Ministerul Educaţiei al Republicii Moldova

Universitatea Tehnică a Moldovei

Departamentul Ingineria Software și Automatică

**RAPORT**

Lucrare de laborator Nr.2

*la disciplina:*

*Programarea în Rețea*

***Tema:*** *Elemente ale procesării concurente*

A efectuat: st. gr. TI-144 Talpa Victor

A verificat: lect. univ. Ostapenco Stepan

Chişinău 2017

**Obiectivele lucrării:**

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

**Sarcina lucrării:**

Sarcina lucrării de laborator cuprinde o diagramă a dependențelor cauzale, care definește o mulțime de evenimente (activități) ordonate de relații de cauzalitate. În esență această diagramă este un graf orientat acicilic, în care evenimentele sunt reprezentate prin noduri, iar dependențele cauzale prin arcuri.

**Link la repozitoriu:** https://github.com/VictorTalpa/PR

**Varianta 9**

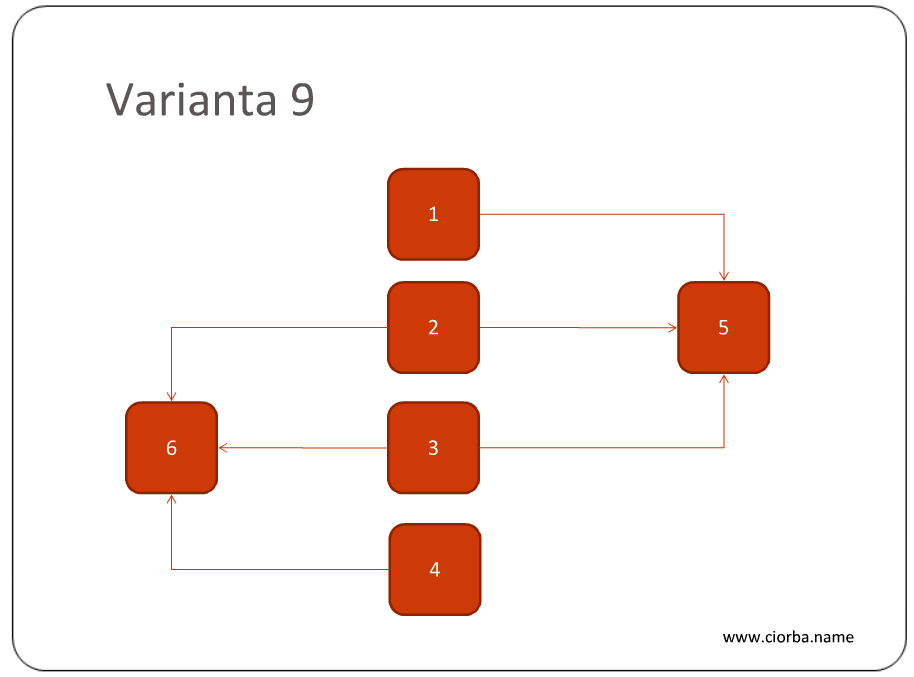
****

Figura 1. Varianta lucrării de laborator

**Modul de lucru:**

În graficul reprezentat mai sus observăm ca aplicația noastra trebuie să execute 6 thread-uri (fire de executie) si dependențele de rulare dintre acestea: thread-ul 5 se va executa la finisarea thread-urilor 1, 2 și 3, iar thread-ul 6 se va executa la finisarea thread-urilor 2, 3 și 4.

Am efectuat acest program în limbajul Java, folosind CountDownLatch ca metoda de gestionare a dependențelor dintre thread-uri. Obiectul de tip CountDownLatch se initializeaza cu o valoare ce reprezinta contorul de așteptare.

CountDownLatch timer1 = **new** CountDownLatch(3);  
CountDownLatch timer2 = **new** CountDownLatch(3);

în cazul dat, ambele obiecte vor aștepta executarea a cite 3 thread-uri.

După executarea unui thread de care depinde un alt thread, acest obiect se decrementează apelînd metoda .countDown();

Pentru aceasta avem nevoie sa extindem clasa Thread și să supraîncărcarea metodei run();

**public class** MyThread **extends** Thread {

@Override

**public void** run() {

**//Operații**

}

}

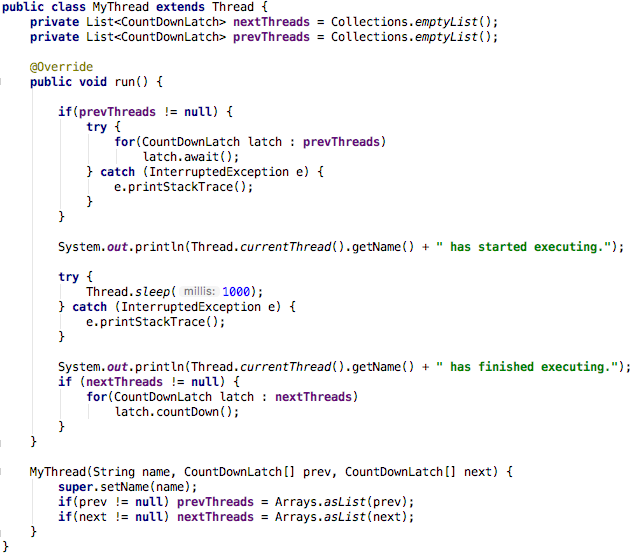


Figura 2. Extinderea clasei Thread

și supraîncărcarea metodei run()

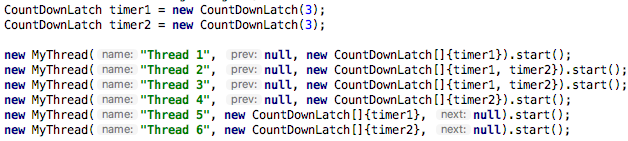


Figura 3. Crearea și rularea thread-urilor

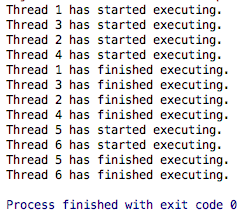


Figura 4. Afișarea rezultatelor

**Concluzie**

Lucrarea dată de laborator, ne-a permis să studiem conceptele legate de multithreading, paralelism și concurență. Am studiat modul de creare a thread-urilor și metodele lor caracteristice, precum și modul de utilizare eficientă a lor.

Conform sarcinii, am implementat o modalitate de simulare a condiției laboratorului, utilizînd Thread și CountDownLatch. Am studiat modul de creare a thread-urilor, prin extinderea clasei Thread.

Astfel a fost creată o aplicație ce execută șase thread-uri în serie respectînd dependențele de executare. Am studiat beneficiile utilizării multihreading-ului și problemele legate de implementarea acestuia.

**Bibliografie**:

1. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>
2. <https://www.tutorialspoint.com/java/java_multithreading.htm>