th3.Start(i);
   i++;
   Console.WriteLine("Запущена нить {0}  ", i);
   ob_1.Cikl(i);
   Console.ReadLine();
  }
 }
}
```

Работа программы:

```
Запущена нить 1
Запущена нить 2
Запущена нить 3
Запущена нить 4
Работает нить 2    20  21  22  23  24  25  26  27  28  29
Работает нить 1    10  11  12  13  14  15  16  17  18  19
Работает нить 4    40  41  42  43  44  45  46  47  48  49
Работает нить 3    30  31  32  33  34  35  36  37  38  39
```

Нәтижеден біздің үдерісімізде барлық төрт тізбекте іске қосылғаны байқалады, бірақ синхрондау контекстін қандай да бір тізбек басып алса, онда

басқа тізбектің қатынауы алдыңғы тізбектің бүкіл жұмысы аяқталған соң ғана басқа тізбекке ұсынылатын болады.

Автоматты түрдегі синхрондау **ContextBoundObject** мұрагері болмайтын (мысалы, WindowsForm) статикалық типтер мен кластардың мүшелерінің қорғанысы үшін қолданылуы мүмкін емес.

12.3 Өзара блоктау жөніндегі түсінік

Синхрондау контексті дара объектінің шегінен шығып таралуы мүмкін. Бастапқыда, егер синхрондалған объект басқа объектінің кодында құрылса, екеуі де бір контекстіні бөліседі (басқаша айтқанда, бір үлкен блоктауды!) Осы тәртіпті **Synchronization** атрибуты конструкторында **SynchronizationAttribute** класында анықталған константаларды қолдана отырып, жалаушаларды қойып, өзгертуге болады:

Константа	Мәні
NOT_SUPPORTED	Синхрондау атрибуттарын пайдаланбауға эквивалентті Эквивалентно не использованию атрибутов синхронизации.
SUPPORTED	Егерде құрастыру синхрондалған объектіден болса, қолданыстағы контекстіге қосылады, әйтпесе синхрондауды қолданбау керек Присоединиться к существующему контексту синхронизации, если создание происходит из синхронизированного объекта, иначе не использовать синхронизацию.
REQUIRED(default)	Егерде құрастыру синхрондалған объектіден болса, қолданыстағы контекстіге қосылу, әйтпесе жаңа контекст құру керек Присоединиться к существующему контексту синхронизации, если создание происходит из синхронизированного объекта, иначе создать новый контекст.
REQUIRES_NEW	Әрдайым синхрондаудың жаңа контекст құру Всегда создавать новый контекст синхронизации.

Сонымен, егер **SynchronizedA** класының объектісі **SynchronizedB** класының объектісін құрса, егер **SynchronizedB** төмендегідей декларацияланған болса, онда оларда синхрондау контекстері әртүрлі болады:

```
[Synchronization(SynchronizationAttribute.REQUIRES_NEW)]
public class SynchronizedB : ContextBoundObject
{ . . . }
```

Синхрондау контексті көбірек кеңейген сайын, басқару жеңілірек болады, бірақ пайдалы параллелизм үшін мүмкіндіктер азая түсетін болады. Екінші тараптан алсақ, синхрондаудың кейбір контекстері өзара блоктауға ұшырайды. Міненки, мысал:

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace ConsoleApplication1
{
 [Synchronization]
 public class Niti : ContextBoundObject
 {
  public Niti ob_ni;
  public void Zapl(object ob)
  {
   int k = (int)ob;
   Console.WriteLine("Работает нить {0}  ", k);
   ob_ni.Cikl1(k);
  }
  void Cikl1(int k)
  {
   for (int i = k * 10; i <= k * 10 + 9; i++)
   {
    Console.Write("  " + i);
    Thread.Sleep(100);
   }
   Console.WriteLine();
  }
 }

 class Program
 {
  static void Main()
  {
   int i=1;
   Niti Ni_1 = new Niti();
   Niti Ni_2 = new Niti();
   Ni_1.ob_ni = Ni_2;
   Ni_2.ob_ni = Ni_1;
   Thread th1 = new Thread(new
ParameterizedThreadStart(Ni_1.Zapl));
   th1.Start(i);
   i++;
  Thread th2 = new Thread(new ParameterizedThreadStart(Ni_2.Zapl));
   th2.Start(i);
   Console.ReadLine();
  }
 }
}
```

Работа программы:

Работает нить 1

Работает нить 2

Niti-дің әр данаса Program деген синхрондалмаған кластың ішінде құрылатындықтан, әр дана өз синхрондау контекстіне ие болады, демек, өз блоктауына да ие. Екі объект бір-бірінің әдістерін шақырғанда,бірден өзара блоктау орын алады. Контекстінің айқын блоктаудан айырмашылығы осында, оларда өзара блоктаулар бұдан гөрі айқынырақ.

Егерде `Cikl1` әдісінің кодын `Zapl` әдісінің ішіне орнатса, онда өзара блоктау мүлдем жоқ, мысалы:

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace ConsoleApplication1
{
 [Synchronization]
 public class Niti : ContextBoundObject
 {
  public Niti ob_ni;
  public void Zapl(object ob)
  {
   int k = (int)ob;
   Console.WriteLine("Работает нить {0}  ", k);
   for (int i = k * 10; i <= k * 10 + 9; i++)
   {
    Console.Write("  " + i);
    Thread.Sleep(100);
   }
   Console.WriteLine();
  }
 }
 class Program
 {
  static void Main()
  {
   int i=1;
   Niti Ni_1 = new Niti();
   Niti Ni_2 = new Niti();
   Ni_1.ob_ni = Ni_2;
   Ni_2.ob_ni = Ni_1;
   Thread th1 = new Thread(new
ParameterizedThreadStart(Ni_1.Zapl));
   th1.Start(i);
   i++;
   Thread th2 = new Thread(new
ParameterizedThreadStart(Ni_2.Zapl));
   th2.Start(i);
   Console.ReadLine();
  }
 }
}
```

Работа программы:

Работает нить 1

10Работает нить 2

20 11 21 12 22 23 13 24 14 15 25 26 16 17 27 18 28 29 19

12.4 EventWaitHandle сигналдық конструкция

Windows жүйесінің синхрондау объектілерін құруға арналған Win32 API функцияларының жиынтығында, мьютекстер және семафорлармен қатар, оқиғаны (event) синхрондау объектілерін құруға арналған функциялар қолданылады. Осы синхрондау объектісінің тізбектермен жұмыс істеуі қарапайым және ыңғайлы. Ол бір тізбекті екінші тізбектің бір үдерісте болсын, әртүрлі үдерістерде болсын, қандай да бір әрекет жасағаны жайлы хабардар ету үшін пайдаланылады. Әрине, .NETFramework-та тізбектерді басқару (синхрондау) үшін осы объектінің артықшылығын пайдаланбай қоймайды. Сондықтанда жіктеу бойынша **Mutex** және **Semaphore** кейін сигналдық блоктаушы конструкциялардың келесі класы **EventWaitHandle** класының негізіндегі сигналдық конструкция болады. Сигналдық конструкция бір немесе бірнеше үдерістердің басқа тізбегінен сигналдың тосуды ұйымдастаруға мүмкіндік беретіндік береді.

EventWaitHandle класының екі туынды класы **AutoResetEvent** және **ManualResetEvent** бар (бұл бұрында қарастырылған C# оқиғалары мен делегатарына ешқандай қатысы жоқ блоктаушы сигналдық конструкция екендігін ескертеміз). Екі класқа да базалық кластың барлық функционалдық мүкіндіктері қол жетімді, жалғыз айырмашылығы – әртүрлі параметрлермен базалық кластың конструкторын шақыртуда.

«**AutoResetEvent** турникетке өте ұқсас – бір билет бір адамға өтуге мүмкіндік береді. “auto” деген сөз ашық турникет біреу өтіп кеткен соң автоматты түрде жабылатындығы немесе біреуге өтуге мүмкіндік берген соң «алынып тасталынатындығы» туралы дерекке қатысты. Ағын **WaitOne** шақыртуымен турникет алдында блокталады (ашылмағанға дейін осы (*one*) турникет алдында тосу (*wait*), ал билет **Set** әдісін шақыртумен қойылады. Егерде бірнеші ағын **WaitOne**-ні шақыртса, турникет артында кезек пайда болады. Билет кез келген ағынды, басқа сөзбен айтсақ, **AutoResetEvent** объектіге қатынауы бар кез келген (блокталмаған) ағынды «қоя алады», бір блокталған ағынды өткізу үшін **Set**-ті шақырта алады.

Егерде **Set** күтуші ағындар жоқ болғанда шақыртылса, хэндл қандай да бір **WaitOne**-ні шақыртқанша дейін ашық күйде болады**.** Осы ерекшелік турникетке жақындайтын ағындар арасында және билетті қыстыратын ағын (“опа, билет микросекундқа ертерек салынды, өкінішке орай, сіз қандай да бір уақыт тосуыңыз керексіз!”) арасындағы жарыстардан бас тартуға мүмкіндік береді. Бірақ, бос турникет үшін **Set** көп рет шақырту бір дегенде тұтас жиынды өткізбейді – тек бір адам өте алады, барлық қалған билеттер босқа жұмсалатын болады.

WaitOne міндетті емес **timeout** параметрді қабылдайды – егер әдіс күту сигналды алумен емес, таймаут бойынша аяқталса, онда **false**-ні қайтарады. тым көп блоктауға түспеу үшін күтуді жалғастыру (егерде втоматты блоктаумен

режім қолданылса) үшін **WaitOne**-ді ағымдағы синхрондаудың контекстінен шығуға үйретуге алады.

Reset әдісі еш күтсулерсіз және блоктаусыз ашық турникеттің жабылуын қамтамасыз етеді – интернеттен сілтеме.

AutoResetEvent блоктаушы конструкциясын турникетпен салыстыру өте сәтті, сондықтанда интернетке сілтеме келтірілген, бірақ AutoResetEvent-тің тағы бір артықшылығын атап кету қажет – оны атаулы етуге болады. Яғни әр тізбекке өз оқиғасын «бекітуге» болады, ол осы тізбектің жұмысын шешетін болады. Әддете, set() көмегімен қосымшаны қосқанда, қандай да бір тізбектің жұмысына рұқсат беретін қандай да бір оқиға бекітіледі. Осы тізбектің жұмысы аяқталған соң Reset-пен оқиғаны «түсіріп тастау» және set-пен келесі тізбектің жұмысына рұқсат беретін келесі оқиғаны орнату және т.б..

AutoResetEvent екі блоктаушы конструкциясының көмегімен екі жұптың жұмысын реттейтін оқу мысалы ретінде EventWaitHandle типінде ev1 және ev2 екі объект құрылатын бағдарлама ұсынылады. ev1 объектісі 1 тізбекті басқару, ал ev2 объектісі 2 тізбекті басқару үшін арналған. Екі тізбекте жұмысқа іске қосылған соң (олардың екеуіде өз объектілерінен рұқсатты тосады-`ev1.WaitOne(); және ev2.WaitOne();`)`ev1.Set();`бірінші оқиға орнатылады, ол 1 тізбектің жұмысына рұқсат береді. 1 тізбектің жұмыс циклы аяқталған соң, 1 оқиға алынып тасталынады, ал 2 оқиға орнатылады және т.б.

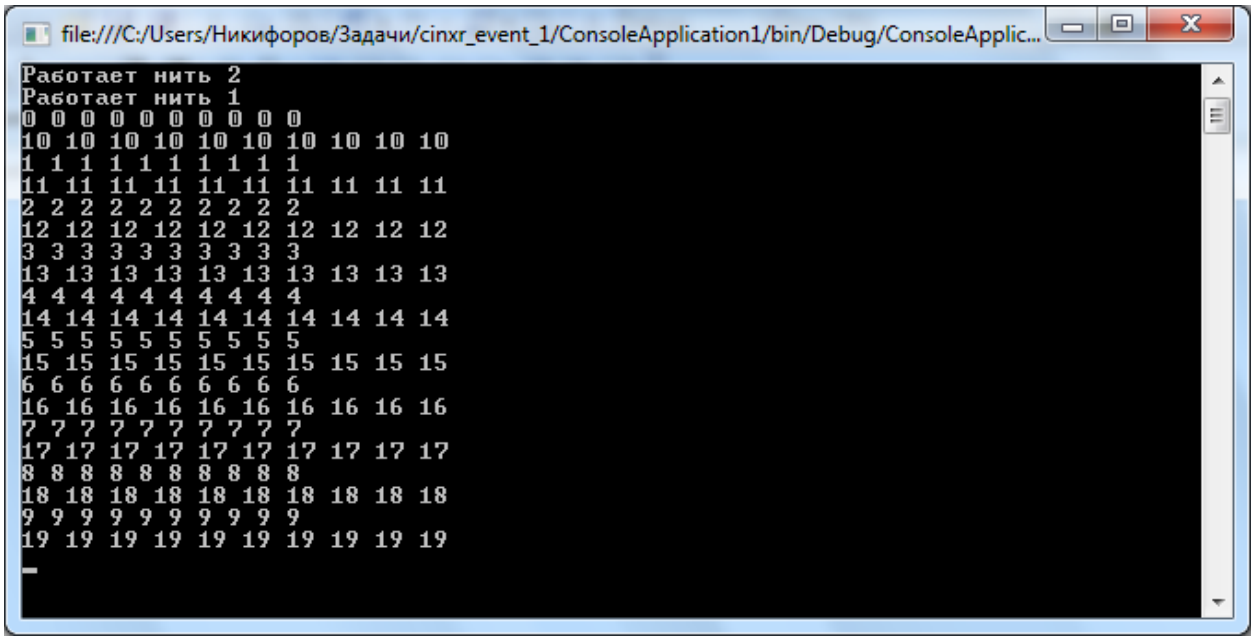
Бағдарлама коды:

```
using System;
using System.Threading;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
 class Program
 {
  static EventWaitHandle ev1 = new AutoResetEvent(false);
  static EventWaitHandle ev2 = new AutoResetEvent(false);
  class Pervaj_1
  {
   public void Niti_1()
   {
    Console.WriteLine("Работает нить 1 ");
    for (int i = 0; i < 10; i++)
    {
     ev1.WaitOne();
     for (int j = 0; j < 10; j++)
         Console.Write(i + " ");
     Console.WriteLine();
     ev1.Reset();
     ev2.Set();
    }
   }
   public void Niti_2()
   {
    Console.WriteLine("Работает нить 2 ");
    for (int i = 10; i < 20; i++)
    {
     ev2.WaitOne();
```

```
      for (int j = 0; j < 10; j++)
          Console.Write(i + " ");
      Console.WriteLine();
      ev2.Reset();
      ev1.Set();
     }
    }
   }
   static void Main(string[] args)
   {
    Pervaj_1 Pe = new Pervaj_1();
    Thread t1 = new Thread(Pe.Niti_1);
    t1.Start();
    Thread t2 = new Thread(Pe.Niti_2);
    t2.Start();
    ev1.Set();
    Console.ReadLine();
   }
  }
}
```

Бағдарлама жұмысы:



12.1-сурет –Тізбектерді AutoResetEvent көмегімен синхрондау

Close әдісін **WaitHandle** қажет болмай қалған соң дереу шақырту қажет - операциялық жүйенің ресурстарын босату үшін. Бірақ, егерде **WaitHandle** қосымшаның бүкіл өмір бойында қолданылса (біздің жағдайымыздағыдай), осы қадамды қалдыруға болады, өйткені ол қосымша жабылғанда автоматты түрде орындалады.