1 Problema clasică producător - consumator.
1.1.Sincronizarea firelor de execuţie.

Există numeroase situaţii când fire de execuţie separate, dar care rulează concurent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-şi transmite dinamic rezultatele "muncii" lor. Cel mai elocvent scenariu în care firele de execuţie trebuie sa comunice între ele este cunoscut sub numele de problema *producătorului/consumatorului*, în care producătorul generează un flux de date care este preluat şi prelucrat de către consumator. Să considerăm de exemplu o aplicaţie Java în care un fir de execuţie (producătorul) scrie date într-un fişier, în timp ce alt fir de execuţie (consumatorul) citeşte date din acelaşi fişier pentru a le prelucra. Sau, să presupunem că producătorul generează numere şi le plasează pe rând într-un buffer, iar consumatorul citeşte numerele din acel buffer pentru a le interpreta. În ambele cazuri avem de-a face cu fire de execuţie concurente care folosesc o resursă comună: un fişier, un vector şi, ele trebuie sincronizate în dependență de activitatea lor. Pentru a înţelege mai bine modalitatea de sincronizare a două fire de execuţie să implementăm efectiv o problemă de tip producător/consumator. Să considerăm următoarea situaţie:

- Producătorul generează numerele întregi de la 1 la 10, fiecare într- un interval aleator cuprins între 0 şi 100 de milisecunde. Pe măsură ce le generează încearcă să le plaseze într-o zonă de memorie (o variabilă întreagă) de unde sa fie citite de către consumator.
- Consumatorul va prelua, pe rând, numerele generate de către producător şi va afişa valoarea lor la ecran.

Pentru a fi accesibilă, ambelor fire de execuţie, vom încapsula variabila ce va conţine numerele generate într-un obiect descris de clasa **`Buffer`** şi care va avea două metode **`put`** (pentru punerea unui număr în buffer) şi **`get`** (pentru obţinerea numărului din buffer). Fără folosirea nici unui mecanism de sincronizare clasa Buffer arată astfel:

```
class Buffer {
       private int number = -1;

       public int get() {
         return number;
       }

       public void put(int number) {
              this.number = number;
       }
}
```

Vom implementa acum clasele Producător şi Consumator care vor descrie cele două fire de execuţie. Ambele vor avea o referinţă comună la un obiect de tip `Buffer` prin intermediul căruia îşi comunică valorile.

```
class Producator extends Thread {
   private Buffer buffer;
      public Producator(Buffer b) {
        buffer = b;
      }
      public void run() {
       for (int i = 0; i < 10; i++) {
          buffer.put(i);
      System.out.println("Producatorul a pus:\t"+ i);
```

```
            try {
              sleep((int)(Math.random() * 100));
              } catch (InterruptedException e) { }
       }
       }
}

class Consumator extends Thread {
   private Buffer buffer;
   public Consumator(Buffer b) {
      buffer = b;
   }
   public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
       value = buffer.get();
       System.out.println("Consumatorul a
                           primit:\t" + value);
      }
   }
}
//Clasa principală
public class TestSincronizare1 {
       public static void main(String[] args) {
              Buffer b = new Buffer();
              Producator p1 = new Producator(b);
              Consumator c1 = new Consumator(b);
              p1.start();
              c1.start();
       }
}
```

Rezultatul rulării acestui program nu va rezolva nici pe departe problema propusă de noi, motivul fiind lipsa oricărei sincronizări între cele două fire de execuţie. Mai exact, rezultatul va fi aproximativ următor:

```
              Consumatorul a primit:   -1
              Consumatorul a primit:   -1
              Producatorul a pus:0
              Consumatorul a primit:   0
              Consumatorul a primit:   0
              Consumatorul a primit:   0
              Producatorul a pus:1
              Producatorul a pus:2
              Producatorul a pus:3
              Producatorul a pus:4
              Producatorul a pus:5
              Producatorul a pus:6
              Producatorul a pus:7
              Producatorul a pus:8
              Producatorul a pus:9
```

Ambele fire de execuţie accesează resursa comună, adică obiectul de tip `Buffer`, într-o manieră haotică şi acest lucru se întâmplă din două motive :

- consumatorul nu aşteaptă înainte de a citi ca producătorul să genereze un număr şi va prelua de mai multe ori acelaşi număr.
- producătorul nu aşteaptă consumatorul să preia numărul generat înainte de a produce un altul, în felul acesta consumatorul va "rata" cu siguranţă unele numere (în cazul nostru, aproape pe toate).

Problema constă în urmatoarele: cine trebuie să se ocupe de sincronizarea celor două fire de execuţie : clasele Producător şi Consumator sau resursa comună `Buffer`? Răspunsul este: resursa comună `Buffer`, deoarece ea trebuie să permită sau nu, accesul la conţinutul său, şi nu firele de execuţie care o folosesc. În felul acesta efortul sincronizării este transferat de la producător/consumator la un nivel mai jos, cel al resursei critice. Activităţile producătorului şi ale consumatorului trebuie sincronizate la nivelul resursei comune în două aspecte:

1. Cele două fire de execuţie nu trebuie să acceseze simultan buffer-ul; acest lucru se realizează prin blocarea obiectului `Buffer` atunci când este accesat de un fir de execuţie, astfel încât nici un alt fir de execuţie să nu-l mai poată accesa.
2. Cele două fire de execuţie trebuie să se coordoneze, adică producătorul trebuie să găsească o modalitate de a "spune" consumatorului că a plasat o valoare în buffer, iar consumatorul trebuie să comunice producătorului că a preluat această valoare, pentru ca acesta să poată genera o altă valoare. Pentru a realiza această comunicare, clasa `Thread` pune la dispoziţie metodele **`wait, notify, notifyAll`**.

Reieşind din cele expuse mai sus clasa `Buffer` va arăta astfel:

```
class Buffer {
	private int number = -1;
	private boolean available = false;
	public synchronized int get() {
		while (!available) {
			try {
				wait();
	//aşteaptă producatorul să pună o valoare
			} catch (InterruptedException e) { }
			}
			available = false;
			notifyAll();
			return number;
	}
	public synchronized void put(int number) {
		while (available) {
			try {
				wait();
	//aşteaptă consumatorul să preia valoarea
			} catch (InterruptedException e) { }
			}
			this.number = number;
			available = true;
			notifyAll();
	}
}
```

Rezultatul realizării:

```
Producatorul a pus:0
Consumatorul a primit:    0
Producatorul a pus:1
Consumatorul a primit:    1
. . .
Producatorul a pus:9
```

2. Metode de sincronizare a thread-urilor

2.1 Blocarea unui obiect (cuvântul cheie synchronized)

Definiţie: **Un segment de cod ce gestionează o resursă comună mai multor fire de execuţie separate şi concurente se numeşte *secţiune critică*.** În Java o secţiune critică poate fi un bloc de instrucţiuni sau o metodă.

Controlul accesului într-o secţiune critică se face prin cuvântul cheie **`synchronized`**. Platforma Java asociază un monitor fiecărui obiect al unui program ce conţine secţiuni critice care necesită sincronizare. Acest monitor va indica dacă resursa critică este accesată de vreun fir de execuţie sau este liberă, cu alte cuvinte "monitorizează" o resursă critică. În cazul în care este accesată, va "pune un lacăt" pe aceasta, astfel încât să împiedice accesul altor fire de execuţie la ea. În momentul când resursa este eliberată "lacătul" va fi eliminat pentru a permite accesul altor fire de execuţie.

În exemplul tip producător/consumator de mai sus, secţiunile critice sunt metodele **`put`** şi **`get,`** iar resursa citică comună este obiectul **`buffer`**. Consumatorul nu trebuie să acceseze buffer-ul când producătorul tocmai pune o valoare în el, iar producătorul nu trebuie sa modifice valoarea din buffer în momentul când aceasta este citită de către consumator.

```
public synchronized int get() {
        ...
}
public synchronized void put(int number) {
        ...
}
```

Ambele metode au fost declarate cu modificatorul **`synchronized`**. Cu toate acestea sistemul asociază un monitor unei instanţe a clasei **`Buffer`** şi nu unei metode anume. În momentul în care se preiamonitorul, firul de execuţie care a făcut apelul, va bloca obiectul a cărui metodă o accesează, ceea ce înseamnă că celelalte fire de execuţie nu vor mai putea accesa resursele critice. Deoarece mai multe secţiuni critice (metode sincrone) ale unui obiect gestionează o singură resursă critică. În exemplul nostru, atunci când producătorul apelează metoda **`put`** pentru a scrie un număr, va bloca tot obiectul de tip **`Buffer`**, prin urmare firul de execuţie consumator nu va avea acces la cealaltă metodă sincronă **`get`**, şi reciproc.

```
public synchronized void put(int number) {
        // buffer blocat de producător
        ...
        // buffer deblocat de producător
}
public synchronized int get() {
        // buffer blocat de consumator
        ...
```

```
        // buffer deblocat de consumator
    }
```

2.2. Metodele `wait, notify şi notifyAll`

Obiectul de tip **`Buffer`** din exemplu de mai sus are o variabilă membră privată numită **`number`**, în care este memorat numărul pe care îl comunică producătorul şi din care îl preia consumatorul. Variabilă privată logică **`available,`** care indică starea buffer-ului: dacă are valoarea **`true`** înseamnă că producătorul a plasat o valoare în buffer şi consumatorul nu a preluat-o încă; dacă este **`false`**, consumatorul a preluat valoarea din buffer, dar producătorul nu a plasat alta în buffer. Metodele clasei **`Buffer`** sunt:

```
public synchronized int get() {
       if (available) {
              available = false;
              return number;
       }
}
public synchronized int put(int number) {
       if (!available) {
             available = true;
       this.number = number;
       }
}
```

Realizarea acestor metode nu va genera un rezultat corect, deoarece firele de execuţie, nu sincronizează accesul la buffer. Situaţiile în care metodele **`get`** şi **`put`** nu fac nimic, vor duce la pierdereaunor numere de către consumator, sau preluarea aceluieşi număr de două ori. Aşadar, cele două fire de execuţie trebuie să se aştepte unul pe celălalt.

```
public synchronized int get() {
       while (!available) {
//nimic - aştept ca variabila să devină true
       }
       available = false;
       return number;
}
public synchronized int put(int number) {
       while (available) {
//nimic - aştept ca variabila să devină false
       }
      available = true;
      this.number = number;
}
```

Programul nu este corect, deoarece cele două metode îşi aşteaptă în mod ”egoist” condiţia de terminare. Ca urmare, corectitudinea funcţionării va depinde de sistemul de operare, ceea ce reprezintă o greşeală de programare. Punerea corectă a unui fir de execuţie în aşteptare se realizează cu metoda `wait` a clasei `Thread`, care are trei forme:

```
void wait( )
void wait( long timeout )
void wait( long timeout, long nanos ).
```

După apelul metodei **wait**, firul de execuţie curent eliberează monitorul asociat obiectului respectiv şi aşteaptă ca una din următoarele condiţii să fie îndeplinită:

- un alt fir de execuţie informează pe cei care "aşteaptă" la un anumit monitor să se "trezească"; acest lucru se realizează printr-un apel al metodei **notifyAll** sau **notify**.
- perioada de aşteptare specificată a expirat.

Metoda **wait** poate produce excepţii de tipul `InterruptedException`, atunci când firul de execuţie este în starea de aşteptare -`Not Runnable` este întrerupt din aşteptare şi trecut forţat în starea `Runnable`. Metoda `notifyAll` informează toate firele de execuţie, care sunt în aşteptare la monitorul obiectului curent, îndeplinirea condiţiei pe care o aşteptau. Metoda `notify` informează doar un singur fir de execuţie.Varianta corectă a metodelor `get` şi `put`este:

```
public synchronized int get() {
   while (!available) {
      try {
         wait();
//aşteaptă producătorul să pună o valoare
         } catch (InterruptedException e) { }
   }
   available = false;
   notifyAll();
   return number;
}

public synchronized void put(int number) {
   while (available) {
     try {
      wait();
//aşteaptă consumatorul să preia valoarea
         } catch (InterruptedException e) { }
    }
  this.number = number;
  available = true;
  notifyAll();
}}
```

2.3 Bariere

În aplicaţiile multithreading este necesar, ca anumite thread-uri să se sincronizeze la un anumit punct. Un exemplu este calculul paralel în faza, în care toate thread-urile trebuie să-şi termine faza de execuţie înainte de a trece, toate concomitent la faza următoare. O **barieră**este un mecanism folosit pentru a sincroniza mai multe thread-uri. Un thread care întâlneşte o barieră intră automat în `wait()`. Când ultimul thread "ajunge" la barieră, semnalează (`notify()`) celorlalte thread-uri, care sunt în aşteptare, rezultând o "trecere" în grup a barierei. Iată un exemplu în acest sens:

```
import java.util.*;
class Barrier { // Clasa Barrier sincronizează toţi
//participanţii private
   int ParticipatingThreads;
   private int WaitingAtBarrier;
   public Barrier(int num){ //Constructorul
    ParticipatingThreads = num;
```

```
        WaitingAtBarrier=0;
    }
     public synchronized void Reached(){
        //Metoda bariera
        WaitingAtBarrier++;
       if(ParticipatingThreads != WaitingAtBarrier){
        //Inseamna ca thread-ul nu este ultimul
          try {
               wait(); //Thread-ul este oprit până ce
                       //este eliberat
              } catch (InterruptedException e) { }
       } else {  // Acesta a fost ultimul thread activ
           notifyAll();
           WaitingAtBarrier=0;//le eliberează pe toate
      }
   }
}
```

Ajungând la barieră toate thread-urile, în fară de ultimul, asteaptă în interiorul metodei synchronized ca ultimul thread ajuns să le ”trezească”. După ce s-au "trezit" un fir de execuție preia monitorul. Celelalte thread-uri continuă să concureze pentru monitor. Numărul de thread-uri participante trebuie să fie cunoscut. În caz contrar se folosește un mecanism de înregistrare a participanţilor.

Lucrare de laborator nr. 4

Tema lucrării: ***Sincronizarea thread-urilor în Java.***

Scopul lucrării:

Scopul principal al acestei lucrări este studierea și aplicarea mecanismelor de sincronizare a firelor de execuție (thread-urilor) prin implementarea problemei clasice producător–consumator, utilizând diverse metode și clase ale limbajului pentru controlul accesului concurent la resurse partajate.

Obiectivele lucrării:

1. Înțelegerea conceptului de concurență și sincronizare în cadrul aplicațiilor multi-threaded.
2. Implementarea unei resurse partajate (buffer comun) între producători și consumatori.
3. Aplicarea metodelor de sincronizare precum `wait()`, `notify()`, `notifyAll()` și blocurile `synchronized`.
4. Dobândirea abilității de a proiecta și testa aplicații multi-threaded stabile și scalabile.

Exemplu de realizare:

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) throws InterruptedException {

        Store store=new Store();
```

```
        Producer p1 = new Producer(store);
        p1.setDaemon(true);
        p1.setName("Producator №1");

        Producer p2 = new Producer(store);
        p2.setDaemon(true);
        p2.setName("Producator №2");

        Producer p3 = new Producer(store);
        p3.setDaemon(true);
        p3.setName("Producator №3");

        Consumer c1 = new Consumer(store);
        c1.setName("Consumator №1");

        Consumer c2 = new Consumer(store);
        c2.setName("Consumator №2");

        Consumer c3 = new Consumer(store);
        c3.setName("Consumator №3");

        Consumer c4 = new Consumer(store);
        c4.setName("Consumator №4");

        p1.start();
        p2.start();
        p3.start();
        c1.start();
        c2.start();
        c3.start();
        c4.start();
        while(c1.isAlive() || c2.isAlive() || c3.isAlive() || c4.isAlive()){ }
        System.out.println("Toate thread-uri au finalizat");
    }
}

class Store {

    ArrayList<Integer> stockList=new ArrayList<Integer>();

    public synchronized void get(String str) {
        while (stockList.size()<1) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        System.out.println(str + " a luat din depozit: "+
stockList.get(stockList.size()-1));
        stockList.remove(stockList.size()-1);
        if(stockList.size()!=0){
            System.out.print("Depozitul are " + stockList.size()+" numere -> ");
            for(int impar : stockList){
                System.out.print(impar+ " ");
            }
            System.out.println(" ");}
        else{
            System.out.println("Depozitul este gol");
```

```
        }
        notifyAll();
    }

    public synchronized void put(String str,int a, int b) {
        while (stockList.size()>=5) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        System.out.print(str + " a pus in dipozit doua numere: ");
        stockList.add(a);
        System.out.print(stockList.get(stockList.size()-1)+", ");
        stockList.add(b);
        System.out.println(stockList.get(stockList.size()-1));
        if(stockList.size()!=0){
            System.out.print("Depozitul are " + stockList.size()+" numere -> ");
            for(int impar : stockList){
                System.out.print(impar+ " ");
            }
            System.out.println(" ");}
        else{
            System.out.println("Depozitul este gol");
        }
        notifyAll();
    }
}

class Producer extends Thread{

    Store s;

    public Producer(Store s) {
        this.s = s;
    }
    @Override
    public void run() {
        int[] impare = new int[]{1,3,5,7,9,11,13,15,17,19};
        while(true){
            s.put(getName(), impare[(int)(Math.random()*9)],
impare[(int)(Math.random()*9)]);
        }
    }
}

class Consumer extends Thread{

    Store s;
    public Consumer(Store s) {
        this.s = s;
    }
    @Override
    public void run() {
        int cons = 0;
        for(int i = 0; i < 2; i++){
            s.get(getName());
        }
```

```
            System.out.println(getName() + " a luat 2 numere. Thread-ul a finalizat");
        }
}
```

Probleme propuse spre realizare:

Sunt dați X producători care generează aleatoriu F obiecte care sunt consumate de Y consumatori. De afişat informaţia despre producerea şi consumarea obiectelor, mesajele despre cazurile când "depozitul e gol sau plin". Toate operaţiile se efectuează până când fiecare consumator este îndestulat cu Z obiecte.

Dimensiunea depozitului este D. Valorile pentru X, Y, Z, D, F sunt indicate în Tabelul 3.

Tabelul 1 Variantele pentru realizarea sarcinii

Nr.	X	Y	Z	D	Tip Obiecte
1	2	3	11	8	Numere pare
2	3	4	2	5	Numere impare
3	4	3	3	10	Vocale
4	3	2	12	11	Consoane
5	2	5	3	12	Numere pare
6	2	4	4	7	Numere impare
7	3	3	5	6	Vocale
8	4	2	4	5	Consoane
9	5	2	3	4	Numere pare
10	3	3	5	2	Numere impare
F - fiecare producător produce câte 2 obiecte de fiecare dată pentru toate variantele					

Criterii de evaluare:

1. Crearea şi iniţializarea thread-urilor pentru realizarea sarcinilor.
2. Alegerea formelor de sincronizare a firelor.
3. Sincronizarea thread-urilor cu formele potrivite.
4. Crearea interfeței grafice a programului.
5. **Corectitudinea codului** - verificarea dacă codul este corect, fără erori de funcționare.
6. **Respectarea instrucțiunilor și cerințelor** - verificarea corectitudenii cerințelor sarcinii, cum ar fi numărul de fire de execuție.
7. **Optimizarea codului** - Evaluarea eficienței codului în utilizarea resurselor și evitarea codului.
8. **Respectarea termenului de susținere** - evaluarea punctajului în funcție de punctualitate, dacă lucrarea a fost predată în termenul stabilit.
9. **Evaluarea cunoștințelor** - explicațiile oferite despre procesul de realizare a lucrării, ceea ce poate include descrierea funcțiilor principale și a logicii utilizate.
10. Utilizarea de către student a IA.

Pentru obținerea notei 5 - 6 sunt obligatorii criteriile 1,2,3,5,8,9

Pentru obținerea notei 7 - 8 sunt obligatorii criteriile 1,2,3,4,5,6,8,9

Pentru obținerea notei 9 - 10 sunt obligatorii criteriile 1-9

Dacă a fost utilizat criteriul 10 nota este scăzută cu 2 baluri, numai dacă studentul sa lămurit în funcționarea codului. În caz contrar lucrare de laborator nu este susținută.

***Întrebări de verificare*:**

1.Ce este un thread (fir de execuție) și cum se creează în Java?
2. Ce reprezintă sincronizarea thread-urilor și de ce este necesară?
3. Care sunt metodele principale de sincronizare oferite de clasa `Object` în Java?
4. Explicați rolul metodelor `wait()`, `notify()` și `notifyAll()`.
5. Ce este un *race condition* și cum poate fi prevenit?
6. În ce situații pot apărea *deadlock* și *starvation*?
7. Cum se poate monitoriza și testa comportamentul concurent al unui program Java?