Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

Lucrare de laborator nr. 2 la Programarea în Rețea

Tema: *Programare Multi-Threading*

Efectuat st. gr. TI-142:

Chicu Roman.

Verificat lect. asistent.:

Ostapenco Stepan.

Scopul lucrării:

Lucrarea de laborator are ca scop studiul și înțelegerea proprietăților firelor.Stările unui fir de execuție.Lansarea, suspendarea și oprirea unui fir de execuție.Grupuri de Thread-uri.Elemente pentru realizarea comunicării și sicronizării.

Obiectiv:

Fiind dată diagrama dependențelor cauzale de modelat activitățile reprezentate de aceastea prin fire de execuție.

Link la repozitoriu: https://github.com/logan11116/lab1

Principiul de **multithreading** presupune execuția mai multor thread-uri în același pipeline, fiecare având propria secțiune de timp în care este menit să lucreze. Odată cu creșterea capabilităților procesoarelor au crescut și cererile de performanță, asta ducând la solicitarea la maxim a resurselor unui procesor. Necesitatea multithreading-ului a venit de la observația că unele procesoare puteau pierde timp prețios în așteptarea unui eveniment pentru o anumită sarcină.

Principiul multithreading-ului cu granularitate fină stă în faptul că fiecare instrucțiune va fi preluată de un alt fir de execuție, astfel neavând două instrucțiuni din același fir de execuție prezente în același timp în pipeline. Avantajul major al acestui tip de procesor este faptul că latența cauzată de anumite evenimente este folosită eficient de alte fire de execuție. Pentru a avea o eficiență maximă sunt necesare cel puțin atâtea fire de execuție câte etape are pipe-ul, altfel fiind mai ineficiente decât procesoarele scalare. Complexitatea hardware crește deoarece fiecare registru trebuie duplicat pentru fiecare fir de execuție, însă complexitatea pipeline-ului scade deoarece fiecare instrucțiune este independentă de toate celelalte.

Mersul lucrării:

Primul pas cuprinde studierea diagramei dependețelor cauzale (Figura 1).

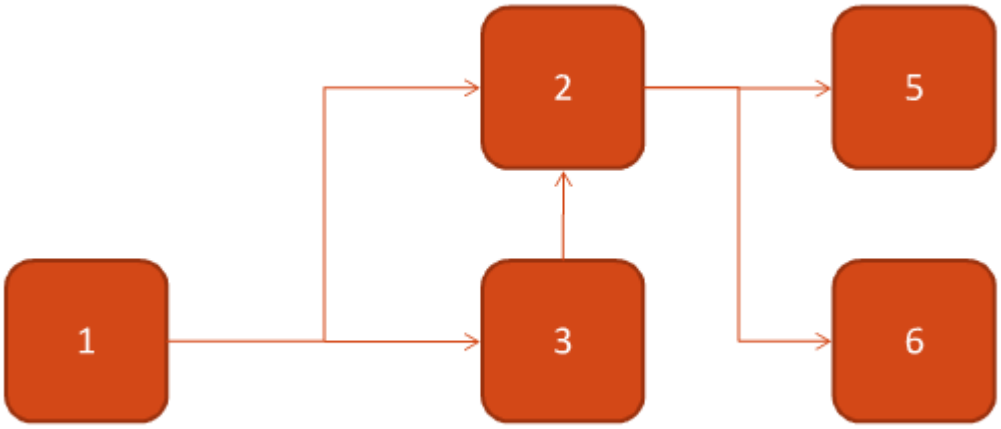


Figura 1 – Diagrama dependențelor cauzale

Următorul pas cuprinde crearea clasei principale *MyThread*.Această clasa cuprinde 3 *CountdownEvent*-uri și 7 *Thread*-uri create de utilizator.

Am folosit *CountdownEvent*-urile deoarece ele servesc ca una din metodele de sincronizare care pornesc firele de așteptare dupa ce au fost semnalate de un anumit număr de ori (Figura 2).

```
static CountdownEvent _countdown = new CountdownEvent(1);
static CountdownEvent _countdown2 = new CountdownEvent(1);
static CountdownEvent _countdown3 = new CountdownEvent(1);
```

Figura 2 – CountdownEvent-urile

Apoi urmează crearea a 7 fire de execuție.Crearea firelor de execuție are loc prin instanțierea unui obiect *Thread*, al carui constructor va cere ca parametru un delegate de tipul .*CurrentThread* (Figura 3). Această proprietate va returna firul de execuție curent in timpul execuției.

```
public void T1()
{
    Thread F = Thread.CurrentThread;
    Console.WriteLine(F.Name);
    _countdown.Signal();

}
```

Figura 3 – Crearea *Thread*-ului

Odată cu crearea Thread-urilor se stabilește paramentrul *_countdown.wait*(), care va permite lansarea *Thread*-urilor dupa ordinea anumită conform variantei.Spre exemplu, *Thread*-ul 4 pentru a fi lansat așteapta lansarea celui de al treilea *Thread* (Figura 4).

```
public void T2()
{
    _countdown.Wait();
    Thread F = Thread.CurrentThread;
    Console.WriteLine(F.Name);
    _countdown.Signal();
    _countdown2.Signal();

}
```

Figura 4 – Parametrul *countdown.wait()*

În final are loc declararea celor 7 *Thread*-uri împreună cu obictele din clasa firelor de execuție al cărui constructor ia o referință a unei clase de tip *ThreadStart*.

ThreadStart-ul stabilește care metodă trebuie efectuată in primul rând atunci când se lansează un *Thread*. Acest paramentrul este numele funcției, care tot-odată este considerată ca o funcție a firului de execuție.

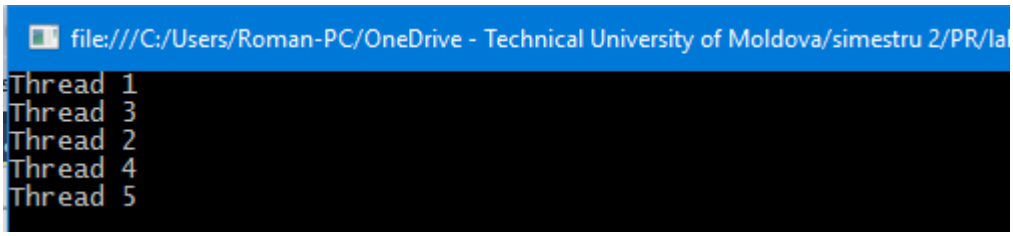
Procesul final al programului dat este prezentat in Figura 5.

```
public static void Main()
        {


          Example F1 = new Example();
          Example F2 = new Example();
          Example F3 = new Example();
          Example F4 = new Example();
          Example F5 = new Example();


         Thread tid1 = new Thread(new ThreadStart(F1.T1));
         Thread tid2 = new Thread(new ThreadStart(F2.T2));
          Thread tid3 = new Thread(new ThreadStart(F3.T3));
          Thread tid4 = new Thread(new ThreadStart(F4.T4));
          Thread tid5 = new Thread(new ThreadStart(F5.T5));

          tid1.Name = "Thread 1";
          tid2.Name = "Thread 2";
          tid3.Name = "Thread 3";
          tid4.Name = "Thread 4";
          tid5.Name = "Thread 5";


           tid1.Start();
          tid2.Start();
          tid3.Start();
          tid4.Start();
          tid5.Start();


         Console.ReadKey();


          }
```

Figura 5 – Lansarea Thread-urilor

Rezultatul execuției programului (Figura6):

Figura 6 – Lansarea Thread-urilor



Concluzie

În urma acestei lucrări de laborator au fost obținute abilități de lucru cu Multi-Threadingul in mediul C#.Au fost create mai multe fire de execuție care au fost pornite dupa o ordine anumită stabilită din diagrama dependențelor cauzale.

Bibliografie

1. [Resursă electronică]:

   https://www.codeproject.com/Articles/1083/Multithreaded-Programming-Using-C

2. [Resursă electronică]:

   https://msdn.microsoft.com/en-us/library/btky721f.aspx

3. [Resursă electronică]:

   http://www.albahari.com/threading/

Anexă

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.IO;
using System.Data;


public class Example
{
    // A semaphore that simulates a limited resource pool.
    //
    private static Semaphore _pool;

    // A padding interval to make the output more orderly.
    private static int _padding;


    static CountdownEvent _countdown = new CountdownEvent(1);
    static CountdownEvent _countdown2 = new CountdownEvent(1);
    static CountdownEvent _countdown3 = new CountdownEvent(1);


    public static void Main()
    {



        Example F1 = new Example();
        Example F2 = new Example();
        Example F3 = new Example();
        Example F4 = new Example();
        Example F5 = new Example();


        Thread tid1 = new Thread(new ThreadStart(F1.T1));
        Thread tid2 = new Thread(new ThreadStart(F2.T2));
        Thread tid3 = new Thread(new ThreadStart(F3.T3));
        Thread tid4 = new Thread(new ThreadStart(F4.T4));
        Thread tid5 = new Thread(new ThreadStart(F5.T5));

        tid1.Name = "Thread 1";
        tid2.Name = "Thread 2";
        tid3.Name = "Thread 3";
        tid4.Name = "Thread 4";
        tid5.Name = "Thread 5";


        tid1.Start();
        tid2.Start();
        tid3.Start();
        tid4.Start();
        tid5.Start();


        Console.ReadKey();


    }

    public void T1()
    {
        Thread F = Thread.CurrentThread;
        Console.WriteLine(F.Name);
```

```
            _countdown.Signal();

        }

        public void T2()
        {
            _countdown.Wait();
            Thread F = Thread.CurrentThread;
            Console.WriteLine(F.Name);
            _countdown.Signal();
            _countdown2.Signal();

        }

        public void T3()
            {
            _countdown.Wait();
            Thread F = Thread.CurrentThread;
            Console.WriteLine(F.Name);
            _countdown3.Signal();
        }

        public void T4()
        {
            _countdown2.Wait();
            Thread F = Thread.CurrentThread;
            Console.WriteLine(F.Name);

        }

        public void T5()
        {
            _countdown2.Wait();
            Thread F = Thread.CurrentThread;
            Console.WriteLine(F.Name);

        }


    }
```