

UNIVERSITATEA TEHNICA A MOLDOVEI
Facultatea „Calculatoare, Informatică și Microelectronică”
Departamentul “Inginerie Software si Automatică”

RAPORT

Programarea în rețea
Lucrare de laborator nr. 2

Tema: Programare multi-threading

A efectuat:

st. gr. TI-142 Donca Diana

A verificat:

lect. asist. Donos Eugenia

Chisinau 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

Obiectivele lucrării

Înțelegerea modelelor de execuție concurentă și cunoașterea tehnicilor esențiale de sincronizare ale activităților bazate pe operațiile atomare ale semaforului, obiectivul specific constând în crearea unei aplicații Java/C# ce ar utiliza sigur diverse structuri într-un context de execuție concurentă.

Sarcina

Fiind dată diagrama dependențelor cauzale de modelat activitățile reprezentate de acestea prin fire de execuție. Diagrama dependențelor care trebuie să fie realizată, este reprezentată în figura 1.

Link-ul la repozitoriu: <https://github.com/ddonca/lab2PR>

Notiuni teoretice:

Programarea concurentă

Multithreading înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp. O astfel de secvență de cod se numește fir de execuție sau *thread*. Datorită posibilității creării mai multor thread-uri, un program Java poate să execute mai multe sarcini simultan (de exemplu, animația unei imagini, transmiterea unei melodii la placa de sunet, comunicarea cu un server).

Marea majoritate a sistemelor de operare actuale (Microsoft Windows, UNIX, OS/2) permit gestionarea multiprogramată a proceselor. Folosirea în comun de către mai multe programe aflate în memorie, a resurselor sistemului duce la o îmbunătățire spectaculoasă a gradului de utilizare al acestora.

Programele rulate sub sistemele cu multiprogramare formează prin urmare un set de procese paralele executate independent între ele. Comportarea lor individuală este practic identică celei din cazul rulării fiecăruia într-un sistem monoprogramat. Eventualele conflicte rezultate din utilizarea comună a unor resurse se rezolvă prin sistemul de operare, fără ca programatorul să ia vreo măsură în acest sens.

Programele care reprezintă procese izolate (programe secvențiale) nu permit rezolvarea unor anumite categorii de probleme. Deseori devine necesară prezenta simultană a mai multor procese, executate în paralel și asincron, dar între care există relații de cooperare (schimb de mesaje, transfer de date) și care gestionează în comun resurse ale sistemului. Ele se numesc procese concurente.

Sincronizarea conform normelor generale este înțeleasă ca acțiunea de a face ca două sau mai multe fapte, evenimente sau fenomene să se petreacă în același timp. Semnalizarea este tehnica prin care un fir informează altul despre o activitate realizată, astfel garantându-se că un fragment de cod al unui fir se va realiza în mod obligatoriu înainte de al altuia. Acest lucru rezolvă problema serializării instrucțiunilor într-un mediu de execuție concurrent, în care ordinea global de execuție din firele programului nu este cunoscută.

Realizarea sarcinii:

Varianta 3

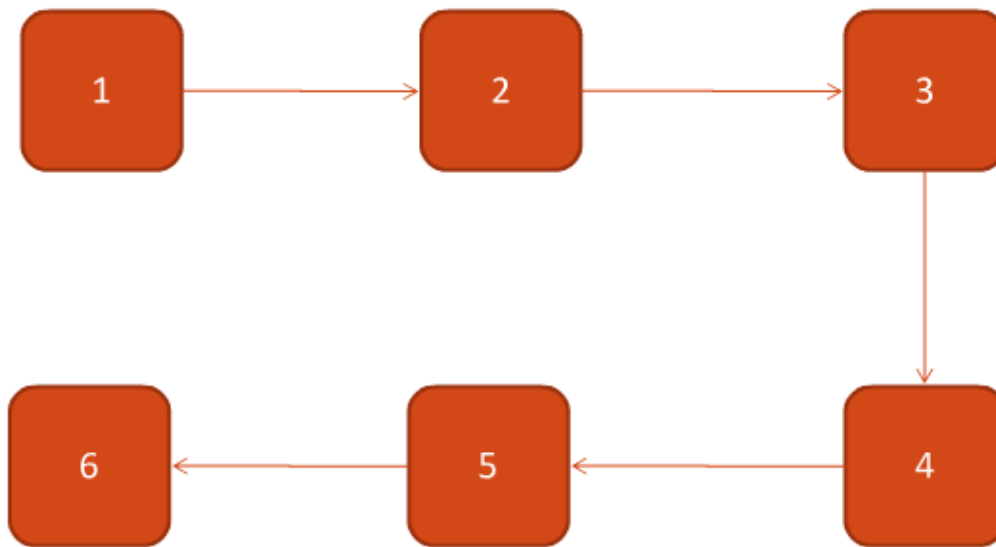


Figura 1 – Varianta lucrării de laborator

1. Declarăm o variabilă de tip `CountDownLatch`, care se folosește la indicarea firelor care trebuie să se aștepte (figura2).

```
static CountdownEvent count1 = new CountdownEvent(1);  
static CountdownEvent count2 = new CountdownEvent(1);  
static CountdownEvent count3 = new CountdownEvent(1);  
static CountdownEvent count4 = new CountdownEvent(1);  
static CountdownEvent count5 = new CountdownEvent(1);  
static CountdownEvent count6 = new CountdownEvent(1);
```

Figura 2 - Declararea variabilei de tip `CountDownLatch`

2. Declararea celor 5 fire de execuție (figura 3). Prin static se înțelege faptul că pentru apelarea metodei nu este nevoie de un obiect a clasei, apelul la metodă poate fi efectuat cu numele clasei - `NumeClasa.NumeMetodaStatice()`.

```
Thread th1 = new Thread>Show1);  
th1.Name = "Thread 1";  
th1.Start();  
  
Thread th2 = new Thread>Show2);  
th2.Name = "Thread 2";  
th2.Start();  
  
Thread th3 = new Thread>Show3);  
th3.Name = "Thread 3";  
th3.Start();  
  
Thread th4 = new Thread>Show4);  
th4.Name = "Thread 4";  
th4.Start();  
  
Thread th5 = new Thread>Show5);  
th5.Name = "Thread 5";  
th5.Start();  
  
Thread th6 = new Thread>Show6);  
th6.Name = "Thread 6";  
th6.Start();
```

Figura 3 – Declararea firelor de execuție

3. Apelarea metodei Print care corespunde pentru fiecare fir, care execută în ordinea dorită firele, prin apelarea semaforului și anume metodele Wait() și Signal() (Figura 4).

```

static void Show1()
{
    Console.WriteLine("Start .. " + Thread.CurrentThread.Name); Thread.Sleep(2000);
    count1.Signal();
    Console.WriteLine("End .. " + Thread.CurrentThread.Name);
}
static void Show2()
{
    count1.Wait();
    Console.WriteLine("Start .. " + Thread.CurrentThread.Name); Thread.Sleep(2000);

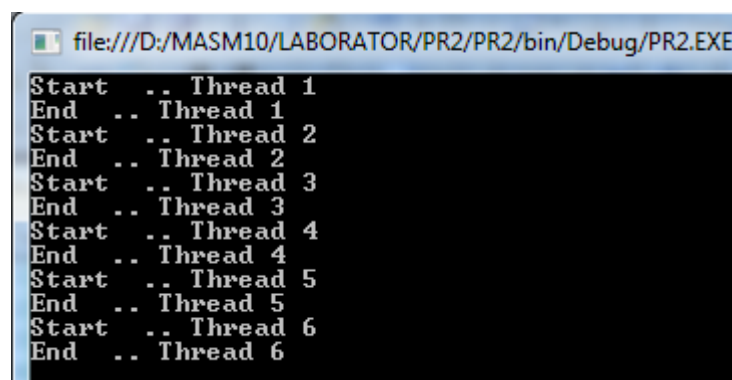
    count2.Signal();
    Console.WriteLine("End .. " + Thread.CurrentThread.Name);
}
static void Show3()
{
    count2.Wait();
    Console.WriteLine("Start .. " + Thread.CurrentThread.Name); Thread.Sleep(2000);

    count3.Signal();
    Console.WriteLine("End .. " + Thread.CurrentThread.Name);
}
static void Show4()
{
    count3.Wait();
    Console.WriteLine("Start .. " + Thread.CurrentThread.Name); Thread.Sleep(2000);
    count4.Signal();
    Console.WriteLine("End .. " + Thread.CurrentThread.Name);
}
static void Show5()
{

```

Figura 4 - Executarea firelor de execuție în ordinea dorită

Rezultatul afișării:



```

file:///D:/MASM10/LABORATOR/PR2/PR2/bin/Debug/PR2.EXE
Start .. Thread 1
End .. Thread 1
Start .. Thread 2
End .. Thread 2
Start .. Thread 3
End .. Thread 3
Start .. Thread 4
End .. Thread 4
Start .. Thread 5
End .. Thread 5
Start .. Thread 6
End .. Thread 6

```

Figura 5 – Rezultatul afișării

Concluzie

Firele de execuție permit executarea simultană a mai multor secvențe de cod. Firele de execuție pot partaja aceleași resurse, iar accesul la acestea poate fi concurent sau secvențial.

Pentru a nu apărea probleme de consistență, este bine ca modificarea datelor comune să se realizeze secvențial. În funcție de importanța operațiilor efectuate, un fir de execuție poate fi mai prioritar decât altele la primirea resurselor necesare rulării de la sistemul de operare, la fel este posibil stoparea unui fir până când altul nu s-a terminat cu ajutorul metodei `join()` din clasa `Thread`

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_Test
{
    class ThreadSafe
    {
        static CountdownEvent count1 = new CountdownEvent(1);
        static CountdownEvent count2 = new CountdownEvent(1);
        static CountdownEvent count3 = new CountdownEvent(1);
        static CountdownEvent count4 = new CountdownEvent(1);
        static CountdownEvent count5 = new CountdownEvent(1);
        static CountdownEvent count6 = new CountdownEvent(1);

        static void Main()
        {
            Thread th1 = new Thread(Show1);
            th1.Name = "Thread 1";
            th1.Start();

            Thread th2 = new Thread(Show2);
            th2.Name = "Thread 2";
            th2.Start();

            Thread th3 = new Thread(Show3);
            th3.Name = "Thread 3";
            th3.Start();

            Thread th4 = new Thread(Show4);
            th4.Name = "Thread 4";
            th4.Start();

            Thread th5 = new Thread(Show5);
            th5.Name = "Thread 5";
            th5.Start();

            Thread th6 = new Thread(Show6);
            th6.Name = "Thread 6";
            th6.Start();

            Console.ReadLine();}

        static void Show1( )
        {
```



```

        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);
        count1.Signal();
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
    static void Show2()
    {
        count1.Wait();
        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);

        count2.Signal();
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
    static void Show3()
    {
        count2.Wait();
        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);

        count3.Signal();
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
    static void Show4()
    {
        count3.Wait();
        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);
        count4.Signal();
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
    static void Show5()
    {
        count4.Wait();
        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);
        count5.Signal();
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
    static void Show6()
    {
        count5.Wait();
        Console.WriteLine("Start .. " + Thread.CurrentThread.Name);
Thread.Sleep(2000);
        Console.WriteLine("End .. " + Thread.CurrentThread.Name);
    }
}

```