Introducere teoretică despre firele de execuție și sincronizare în Java

Un **fir de execuție** (*thread*) reprezintă o secvență de execuție independentă în cadrul unui proces, rulând în paralel (sau cvasi-paralel, pe un singur procesor) cu alte fire. Java suportă multithreading-ul la nivel de limbaj, permițând crearea de fire fie prin moștenirea clasei Thread, fie prin implementarea interfeței Runnable. Lansarea unui fir nou de execuție se face prin metoda start(), care va apela intern metoda run() a thread-ului creat, executându-l concurent cu firul curent.

Atunci când mai multe thread-uri rulează în același program, pot apărea situații în care acestea *interacționează* sau *comunică* între ele. În mod particular, dacă împart **resurse comune** (variabile, obiecte, fișiere etc.), există riscul ca operațiile lor intercalate să ducă la inconsistența datelor. Comutarea de context între thread-uri se poate produce în orice moment, astfel încât două fire pot încerca să acceseze simultan aceeași resursă, corupându-i starea[biblioteca.utcluj.ro](). Pentru a preveni aceste condiții de cursă (*race conditions*), este necesar un mecanism prin care accesul concurent la resurse partajate să fie controlat – adică să se asigure **excluderea mutuală** a firelor din secțiunile critice de cod[biblioteca.utcluj.ro](). O **secțiune critică** este un segment de cod care manipulează o resursă comună și care nu trebuie executat de mai mult de un fir de execuție simultan.

Java oferă un mecanism de bază pentru sincronizare sub forma **monitorului**. În Java, orice obiect are asociat un monitor intern, care acționează ca un *lock* și o *coadă de așteptare* pentru fire[biblioteca.utcluj.ro](). Când un thread intră într-o secțiune critică marcată cu synchronized, el trebuie să obțină monitorul obiectului respectiv (”ia lacătul”). Pe durata deținerii monitorului de către un fir, niciun alt thread nu mai poate intra în secțiunile sale critice sincronizate pe același obiect – eventualele thread-uri care solicită intrarea vor fi blocate automat, așteptând eliberarea monitorului[else.fcim.utm.mdelse.fcim.utm.md](). În momentul ieșirii din secțiunea critică, monitorul este eliberat (“lacătul” este descuiat), permițând altui fir accesul în resursa critică[else.fcim.utm.md]().

Pe lângă excluderea reciprocă, adesea este necesară și **sincronizarea condițională** a activităților firelor. Asta înseamnă că un fir producător trebuie să poată *semnaliza* unui fir consumator că datele au fost produse, iar consumatorul să *aștepte* până ce are date disponibile de procesat, și invers – consumatorul poate semnaliza că a consumat o resursă, permițând producătorului să continue. Java suportă acest tip de coordonare prin metodele wait() și notify()/notifyAll(), care fac parte din API-ul oricărui obiect (din clasa de bază Object). Un fir poate apela wait() pe monitorul unui obiect sincronizat pentru a elibera lacătul și a intra în așteptare, până când un alt fir (deținând același monitor) va apela notify() sau notifyAll() pentru a trezi firele aflate în așteptare. Acest mecanism stă la baza comunicării între fire în modelul monitoarelor[else.fcim.utm.md]().

În rezumat, platforma Java pune la dispoziție mai multe instrumente pentru sincronizare: de la cuvântul cheie synchronized și metodele wait/notify inerente oricărui obiect, până la API-ul extins din pachetul java.util.concurrent (locks, conditii, cozi blocante, etc.). În secțiunile următoare vom descrie problema producător-consumator și vom explora diferite soluții de sincronizare a firelor pentru a o implementa corect.

4. Descrierea problemei Producător-Consumator

Problema producător-consumator este un exemplu clasic de sincronizare și comunicare între fire de execuție. Scenariul presupune existența a două tipuri de fire concurente care partajează o resursă comună (de obicei un buffer): un fir (sau grup de fire) *Producător* care generează în mod repetat date și le plasează în buffer, și un fir (sau grup de fire) *Consumator* care preia aceste date din buffer și le procesează[else.fcim.utm.md](). De exemplu, un producător poate genera un șir de

numere sau mesaje, iar consumatorul le citește pe rând și le afișează sau prelucrează. Atât producătorul cât și consumatorul operează permanent (teoretic, într-un buclă infinită): producătorul *produce* noi elemente, iar consumatorul *consumă* elementele produse.

Principala provocare în această problemă este coordonarea accesului la **bufferul comun**. Bufferul are o anumită capacitate (poate fi minim 1 element, sau un buffer limitat de dimensiune fixă). Pentru a menține *consistența datelor* și a evita erorile, trebuie respectate următoarele reguli de sincronizare:

- **Excluderea reciprocă la accesul bufferului:** La un moment dat, doar un fir (producător sau consumator) trebuie să manipuleze bufferul. Cu alte cuvinte, operațiile de depunere și extragere în/din buffer constituie secțiuni critice care nu trebuie executate simultan. Această cerință asigură integritatea structurii de date a bufferului (de exemplu, prevenind citirea și scrierea concomitentă pe aceleași poziții)mobylab.docs.crescdi.pub.ro.
- **Consumatorul așteaptă dacă bufferul este gol:** Dacă nu există niciun element disponibil (bufferul este gol), firele consumatoare trebuie să se **blocheze** până când un producător produce un element nou. În absența acestei așteptări, un consumator nesincronizat ar încerca să citească date inexistente, ducând la rezultate invalide (sau a prelua același element de mai multe ori)else.fcim.utm.mdmobylab.docs.crescdi.pub.ro.
- **Producătorul așteaptă dacă bufferul este plin:** Dacă bufferul are o capacitate limitată și este deja plin (nu mai există spațiu pentru elemente noi), un fir producător trebuie să se **suspende** până când un consumator extrage cel puțin un element, eliberând spațiu. Fără această sincronizare, un producător nesincronizat ar suprascrie date încă neconsumate sau ar pierde elemente (dacă bufferul suprascrie vechile valori)else.fcim.utm.mdmobylab.docs.crescdi.pub.ro.
- **Fără pierdere de date și fără interblocaje:** Soluția trebuie să garanteze că **toate elementele produse sunt în final consumate o dată** (nu se pierd și nu se duplică valori) și că firele nu ajung într-o stare de așteptare reciprocă permanentă (*deadlock*). Consumatorii nu trebuie să încerce niciodată să consume dintr-un buffer gol, iar producătorii să nu continue producerea în buffer plin – aceste condiții fiind asigurate de mecanismele de sincronizare menționate mai sushabr.com. Totodată, dacă sincronizarea este corect realizată (de ex. folosind notificări adecvate), **deadlock-ul** este evitat, iar sistemul poate rula indefinit, producând și consumând date fără să se blocheze complet.

Problema producător-consumator ilustrează necesitatea sincronizării atât pentru excludere mutuală, cât și pentru comunicare între fire. În continuare, vom prezenta diferitele mecanisme de sincronizare oferite de Java și vom demonstra implementarea problemei P-C folosind aceste mecanisme.

5. Prezentarea claselor de sincronizare Java utilizate

Java pune la dispoziție mai multe **mecanisme de sincronizare** a execuției concurente, de la cele încorporate în limbaj până la utilitare din bibliotecile de nivel înalt. În această lucrare ne vom concentra pe patru abordări principale: utilizarea cuvântului cheie synchronized (monitoare interne), utilizarea clasei ReentrantLock (lacăt reentrant explicit) împreună cu interfața Condition (variabile de condiție), și utilizarea colecțiilor de tip BlockingQueue din pachetul java.util.concurrent. În continuare, prezentăm pe scurt fiecare dintre aceste mecanisme și modul în care contribuie la sincronizarea firelor în contextul problemei producător-consumator.

synchronized (monitoare interne Java): Cuvântul cheie synchronized este mecanismul fundamental prin care Java asigură excluderea mutuală pe secțiunile critice. Un bloc sau o metodă declarată synchronized este protejată de *monitorul* unui anumit obiect. Doar un singur fir poate deține monitorul obiectului la un moment dat, astfel încât alte fire care încearcă să intre în secțiunile sincronizate pe același obiect vor fi blocate până la eliberarea monitoruluielse.fcim.utm.mdelse.fcim.utm.md. Monitorul acționează ca un *lock* exclusiv asociat resursei comune, împiedicând accesul simultan al firelor la aceasta. De exemplu, în implementarea noastră a bufferului comun, vom marca metodele put și get ca fiind synchronized, pentru a preveni interferența producătorului și consumatorului la citirea și scrierea valorilorelse.fcim.utm.mdelse.fcim.utm.md.

Pe lângă blocarea accesului, monitoarele Java oferă și mecanisme de **așteptare și notificare** prin intermediul metodelor din clasa Object: wait(), notify() și notifyAll(). Orice fir care deține monitorul unui obiect sincronizat poate apela wait() pentru a elibera temporar monitorul și a intra în așteptare până la apariția unui anumit eveniment (o condiție dorită). Un alt fir, după ce realizează acțiunea așteptată, poate apela notify() (pentru a trezi un singur fir aflat în așteptare) sau notifyAll() (pentru a trezi toate firele care așteaptă pe acel monitor). În contextul problemei P-C, producătorul, după ce plasează un element în buffer, va apela notifyAll() pentru a trezi eventualul consumator blocat să aștepte date noi. Invers, consumatorul, după ce extrage un element, poate notifica producătorul că s-a eliberat loc în buffer. Folosirea wait() și notifyAll() asigură coordonarea celor două fire astfel încât să se respecte condițiile de așteptare (buffer plin/gol) menționate mai suselse.fcim.utm.mdelse.fcim.utm.md. (Notă: se preferă notifyAll() în loc de notify() în acest tip de problemă pentru a evita **înfundarea** în așteptare a firului rămas – de exemplu, dacă există mai mulți producători/consumatori, notifyAll() se asigură că toți cei care pot acționa sunt treziți.)

ReentrantLock (lacăt reentrant explicit): Clasa ReentrantLock face parte din pachetul java.util.concurrent.locks și oferă o alternativă mai flexibilă la mecanismul integrat al monitoarelor. Un ReentrantLock se comportă similar cu un lock implicit (monitor) asociat cu synchronized, dar expune metode explicite pentru a **bloca** (lock()) și **debloca** (unlock()) resursaofficialcto.com. Programatorul are astfel un control mai fin asupra momentului și duratei blocării, putând de exemplu să încerce preluarea lock-ului cu timp de așteptare (tryLock() cu timeout) sau să specifice o politică de echitate (*fairness*) la acordarea lock-ului. Numele *reentrant* indică faptul că același fir poate dobândi de mai multe ori același lock (intrând recursiv în secțiuni critice protejate de același lacăt) fără a rămâne blocat – trebuie însă să elibereze lock-ul de tot atâtea ori pe câte l-a dobândit.

Un avantaj al lui ReentrantLock este **scalabilitatea mai bună** în scenariile cu multe thread-uri, oferind performanțe îmbunătățite și caracteristici suplimentare față de synchronizedbiblioteca.utcluj.ro. De exemplu, se pot folosi *lock-uri echitabile* (opțiunea de a crea lock-ul în modul *fair*, care asigură că firele sunt servite în ordinea în care solicită lacătul, evitând *starvation*-ul). De asemenea, ReentrantLock permite implementarea unor scheme de blocare neblocantă sau cu timp limitat (metoda tryLock() poate returna imediat dacă lock-ul e ocupat, sau poate aștepta un interval limitat)biblioteca.utcluj.ro. Totuși, utilizarea lacătelor explicite impune o atenție sporită din partea programatorului: este **obligatoriu** ca orice apel lock() să fie urmat de un apel corespunzător unlock() (de obicei plasat într-un bloc finally) pentru a evita interblocarea permanenta a resursei. Vom exemplifica utilizarea lui ReentrantLock în rezolvarea problemei P-C, arătând cum putem obține același comportament ca în varianta cu synchronized.

Condition (variabilă de condiție): Interfața Condition din pachetul java.util.concurrent.locks lucrează împreună cu un Lock și reprezintă echivalentul obiectual al mecanismului wait/notify.

Practic, o instanță de Condition asociată unui ReentrantLock oferă metode precum await() (echivalentă cu wait(), care eliberează lock-ul și suspendă firul curent până la semnal) și signal()/signalAll() (echivalente cu notify()/notifyAll(), pentru a trezi firele în așteptare pe acea condiție)biblioteca.utcluj.ro. Diferența față de monitoarele obișnuite este că putem avea **mai multe obiecte Condition pentru același lock**, permițând grupuri diferite de fire să aștepte diferite condițiics.ubbcluj.ro. În problema noastră, acest lucru este util pentru a separa situațiile de așteptare: de exemplu, putem crea o condiție notEmpty pe care să o aștepte consumatorii când bufferul e gol și o condiție notFull pe care să o aștepte producătorii când bufferul e plin. Astfel, un fir care produce poate da signal() pe condiția notEmpty când pune un element (trezind un eventual consumator), iar un consumator poate da signal() pe notFull când scoate un element (trezind un eventual producător). Utilizarea Condition oferă o **granularitate mai fină** a sincronizării condiționale comparativ cu un singur monitor unde toate firele așteaptă pe același set de condiții. Pentru a folosi Condition, se creează obiectul prin apelul lock.newCondition() pe un ReentrantLock existent. Vom vedea în exemplul practic cum Condition ajută la exprimarea clară a logicii de așteptare în producător-consumator.

BlockingQueue (coadă blocantă): Interfața BlockingQueue (din pachetul java.util.concurrent) definește o structură de date de tip coadă thread-safe, concepută special pentru scenarii producător-consumator. O coadă blocantă implementează operațiile de inserare și extragere în mod **blocant**: dacă un fir încearcă să extragă din coadă când aceasta este goală, firul va fi blocat automat până când un element devine disponibil; similar, dacă un fir încearcă să insereze într-o coadă plină (în cazul cozii cu capacitate limitată), firul producător va fi blocat până când există spațiu liberbiblioteca.utcluj.ro. Acest comportament este oferit implicit, deci programatorul nu trebuie să gestioneze explicit wait()/notify – ele sunt implementate intern de către metodologiile BlockingQueue. Există mai multe implementări disponibile, cum ar fi ArrayBlockingQueue (coadă cu capacitate fixă, bazată pe tablouri), LinkedBlockingQueue (coadă cu capacitate opțională, bazată pe listă legată), PriorityBlockingQueue (coadă blocantă ce serveste elementele pe baza unei priorități) sau SynchronousQueue (o coadă specială fără capacitate, unde fiecare inserare blochează până este preluată)biblioteca.utcluj.ro.

Utilizarea unei cozi blocante simplifică foarte mult rezolvarea problemei producător-consumator, deoarece nu mai trebuie să scriem manual cod de sincronizare condițională – blocarea și deblocarea firelor se realizează automat în spatele scenelor. Astfel, BlockingQueue poate fi privită ca un *monitor specializat* pentru transferul de date între fire. După cum vom arăta în exemplul practic, un producător poate pur și simplu să apeleze metoda put(element) pe coadă, iar un consumator metoda take(), știind că aceste apeluri vor aștepta dacă operația nu poate fi efectuată imediat. În plus, aceste clase sunt bine optimizate și testate, ceea ce conduce la implementări mai **fiabile** și mai **curate** ale codului concurentbaeldung.com. Practic, în loc să ne ocupăm noi de variabile de stare precum available sau de apeluri wait/notify, folosim BlockingQueue care *știe* să blocheze producătorii până când există spațiu și să blocheze consumatorii până când există elemente de consumat.

Pentru a demonstra sincronizarea firelor în contextul problemei producător-consumator, vom prezenta trei variante de implementare, fiecare folosind un alt mecanism de sincronizare. În toate variantele, vom considera un **buffer comun** în care un fir **Producător** introduce elemente (numere întregi), iar un fir **Consumator** le extrage. Producătorul va genera, de exemplu, numere de la 1 la 10 (cu o mică întârziere între ele pentru a simula lucru real), iar consumatorul le va prelua și afișa. Scopul este ca, prin sincronizare, consumatorul să nu încerce să citească atunci când nu sunt date disponibile, iar producătorul să nu scrie date noi peste altele neconsumate încă.

Soluție 1: Utilizarea ReentrantLock și Condition

Prima soluție implementează ReentrantLock în loc de synchronized, împreună cu două condiții: una pentru "buffer nu e gol" (notEmpty) și alta pentru "buffer nu e plin" (notFull). Deși în exemplul nostru bufferul are capacitate 1 (similar cu soluția precedentă), vom folosi totuși două obiecte Condition separate pentru a ilustra avantajul acestui mecanism – în cazul extinderii la un buffer cu mai multe sloturi, două condiții ne permit să semnalăm selectiv consumatorii sau producătorii, în loc să trezim toate firele de fiecare dată. Codul este prezentat mai jos:

```
// Soluția 1: utilizarea ReentrantLock cu Condition

import java.util.concurrent.locks.*;

class BufferLock {
   private final Lock lock = new ReentrantLock();
   private final Condition notEmpty = lock.newCondition();
   private final Condition notFull  = lock.newCondition();
   private int valoare = -1;
   private boolean disponibil = false;


   public void put(int x) throws InterruptedException {
      lock.lock();                  // obține lacătul explicit
      try {
         while (disponibil) {
            notFull.await();        // așteaptă până se eliberează bufferul
         }
         valoare = x;
         disponibil = true;
         System.out.println("Producătorul a pus: " + x);
         notEmpty.signal();         // semnalizează că bufferul nu mai este gol
      } finally {
         lock.unlock();             // eliberează lacătul în bloc finally
      }
   }


   public int get() throws InterruptedException {
      lock.lock();
```

```
        try {
            while (!disponibil) {
                notEmpty.await();      // așteaptă până există o valoare de consumat
            }
            int rezultat = valoare;
            disponibil = false;
            System.out.println("Consumatorul a luat: " + rezultat);
            notFull.signal();          // semnalizează că s-a eliberat un slot în buffer
            return rezultat;
        } finally {
            lock.unlock();
        }
    }
}

public class TestLockCondition {
    public static void main(String[] args) {
        BufferLock buf = new BufferLock();
        // Creăm un fir producător folosind un Runnable
        Runnable prodTask = () -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    buf.put(i);
                    Thread.sleep(50);
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        };
        // Creăm un fir consumator folosind un alt Runnable
        Runnable consTask = () -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    buf.get();
```

```
                Thread.sleep(100);
            } catch (InterruptedException e) { e.printStackTrace(); }
        }
    };
    // Pornim firele
    new Thread(prodTask).start();
    new Thread(consTask).start();
  }
}
```

Explicații: În această variantă, BufferLock folosește un obiect ReentrantLock numit lock pentru a proteja secțiunile critice. Metodele put și get nu mai sunt declarate synchronized, ci în schimb apelăm explicit lock.lock() la început și lock.unlock() la final (în blocul finally, pentru a ne asigura că deblocarea se întâmplă chiar și în caz de excepții). Condițiile notEmpty și notFull sunt create prin lock.newCondition(). Ele reprezintă cozi de așteptare separate asociate aceluiași lacăt:

- notFull este folosită de producător pentru a aștepta cât timp bufferul este plin. În cod, dacă disponibil == true, producătorul apelează notFull.await(), eliberând lacătul și blocându-se până când un alt fir va semnala această condiție. Când consumatorul extrage o valoare, setează disponibil = false și apelează notFull.signal(), ceea ce va trezi unul dintre firele aflate în așteptare pe notFull (în cazul nostru, producătorul).
- notEmpty este folosită simetric de consumator. Dacă disponibil == false (buffer gol), consumatorul face notEmpty.await(), suspendându-se până ce un producător pune o valoare și apelează notEmpty.signal().

Observăm că logica de așteptare și semnalizare este foarte asemănătoare cu cea din varianta anterioară, doar că în loc de wait()/notifyAll() pe monitor folosim await()/signal() pe obiecte de condiție. Un avantaj aici este că putem trezi *exact* tipul de fir care are nevoie de a fi trezit: de exemplu, când bufferul devine ne-gol, semnalăm notEmpty (trezind un consumator, nu și alți producători care ar rămâne tot blocați deoarece condiția lor notFull e încă falsă). Astfel se evită trezirea inutilă a unor fire care apoi constată că nu pot acționa și intră iar în așteptare. În exemplul cu un singur producător și consumator diferența nu este vizibilă, dar în scenarii cu mai multe fire această separare a "wait-set"-urilor crește eficiența sincronizăriics.ubbcluj.ro.

Ca și în cazul precedent, folosim bucle while pentru a verifica condițiile înainte de așteptare, și tratăm întreruperile posibile ale thread-urilor (aruncând InterruptedException din metodele put/get către apelant). Am folosit aici sintaxa cu expresii lambda pentru a crea thread-urile producător și consumator (Runnable prodTask și consTask), acestea având comportament echivalent cu clasele Producator și Consumator din prima soluție.

Rulare și rezultat: Programul TestLockCondition va produce, de genul intercalat Producătorul a pus X / Consumatorul a luat X pentru X de la 1 la 10, demonstrând că și cu ReentrantLock coordonarea este corectă. Din punct de vedere al funcționalității, rezultatul este același. Intern, însă, abordarea cu lock explicit oferă posibilitatea extinderii mai facile la mai multe fire sau la buffere cu mai multe sloturi. De exemplu, dacă am modifica bufferul să conțină un vector circular de dimensiune N > 1, am putea ajusta condițiile while să verifice count == N (buffer plin) sau

count == 0 (buffer gol), am menține două Condition (notFull/notEmpty), iar semnalizările ar rămâne la fel. ReentrantLock și Condition ne permit să implementăm această **problemă a bufferului limitat** într-un mod foarte clar (practic reproducând manual ceea ce oferă și clasa următoare, BlockingQueue, dar cu flexibilitate și control sporit asupra detaliilor). Totodată, ReentrantLock ne-ar permite și încercarea de a obține lock-ul cu timeout, dacă am dori (de exemplu, un producător ar putea renunța să mai producă dacă nu reușește să obțină lock-ul într-un anumit timp) – astfel de funcționalități nu sunt posibile cu synchronized simplu[biblioteca.utcluj.ro](biblioteca.utcluj.ro).

Soluție 2: Utilizarea unei cozi blocante (BlockingQueue)

Cea de-a doua abordare folosește o abstracție de nivel înalt din Java: colecțiile de tip BlockingQueue. Acestea încorporează deja mecanismele de sincronizare necesare, ceea ce ne permite să implementăm producătorul și consumatorul **fără a scrie codul de blocare și notificare manual**. Vom utiliza ArrayBlockingQueue<Integer> – o coadă blocantă cu capacitate fixă (vom alege o capacitate, de exemplu 5, pentru ilustrare). Producătorul va folosi metoda put() a cozii, iar consumatorul metoda take(). Ambele metode aruncă InterruptedException și blochează firul curent dacă operația nu poate fi realizată instant: put() blochează dacă coada este plină, iar take() blochează dacă este goală. Iată implementarea:

```
// Soluția 2: utilizarea unei cozi blocante

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;


public class TestBlockingQueue {

   public static void main(String[] args) {

      BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5); // buffer cu 5 poziții

      // Firul producător

      Thread producer = new Thread(() -> {

         for (int i = 1; i <= 10; i++) {

            try {

               queue.put(i);  // blochează dacă coada e plină

               System.out.println("Producătorul a pus: " + i);

               Thread.sleep(50);

            } catch (InterruptedException e) {

               Thread.currentThread().interrupt();

            }

         }

      });

      // Firul consumator
```

```
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    int val = queue.take();  // blochează dacă coada e goală
                    System.out.println("Consumatorul a luat: " + val);
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        producer.start();
        consumer.start();
    }
}
```

Explicații: Am creat o coadă blocantă queue cu capacitatea 5. Aceasta va acționa ca buffer partajat între firul producer și firul consumer. Observați că nu mai este nevoie de niciun bloc synchronized, de niciun lock sau condiție explicită: metodele queue.put() și queue.take() se ocupă de sincronizare. Conform specificației BlockingQueue, dacă producer apelează put(x) iar coada este deja plină (are 5 elemente neconsumate), atunci firul producător va **aștepta blocat** până când consumer extrage ceva (eliberează spațiu)biblioteca.utcluj.ro. Invers, dacă consumer apelează take() pe o coadă goală, firul consumator va aștepta până când producătorul inserează un element și îl va trezi. Această funcționalitate este asigurată intern de ArrayBlockingQueue prin utilizarea propriilor lock-uri și condiții. Practic, ArrayBlockingQueue are implementat intern un mecanism foarte similar cu cel din soluția 2 (folosește două condiții, notEmpty și notFull, și o blocare intrinsecă)officialcto.comofficialcto.com. Pentru programator, însă, detaliile sunt ascunse, el având la dispoziție o interfață simplă de tip put/take.

Codul de mai sus pornește cele două thread-uri și acestea se vor sincroniza automat prin coadă. Putem folosi direct Thread.sleep() pentru a simula timpi diferiți de producție/consum, știind că blocările necesare sunt gestionate de coadă. Avantajele unei astfel de soluții sunt claritatea și siguranța: posibilitatea de a uita să faci un notify() sau de a folosi greșit condițiile dispare, reducând semnificativ complexitatea. De asemenea, soluția generalizează ușor – dacă dorim să avem mai mulți producători și mai mulți consumatori, este suficient să lansăm mai multe thread-uri care folosesc aceeași coadă. Toți producătorii vor fi blocați automat dacă coada e plină (până când un consumator face loc), și toți consumatorii vor aștepta automat dacă nu sunt elemente (până când un producător adaugă ceva)biblioteca.utcluj.ro. Acest model de programare este unul **coopertant** și thread-safe by design, oferit de bibliotecile Java.

Rulare și rezultat: La rularea programului TestBlockingQueue, se va observa, similar, o alternanță coerentă a mesajelor "Producătorul a pus X" și "Consumatorul a luat X". Ordinea exactă

poate diferi, însă la fel ca înainte, niciun consumator nu va încerca să consume din gol (dacă consumatorul ajunge în buclă înaintea producătorului, se va bloca în take() până apare un element) și niciun producător nu va umple excesiv bufferul (dacă producătorul este mai rapid și umple cele 5 locuri, la a 6-a inserare va sta blocat în put() până când consumatorul preia ceva). Astfel, invariantele problemei P-C sunt menținute automat prin BlockingQueue. Această abordare evidențiază puterea abstractizărilor din java.util.concurrent: dezvoltatorul poate rezolva corect problema concurentă fără să gestioneze direct condițiile de sincronizare, ceea ce duce la un cod mai concis și mai puțin predispus la eroribaeldung.com.

Concluzii

În această lucrare de laborator am explorat diferite metode de sincronizare a firelor de execuție în Java aplicate problemei **Producător-Consumator**. Am implementat o soluție echivalentă folosind lacăte explicite (ReentrantLock) și variabile de condiție (Condition), iar în final am recurs la o structură de date concurentă de nivel înalt (BlockingQueue) ce simplifică considerabil problema.

Toate cele trei soluții duc la același comportament corect: producătorul și consumatorul se coordonează astfel încât să nu piardă date, să nu intre în conflict și să nu rămână blocați permanent. Cu toate acestea, există diferențe notabile între abordări din perspectiva complexității implementării și a flexibilității:

- Soluția cu ReentrantLock și Condition este puțin mai **verbosă**, însă oferă un control mai fin. Am putut separa semnalele pentru condiții diferite (notEmpty/notFull), ceea ce devine foarte util când extindem problema (e.g. buffer de dimensiune mai mare sau mai mulți producători/consumatori). De asemenea, ReentrantLock permite opțiuni precum fairness (lacăte echitabile) și tryLock(), care pot preveni situații de *starvation* sau pot oferi modalități de a evita blocarea completăofficialcto.comofficialcto.com. Pe de altă parte, programatorul trebuie să gestioneze manual deblocarea și să fie atent la utilizarea corectă a finally pentru a nu crea deadlock-uri. În general, pentru majoritatea cazurilor de utilizare obișnuite, mecanismul implicit synchronized este suficient, iar ReentrantLock se folosește când este realmente nevoie de acele caracteristici avansateofficialcto.com.
- Soluția cu BlockingQueue demonstrează puterea abstractizărilor de nivel înalt din Java. Aceasta este cea mai **succintă** și mai puțin predispusă la erori soluție, deoarece delegă sarcina sincronizării unei componente standard din bibliotecă. O coadă blocantă este, în esență, o implementare specializată a pattern-ului producător-consumator, oferind un *buffer thread-safe* și mecanisme de blocare/deblocare interne optimizate. Folosind-o, codul nostru se simplifică considerabil, eliminând complet logicile explicite de așteptare și notificare. Recomandarea generală este ca, acolo unde este posibil, să folosim astfel de unelte de nivel înalt (BlockingQueue, colecții concurente, thread pool-uri cu executori etc.), deoarece ele au fost deja construite și testate pentru a evita erori subtile de sincronizare. Soluția manuală cu monitoare sau lock-uri rămâne însă importantă pentru înțelegerea conceptelor de bază și pentru cazurile când logica de sincronizare nu se potrivește exact unui utilitar existent.

În concluzie, rezolvarea problemei producător-consumator ne-a permis să ilustrăm modul în care diferitele mecanisme de sincronizare din Java pot fi folosite în practică. Am evidențiat importanța utilizării corecte a acestor mecanisme pentru a obține programe concurente corecte și robuste. Experimentele realizate confirmă că toate elementele produse sunt consumate în mod corespunzător, fără acces concurent neprotejat la buffer și fără blocaje indefinite. Prin compararea abordărilor, studenții pot înțelege mai bine trade-off-urile între simplitatea oferită de sincronizarea

implicită față de flexibilitatea lacătelor explicite și comoditatea structurilor de nivel înalt. Pentru majoritatea aplicațiilor, cuvântul cheie synchronized și colecțiile din java.util.concurrent vor fi suficiente și preferate, în timp ce mecanismele ca ReentrantLock devin utile când este nevoie de funcționalități suplimentare (multiple condiții, fairness, tryLock etc.)officialcto.com. Indiferent de metodă, este esențial ca programatorul să respecte principiile de bază ale sincronizării (protectorarea secțiunilor critice, așteptarea în buclă a condițiilor, notificarea corectă a firelor) pentru a evita erori greu de depistat în programele multithread.

Lucrarea de laborator 6.

Sincronizarea firelor de execuție prin clase de sincronizare în problema producător-consumator

Scopul lucrării

Scopul acestei lucrări de laborator este de a familiariza studenții cu programarea concurentă în Java, în special cu **sincronizarea firelor de execuție** folosind mecanisme dedicate. Prin intermediul problemei clasice *Producător-Consumator*, lucrarea își propune să demonstreze atât aspectele teoretice, cât și pe cele practice ale sincronizării. Studenții vor înțelege necesitatea coordonării firelor care accesează resurse comune și vor învăța să utilizeze diferite **clase de sincronizare** oferite de Java (precum synchronized, ReentrantLock, Condition, BlockingQueue) pentru a asigura cooperarea corectă între fire. Lucrarea urmărește dezvoltarea abilităților de a identifica secțiunile critice din cod, de a preveni condițiile de cursă și blocajele, și de a implementa soluții concurrente robuste și eficiente.

Obiective

Prin parcurgerea acestei lucrări de laborator, studenții își vor atinge următoarele obiective:

- **Înțelegerea firelor de execuție (threads)** – ce este un fir de execuție, cum se creează și pornește un thread în Java, și diferența dintre execuție paralelă și concurentă.
- **Recunoașterea necesității sincronizării** – identificarea situațiilor în care accesul concurent la resurse comune poate duce la probleme (condiții de cursă, rezultate incorecte) și motivarea utilizării mecanismelor de excludere mutuală.
- **Studiul problemei Producător-Consumator** – formularea problemei clasice de sincronizare și înțelegerea provocărilor acesteia (coordonarea producerii și consumului de date fără pierderi sau conflicte).
- **Utilizarea monitoarelor Java (synchronized)** – dobândirea abilității de a proteja secțiunile critice cu blocuri sincronizate și de a folosi metodele wait()/notify()/notifyAll() pentru comunicarea între thread-uri.
- **Utilizarea lacătelor explicite (ReentrantLock și Condition)** – familiarizarea cu blocarea și deblocarea manuală a resurselor critice și cu așteptarea condiționată folosind obiecte de tip Condition.
- **Utilizarea cozii blocante (BlockingQueue)** – înțelegerea modului în care colecțiile concurente de nivel înalt pot simplifica sincronizarea producer-consumer prin blocarea automată a firelor atunci când este cazul.

- **Implementare practică multi-threading** – dezvoltarea unei aplicații Java care rezolvă problema producător-consumator folosind mai multe metode de sincronizare și compararea soluțiilor obținute.
- **Analiza și concluzii** – evaluarea beneficiilor și limitărilor diferitelor abordări de sincronizare folosite, precum și asigurarea unor bune practici (de exemplu, evitarea deadlock-ului, folosirea try-finally la deblocare etc.).

Exemplu de realizare:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Random;
import java.util.concurrent.Phaser;
import java.util.concurrent.locks.ReentrantLock;


public class ProdConsPhaser59 {
    static final int CAPACITY = 9;
    static final int TARGET_TOTAL = 59;
    static final int PRODUCERS = 3;
    static final int CONSUMERS = 4;
    // Faza pară = FILL (umplere), faza impară = DRAIN (golire)
    static class Depot {
        final Deque<Integer> buffer = new ArrayDeque<>(CAPACITY);
        final ReentrantLock lock = new ReentrantLock(true);
        final Random rnd = new Random();
        int producedTotal = 0;
        int consumedTotal = 0;
        volatile boolean done = false;

        void printFullIfNeeded() {
            if (buffer.size() == CAPACITY) {
                System.out.printf(">>> Depozitul e plin (%d/%d)%n", buffer.size(), CAPACITY);
            }
        }
        void printEmptyIfNeeded() {
```

```
    if (buffer.isEmpty()) {
        System.out.printf("<<< Depozitul e gol (0/%d)%n", CAPACITY);
    }
}

// întoarce true dacă a produs ceva în această încercare
boolean tryProduce(String name) {
    lock.lock();
    try {
        if (producedTotal >= TARGET_TOTAL) return false; // am terminat de produs
        if (buffer.size() == CAPACITY)   return false;   // plin -> așteptăm faza DRAIN
        int value = 31 + rnd.nextInt(59); // [31..89], doar ca să fie lizibil
        buffer.addLast(value);
        producedTotal++;
        System.out.printf("%s a produs %d | stoc=%d/%d | totalProd=%d%n",
                name, value, buffer.size(), CAPACITY, producedTotal);
        if (buffer.size() == CAPACITY) printFullIfNeeded();
        return true;
    } finally {
        lock.unlock();
    }
}
// întoarce true dacă a consumat ceva în această încercare
boolean tryConsume(String name) {
    lock.lock();
    try {
        if (buffer.isEmpty()) return false;
        int value = buffer.removeFirst();
        consumedTotal++;
        System.out.printf("%s a consumat %d | stoc=%d/%d | totalCons=%d%n",
                name, value, buffer.size(), CAPACITY, consumedTotal);
```

```
            if (buffer.isEmpty()) printEmptyIfNeeded();
            if (consumedTotal >= TARGET_TOTAL && buffer.isEmpty()) {
                done = true; // am atins ținta și am golit depozitul
            }
            return true;
        } finally {
            lock.unlock();
        }
    }
}
static class Producer extends Thread {
    private final Depot depot;
    private final Phaser phaser;
    Producer(Depot depot, Phaser phaser, String name) {
        super(name);
        this.depot = depot;
        this.phaser = phaser;
    }
    @Override public void run() {
        try {
            while (!depot.done) {
                int phase = phaser.getPhase();
                if (phase % 2 == 0) { // FILL
                    // Producem până când depozitul e plin sau am atins ținta
                    boolean madeSomething = true;
                    while (!depot.done) {
                        madeSomething = depot.tryProduce(getName());
                        if (!madeSomething) break; // plin sau produs tot
                    }
                    // semnalăm bariera: am terminat contribuția pe această fază
                    phaser.arriveAndAwaitAdvance();
                } else {
```

```
                // Fază DRAIN -> producătorii stau
                phaser.arriveAndAwaitAdvance();
            }
          }
        } catch (Exception ignored) {
        } finally {
          // la final, deregistare grațioasă
          try { phaser.arriveAndDeregister(); } catch (IllegalStateException ignored) {}
        }
    }
}

static class Consumer extends Thread {
    private final Depot depot;
    private final Phaser phaser;
    Consumer(Depot depot, Phaser phaser, String name) {
        super(name);
        this.depot = depot;
        this.phaser = phaser;
    }
    @Override public void run() {
        try {
            while (!depot.done) {
                int phase = phaser.getPhase();
                if (phase % 2 == 1) { // DRAIN
                    // Consumăm până când depozitul e gol (sau am finalizat ținta)
                    boolean tookSomething = true;
                    while (!depot.done) {
                        tookSomething = depot.tryConsume(getName());
                        if (!tookSomething) break; // gol -> trecem la faza următoare
                    }
                    phaser.arriveAndAwaitAdvance();
```

```
                } else {
                    // Fază FILL -> consumatorii stau
                    phaser.arriveAndAwaitAdvance();
                }
            }
        } catch (Exception ignored) {
        } finally {
            try { phaser.arriveAndDeregister(); } catch (IllegalStateException ignored) {}
        }
    }
}
public static void main(String[] args) throws InterruptedException {
    Depot depot = new Depot();
    // Phaser cu 3 producători + 4 consumatori (=7 "parties")
    Phaser phaser = new Phaser(PRODUCERS + CONSUMERS);
    Thread[] producers = new Thread[PRODUCERS];
    Thread[] consumers = new Thread[CONSUMERS];
    for (int i = 0; i < PRODUCERS; i++) {
        producers[i] = new Producer(depot, phaser, "Producator-" + (i + 1));
    }
    for (int i = 0; i < CONSUMERS; i++) {
        consumers[i] = new Consumer(depot, phaser, "Consumator-" + (i + 1));
    }
    // La start: faza 0 (FILL). Consumatorii vor aștepta, producătorii umplu.
    for (Thread t : producers) t.start();
    for (Thread t : consumers) t.start();
    // Așteptăm finalizarea logică
    for (Thread t : producers) t.join();
    for (Thread t : consumers) t.join();
        System.out.println("=== Gata: au fost produse și consumate " + TARGET_TOTAL + "
obiecte în cicluri FILL→DRAIN. ===");
}}
```

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". Dimensiunea depozitului este D. Producătorii comlectează depozitul cu D obiecte. Consumatorii nu pot consuma, până când depozitul nu va fi plin. După care consumatorii consumă. Producărorii nu pot produce, până când depozitul nu va fi gol. Toate operaţiile se efectuează până când nu vor fi produse și consumate Z obiecte.

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

Sarcina de realizat în grup. Un membru a grupului realizează sarcina și sincronizarea producătorului. Al doilea membru a grupului realizează sarcina și sincronizarea consumatorului.

Sarcina poate fi realizată prin thread-uri clasice sau prin pool-uri de thread-uri. Sincronizarea de realizat numai prin clase se sincronizare din Java.

Tabelul 1 Variantele pentru realizarea sarcinii

Nr.	X	Y	Z	D	Tip Obiecte
1	2	3	40	8	Numere pare
2	3	4	45	5	Numere impare
3	4	3	50	10	Vocale
4	3	2	66	11	Consoane
5	2	5	60	12	Numere pare
6	2	4	42	7	Numere impare
7	3	3	42	6	Vocale
8	4	2	45	5	Consoane
9	5	2	44	4	Numere pare
10	3	3	48	8	Numere impare
F - fiecare producător produce câte 1 obiect 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 interfeţei 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.

8. Întrebări de verificare:

Pentru a verifica înțelegerea aspectelor teoretice și practice acoperite în această lucrare de laborator, răspundeți la următoarele întrebări:

1. Explicați conceptul de *secțiune critică*. De ce este necesară sincronizarea atunci când mai multe thread-uri accesează aceeași resursă?
2. Comparați pe scurt mecanismele synchronized și ReentrantLock. Menționați două avantaje pe care ReentrantLock le are față de synchronized și o responsabilitate suplimentară pe care o impune programatorului.
3. Ce este un obiect Condition și cum se obține unul? Cum diferă utilizarea await()/signal() față de wait()/notify()?
4. În exemplul problemei producător-consumator, de ce am folosit două obiecte Condition separate (notEmpty și notFull)? Ce s-ar întâmpla dacă am folosi un singur obiect Condition pentru ambele tipuri de așteptare?
5. Ce este o coadă blocantă (*blocking queue*) și cum facilitează ea comunicarea între producători și consumatori? Explicați în propriile cuvinte cum funcționează metodele put() și take() din BlockingQueue.
6. Dați exemplu de cel puțin două implementări concrete ale interfetei BlockingQueue în Java și specificați diferențe între ele (de exemplu, legat de capacitate sau tipul de ordine).
7. În codul soluției (cu ReentrantLock), ce s-ar putea întâmpla dacă am uita să apelăm lock.unlock() în blocul finally al metodei get()? Dar dacă am apela notFull.signal() înainte de a elibera lacătul?
8. Explicați pe scurt cum ați extinde aplicația realizată pentru cazul în care producătorul și consumatorul ar trebui să funcționeze la nesfârșit (nu doar 10 elemente). Cum ați semnala terminarea firelor în absența unui număr finit de elemente? (Sugestie: *poison pill* – un element special de terminare în coadă)

Bibliografie

1. **FCIM, UTM Chișinău** – *Problema clasică producător-consumator – Sincronizarea firelor de execuție*, lucrare de laborator, Facultatea Calculatoare, Informatică și Microelectronicăelse.fcim.utm.mdelse.fcim.utm.md (limba română).
2. **Cosmina Ivan** – *Tehnologii și Aplicații în Calculul Paralel și Distribuit*, Editura U.T. Pres, Cluj-Napoca, 2013biblioteca.utcluj.robiblioteca.utcluj.ro (limba română).
3. **Krzysztof Dymek** – *Java: продвинутая конкурентность* (traducere în limba rusă), Habr, 2022habr.com (limba rusă).
4. **Official CTO Blogs** – *Explicit Locks in Java – ReentrantLock*, articol onlineofficialcto.comofficialcto.com (limba engleză).
5. **Baeldung** – *Guide to java.util.concurrent.BlockingQueue*, articol onlinebaeldung.com (limba engleză).