Ministerul Educaţiei al Republicii Moldova

Universitatea Tehnică a Moldovei

Departamentul Ingineria Software şi 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

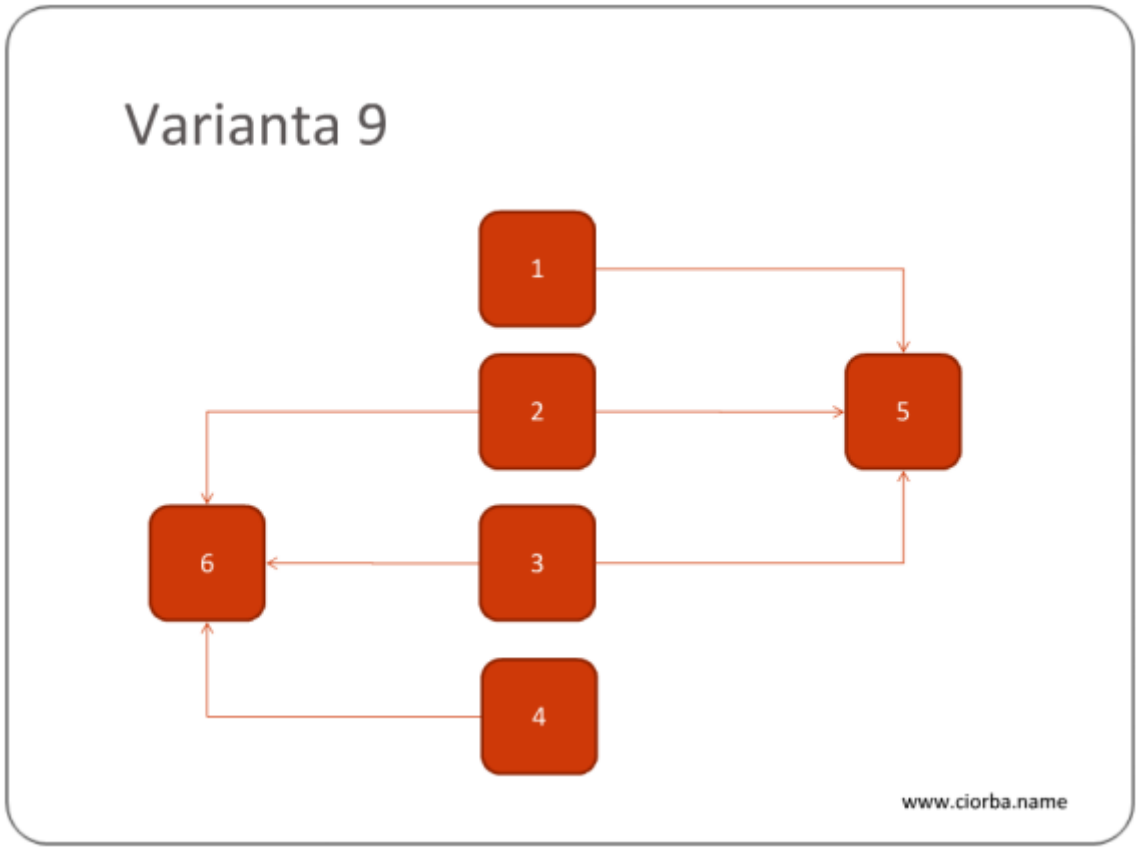

Figura 1. Varianta lucrării de laborator

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() {
		//Operaț ii
	}
}
```

```java
public class MyThread extends Thread {
    private List<CountDownLatch> nextThreads = Collections.emptyList();
    private List<CountDownLatch> prevThreads = Collections.emptyList();

    @Override
    public void run() {

        if(prevThreads != null) {
            try {
                for(CountDownLatch latch : prevThreads)
                    latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println(Thread.currentThread().getName() + " has started executing.");

        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " has finished executing.");
        if (nextThreads != null) {
            for(CountDownLatch latch : nextThreads)
                latch.countDown();
        }
    }

    MyThread(String name, CountDownLatch[] prev, CountDownLatch[] next) {
        super.setName(name);
        if(prev != null) prevThreads = Arrays.asList(prev);
        if(next != null) nextThreads = Arrays.asList(next);
    }
}
```

Figura 2. Extinderea clasei Thread
și supraîncărcarea metodei run()

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

new MyThread( name: "Thread 1",  prev: null, new CountDownLatch[]{timer1}).start();
new MyThread( name: "Thread 2",  prev: null, new CountDownLatch[]{timer1, timer2}).start();
new MyThread( name: "Thread 3",  prev: null, new CountDownLatch[]{timer1, timer2}).start();
new MyThread( name: "Thread 4",  prev: null, new CountDownLatch[]{timer2}).start();
new MyThread( name: "Thread 5", new CountDownLatch[]{timer1},  next: null).start();
new MyThread( name: "Thread 6", new CountDownLatch[]{timer2},  next: null).start();
```

Figura 3. Crearea și rularea thread-urilor

```
Thread 1 has started executing.
Thread 3 has started executing.
Thread 2 has started executing.
Thread 4 has started executing.
Thread 1 has finished executing.
Thread 3 has finished executing.
Thread 2 has finished executing.
Thread 4 has finished executing.
Thread 5 has started executing.
Thread 6 has started executing.
Thread 5 has finished executing.
Thread 6 has finished executing.

Process finished with exit code 0
```

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