

**MINISTERUL EDUCATIEI REPUBLICII MOLDOVA**

**UNIVERSITATEA TEHNICA A MOLDOVEI**

**Facultatea „Calculatoare, Informatică și  
Microelectronică”**

**Departamentul Inginerie Software și Automatica**

# **RAPORT**

## **Programarea în rețea**

Lucrare de laborator Nr. 2

Tema: Programarea multi-threading

A elaborat:

st. gr. TI-142 Țurcan Tudor

A verificat:

asist. univ. Donos Eugenia

Chișinău 2017

## **Cuprins**

Scopul lucrării .....	3
Sarcina lucrării .....	3
Procese și Fire .....	4
Desfășurarea lucrării .....	5
Concluzie .....	8
Bibliography .....	9
Anexa A .....	10

## Scopul lucrării

Realizarea firelor de execuție în Java/C#. Proprietățile firelor. Stările unui fir de execuție. Lansarea, suspendarea și oprirea unui fir de execuție. Grupuri de Thread-uri. Elemente pentru realizarea comunicării și sincronizării.

<https://github.com/TurcanTudor/Programarea-in-Retea.git>

## Sarcina lucrării

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

Diagrama dependențelor cauzale determină o mulțime ordonată de evenimente/ activități ordonate de relația de cauzalitate. Evenimentele/ activitățile sunt reprezentate printr-un dreptunghi rotunjit, iar dependențele prin săgeți, primind astfel un graf orientat aciclic. (1)

Graful orientat aciclic conform variantei este prezentat mai jos în figura 1.

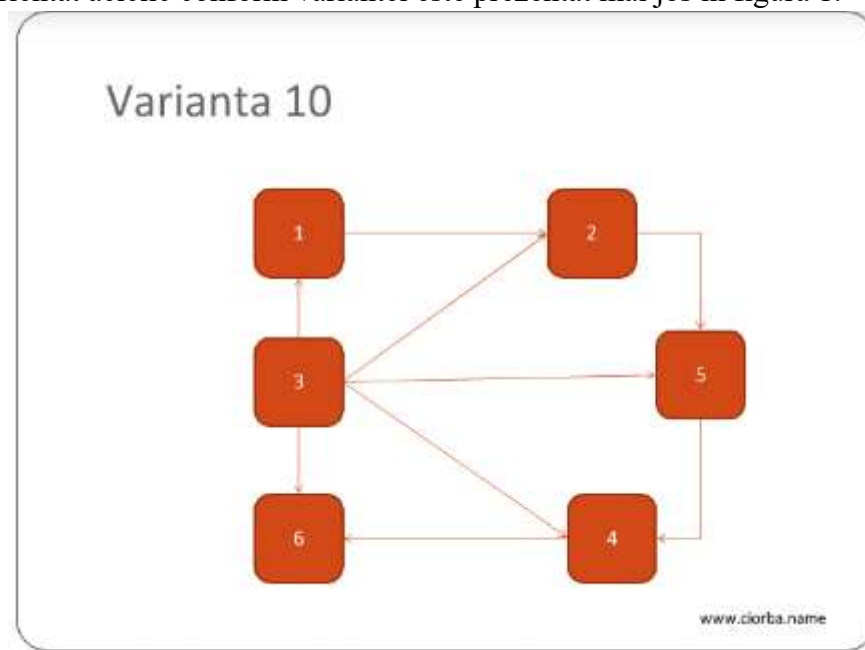


Figura 1 Diagrama dependențelor

## Procese și Fire

Un fir de execuție (thread) este un program secvențial, care poate fi executat concurrent cu alte fire. Un thread este o unitate de execuție într-un proces. Un proces poate avea mai multe fire de execuție, el numindu-se multithread. (2)

Dacă un calculator are mai multe procesoare sau un procesor cu mai multe nuclee, el poate executa mai multe fire de execuție simultan.

Diferența dintre un proces și un thread este că procesele sunt izolate total unul de celălalt, în timp ce thread-urile împart aceeași memorie (heap) cu alte thread-uri care rulează în aceeași aplicație (un thread poate prelua informații noi, în timp ce un alt thread le prelucrează pe cele existente). Folosirea firelor de execuție este o soluție la îmbunătățirea performanței.

Spațiul de nume `System.Threading` conține clasele din .Net FCL necesare pentru crearea și gestionarea firelor de execuție. O instanță a clasei `ThreadStart` (o clasă delegat) va încapsula metoda pe care o va executa o instanță a clasei `Thread` (adică un fir de execuție). Clasa `Thread` poate fi instantiată în mod uzual, cu `New`, sau se poate executa metoda statică `Thread.CurrentThread` pentru a obține o referință la firul de execuție curent.

Execuția unui fir se pornește cu metoda `Start`.

## Desfășurarea lucrării

Pentru a realiza diagrama dependențelor cauzale prin fire de execuție am avut nevoie de ales un limbaj de programare ce suporta lucrul cu firele de execuție. Astfel limbajul ales a fost C#. Acesta ne oferă librăria *System.Threading* care ne oferă librării și interfețe pentru lucrul cu firele de execuție. Totodată aici găsim clase pentru sincronizarea firelor de execuție prin diferite metode și accesul la date.

Conform schemei date anterior avem nevoie de creat 6 fire de execuție. Cu ajutorul clasei *Thread* cream 6 variabile care le instanțiem cu ajutorul constructorul clasei respective, oferindu-i ca parametru o funcție de afișare a unui mesaj la consolă. Declararea variabilelor și instanțierea acestora este prezentată în figura 2.

```
Thread t1 = new Thread(Write1);  
Thread t2 = new Thread(Write2);  
Thread t3 = new Thread(Write3);  
Thread t4 = new Thread(Write4);  
Thread t5 = new Thread(Write5);  
Thread t6 = new Thread(Write6);
```

Figura 2 Declararea variabilelor

Astfel având firele de execuție create avem nevoie să le lansăm concomitent. Realizarea acestei acțiuni este prezentată în figura 3. Utilizând metoda *Start()*; pentru fiecare fir în parte, noi schimbăm starea firelor de execuție din *unstarted* în *running*. (3)

```
t1.Start();  
t2.Start();  
t3.Start();  
t4.Start();  
t5.Start();  
t6.Start();  
t7.Start();
```

Figura 3 Lansarea firelor de execuție

După lansarea acestora principală problema cu care ne vom confrunta va fi sincronizarea firelor de execuție. Această problemă apare din cauza că noi nu știm cărui fir de execuție îi va fi predat controlul. Din această cauză avem nevoie de utilizat anumite structuri de sincronizare pentru firele noastre de execuție.

Structurile care au fost utilizate sunt *ManualResetEvent*, *AutoResetEvent*. (4) În figura 4 de mai jos sunt prezentate variabilele create cu ajutorul claselor structurilor date, ce vor fi utilizate pentru sincronizarea firelor de execuție.

```
static ManualResetEvent manual1 = new ManualResetEvent(false);
static AutoResetEvent auto1 = new AutoResetEvent(false);
static AutoResetEvent auto2 = new AutoResetEvent(false);
static AutoResetEvent auto3 = new AutoResetEvent(false);
static AutoResetEvent auto4 = new AutoResetEvent(false);
```

Figura 4 Declararea variabilelor

Analizând figura 1, observăm ca pentru lansarea firelor de execuție 1,2, 4,5 și acestea au nevoie să aștepte finalizarea firului 3 de execuție. Pentru aceasta am utilizat clasa *ManualResetEvent* care funcționează ca o poartă, ea permite deblocarea de către un singur fir de execuție a mai multor fire de execuție. Acest fapt se realizează prin utilizarea metodelor din clasa dată, și anume a metodei *Set()* care deschide poarta, permițând oricărui fir ce a apelat metoda *WaitOne()* să treacă prin ea. Utilizarea metodei *Set()* și *WaitOne()* este afișată în figura 5.

```
static void Write1()
{
    manual1.WaitOne();
    Console.WriteLine("Thread 1");
    auto1.Set();
}

static void Write2()
{
    auto1.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 2");
    auto2.Set();
}

static void Write3()
{
    Console.WriteLine("Thread 3");
    manual1.Set();
}

static void Write4()
{
    auto4.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 4");
    auto3.Set();
}
```

Fig. 5 Implementarea în cod

Astfel se poate observa că metoda *Set()* a fost chemată din primul fir de execuție, iar metoda *WaitOne()* a fost chemată din firele 1,2,5,4 și 6.

O altă structură utilizată a fost *AutoResetEvent* care seamănă ca un bilet pentru un turnichet. Astfel inserarea unui bilet permite deblocarea unui singru fir de execuție. Cu alte cuvinte firul ce invocă metoda *WaitOne()* așteaptă la turnichet pînă cînd acesta este deschis de un alt fir de execuție ce invocă metoda *Set()*. Implementarea acesteia se observa în figura 6, unde firul 1,2 , 5, 4 apelează metoda *Set()* iar firul 2,5,4, 6 apelează metoda *WaitOne()*. Al dependenței pe care o conține (5)

```
{
    manual1.WaitOne();
    Console.WriteLine("Thread 1");
    auto1.Set();
}

static void Write2()
{
    auto1.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 2");
    auto2.Set();
}

static void Write3()
{
    Console.WriteLine("Thread 3");
    manual1.Set();
}

static void Write4()
{
    auto4.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 4");
    auto3.Set();
}

static void Write5()
{
    auto2.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 5");
    auto4.Set();
}

static void Write6()
{
    auto3.WaitOne();
    manual1.WaitOne();
    Console.WriteLine("Thread 6");
}
```

Fig. 6 Apelarea metodei *Wait()*

## Concluzie

În urma efectuarea lucrării date de laborator s-au obținut deprinderi de lucru cu firele de execuție și sincronizarea acestora. Sa observat ca firele de execuție sunt concurente și independente unele de altele, necesitând structuri de sincronizare pentru realizarea sarcinii propuse. Pentru rezolvarea problemei propuse s-au utilizat următoarele clase *ManualResetEvent* și *AutoResetEvent* care mi-a permis sincronizarea acțiunilor firelor de execuție.



## Bibliography

1. **Dumitru, Ciorbă.** Programarea multi-threading. Note de laborator. [Interactiv] [Citat: 3 Martie 2016.] <http://moodle.ati.utm.md/mod/book/view.php?id=1855>.
2. **@Ale.** Ale's blog. *Fire de execuție*. [Online] Noiembrie 17, 2010. [Cited: Martie 13, 2016.] <http://ale-d-ale-vietii.blogspot.md/2010/11/fire-de-executie.html>.
3. **Joseph Albahari, O'Reilly Media, Inc.** ThreadState. 2006-2014.
4. —. Signaling with Event Wait Handles. 2006-2014.
5. **Joseph Albahari, O'Reilly Media, Inc.** *Threading in C# from Chapters 21 and 22*. 2006-2014.
6. **Dumitru, Ciorbă.** Bursting bubble-threading. [Interactiv] 12 Septembrie 2013. [Citat: 10 Martie 2016.] <http://ciorba.name/?p=1308>.

# Anexa A

Codul sursă

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace PR
{
    class Program
    {
        static ManualResetEvent manual1 = new ManualResetEvent(false);
        static AutoResetEvent auto1 = new AutoResetEvent(false);
        static AutoResetEvent auto2 = new AutoResetEvent(false);
        static AutoResetEvent auto3 = new AutoResetEvent(false);
        static AutoResetEvent auto4 = new AutoResetEvent(false);

        static void Main(string[] args)
        {

            Thread t1 = new Thread(Write1);
            Thread t2 = new Thread(Write2);
            Thread t3 = new Thread(Write3);
            Thread t4 = new Thread(Write4);
            Thread t5 = new Thread(Write5);
            Thread t6 = new Thread(Write6);

            t1.Start();
            t2.Start();
            t3.Start();
            t4.Start();
            t5.Start();
            t6.Start();

            Console.ReadLine();

        }

        static void Write1()
        {
            manual1.WaitOne();
            Console.WriteLine("Thread 1");
            auto1.Set();
        }

        static void Write2()
        {
            auto1.WaitOne();
            manual1.WaitOne();
            Console.WriteLine("Thread 2");
            auto2.Set();
        }

        static void Write3()
        {

```

```
        Console.WriteLine("Thread 3");
        manual1.Set();
    }
    static void Write4()
    {
        auto4.WaitOne();
        manual1.WaitOne();
        Console.WriteLine("Thread 4");
        auto3.Set();
    }

    static void Write5()
    {
        auto2.WaitOne();
        manual1.WaitOne();
        Console.WriteLine("Thread 5");
        auto4.Set();
    }
    static void Write6()
    {
        auto3.WaitOne();
        manual1.WaitOne();
        Console.WriteLine("Thread 6");
    }
}
}
```