MINISTERUL EDUCATIEI REPUBLICII MOLDOVA

UNIVERSITATEA TEHNICA A MOLDOVEI

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Inginerie Software si Automatica

RAPORT

Programarea în rețea

Lucrare de laborator Nr. 2

Tema: Programarea multi-threading

A elaborat: st. gr. TI-142 Comanda Artur

A verificat: lector asistent Ostapenco Stepan

Chișinău 2017

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/MihaiCapra/PR2

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.

https://github.com/Aillyrored/GitPR

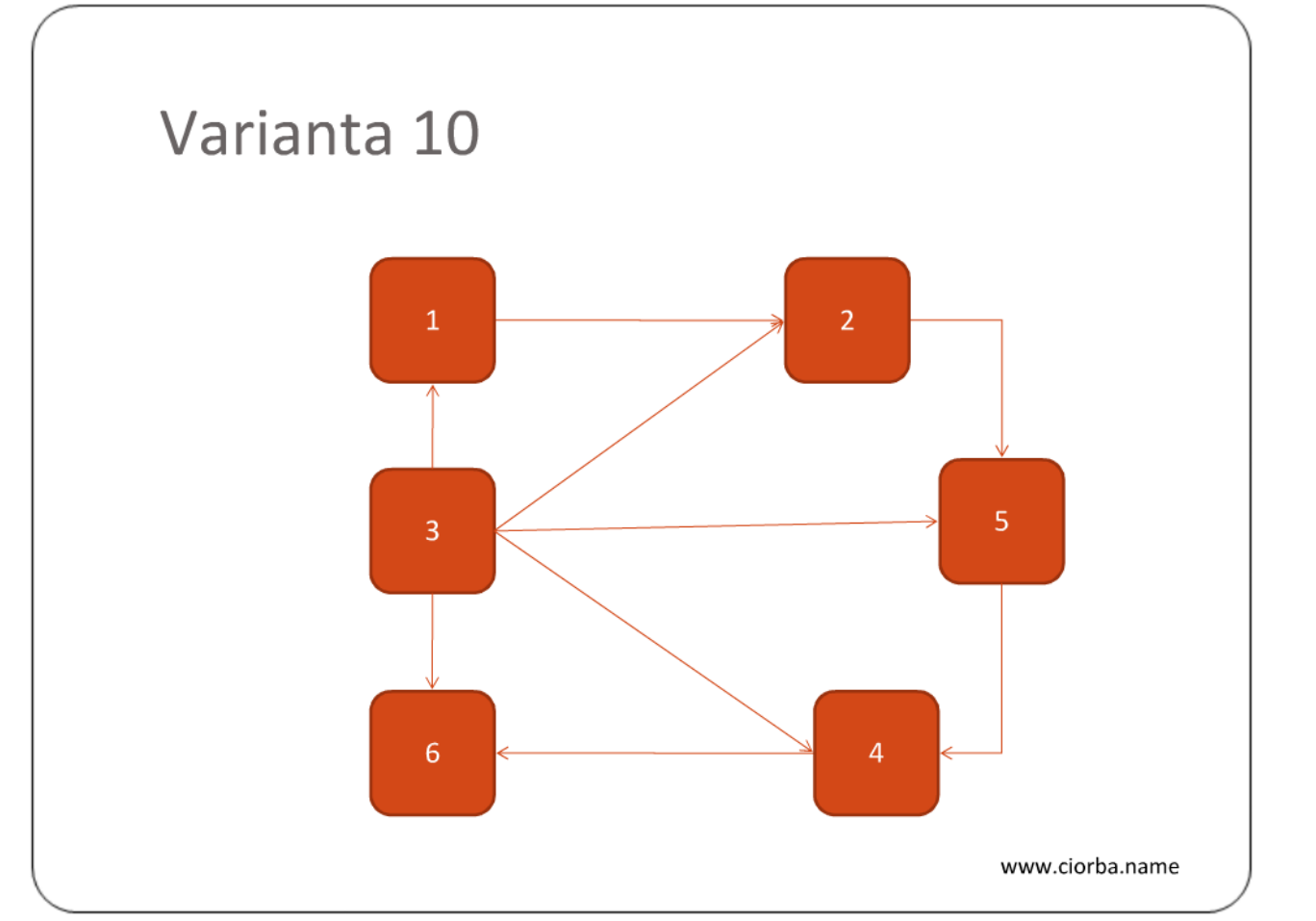

Figura 1 – Diagrama dependențelor.

Procese și Fire

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

Daca 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 si un thread este ca procesele sunt izolate total unul de celălalt, in timp ce thread-urile împart aceeași memorie (heap) cu alte thread-uri care rulează in aceeași aplicație (un thread poate prelua informații noi, in timp ce un alt thread le prelucrează pe cele existente). Folosirea firelor de execuție este o soluție la îmbunătățirea performantei.

Spațiul de nume System.Threading conține clasele din .Net FCL necesare pentru crearea si gestionarea firelor de execuție. O instanța a clasei ThreadStart (o clasa delegat) va încapsula metoda pe care o va executa o instanța a clasei Thread (adică un fir de execuție). Clasa Thread poate fi instantiata in mod uzual, cu New, sau se poate executa metoda statica 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.* 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 constructorului clasei respective, oferindu-i ca parametru o funcție de afișare a unui mesaj la consolă. Declararea variabilelor și instanțierea acestora este prezentată in Figura 2.

```
private static readonly AutoResetEvent[] Handlers =
{
    new AutoResetEvent(false),
    new AutoResetEvent(false),
    new AutoResetEvent(false),
    new AutoResetEvent(false),
    new AutoResetEvent(false),
    new AutoResetEvent(false)
};
```

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.*

După lansarea acestora, principală problemă cu care ne vom confrunta va fi sincronizarea firelor de execuție. Această problemă apare din cauza că noi nu știm cărui fir de execuție ii 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.* (3) În Figura 3 de mai jos sunt prezentate variabilele create cu ajutorul claselor structurilor date, ce vor fi utilizate pentru sincronizarea firelor de executie.

```
static void Main(string[] args)
{
    UseTasks();
    UseThreads();
    UseThreadsSyncEvent();

}
```

Figura 3 - Declararea variabilelor.

Analizând Figura 1 ,observam ca pentru lansarea firelor de execuție 2 și 3 trebuie sa aștepte finalizarea execuției firului 1 .Pentru aceasta am utilizat clasa *ManualResetEvent* care functioneaza ca o poarta , 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 data , și anume a metodei *Sett()* care deschide poarta , permițind oricărui fir ce a apelat medoda *WaitOne()* ss treacă prin ea. Utilizarea metodelor *Set()* și *WaitOne()* sunt afișat în figura 4.

```
private static void UseThreads()
{
    var threads = new Thread[6];
    for (var i = 0; i < threads.Length; i++)
    {
        threads[i] = new Thread(Run) { Name = $"Thread {i + 1}" };
    }
    threads[2].Start(null);
    threads[0].Start(threads[2]);
    threads[1].Start(threads[0]);
    threads[4].Start(threads[1]);
    threads[3].Start(threads[4]);
    threads[5].Start(threads[3]);
    threads[5].Join();
    Console.WriteLine("All threads have finished the work");
}
```

Figura 4 – Implementarea în cod.

Astfel se poate de observant că metoda *Set()* a fost chemata din primul fir de execuție , iar metoda *WaitOne() a* fost chemată din firele 3,4,5,6.

O altă structură utilizată a fost *AutoResetEvent* care seamănă ca un bilet pentru un turnichet .Astfel înserarea unui bilet permite deblocarea unui singru fir de execuție .Cu alte cuvinte firul ce invocă metoda *WaitOne()* așteaptă la turnichet pînă când acesta este deschis de un alt fir de execuție ce invocă metoda *Set()* .

```
static void UseTasks()
{
    var tasks = new Task[6];
    tasks[2] = Task.Run(() =>
    {
        for (var i = 0; i < 3; i++)
        {
            Console.WriteLine($"Task 3 - {i}, TaskId: {Task.CurrentId}");
            Task.Delay(500).Wait();
        }
    });
    tasks[0] = tasks[2].ContinueWith((t) =>
    {
        for (var i = 0; i < 3; i++)
        {
            Console.WriteLine($"Task 1 - {i}, TaskId: {Task.CurrentId}");
            Task.Delay(500).Wait();
        }
    });
```

Figura 5-Implementarea metodei *WaitOne()*.

Concluzie

În procesul de efectuarea a lucrarii d de laborator am 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 proplemei propuse s-au utilizat următoarele clase *ManualResetEvent și AutoResetEvent* care mi-a permis sincronizarea acțiunilor firelor de execuție.

Bibliografie

1. **@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.

2. **Joseph Albahari, O'Reilly Media, Inc.** ThreadState. 2006-2014.

3. —. Signaling with Event Wait Handles. 2006-2014.

4. **Joseph Albahari, O'Reily Media, Inc.** *Threading in C# from Chapters 21 and 22.* 2006- 2014.