

# Tema TPM

Rosca Alexandru-David, Octavian Regatun

Noiembrie 2024

## 1 Exercițiul 1

Avem secvența de execuție de mai jos.

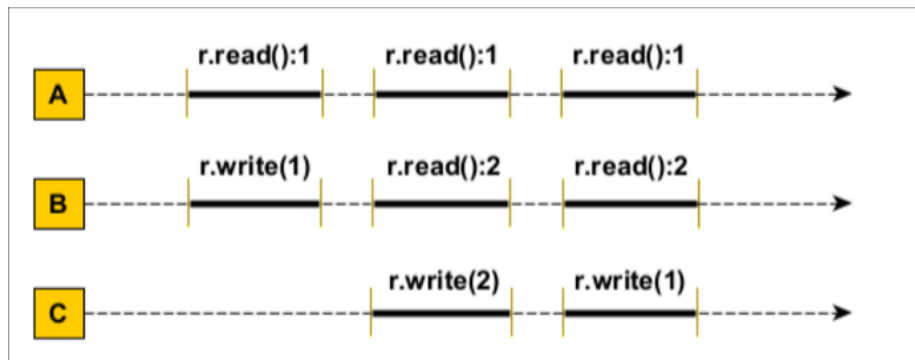


Figure 1: Secvența de Execuție

### 1.1 Liniarizabilitatea

Este aceasta liniarizabila?

**Da, secvența de execuție este liniarizabila** deoarece există o ordine secvențială în care toate operațiile asupra variabilei `r` par să se fi desfășurat instantaneu, fiecare la un moment specific din această ordine, iar valorile obținute la citiri reflectă scrierile anterioare corespunzătoare la care ne așteptăm.

Acest concept se poate observa grafic în exemplul de mai jos care ilustrează o secvență liniarizabilă.

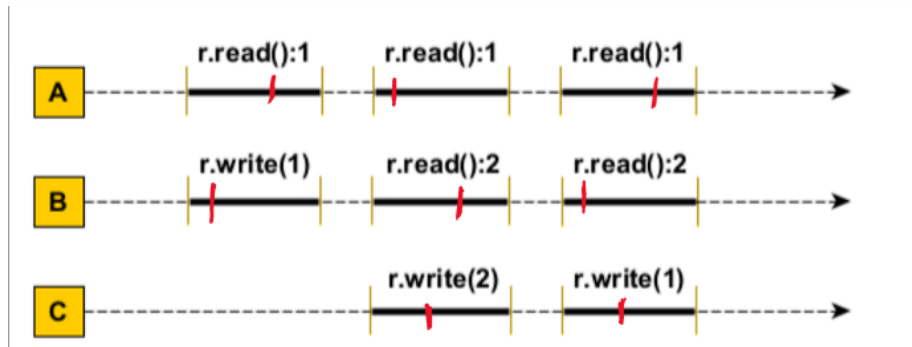


Figure 2: Exemplu Liniarizabilitate

Vom explica acum de ce figura de mai sus demonstreaza faptul ca secventa de executie este liniarizabila, aratand istoria secventei de executie si împartind fiecare sectiune transversal si analizând fiecare caz pentru a determina valorile de intrare, intermediare si de iesire posibile.

### 1.1.1 Sectiunea 1

În prima sectiune, observam urmatoarele comportamente:

- Se intra cu valoarea  $r = 0$  în sectiune, deci ca Thread-ul A sa citeasca 1, cineva trebuie sa scrie aceasta valoare în  $r$ , lucru pe care poate sa-l faca Thread-ul B, lucru marcat mai jos ca atare.
- Indiferent de momentul în care are loc scrierea, variabila  $r$  va avea valoarea 1 la finalul acestei sectiuni

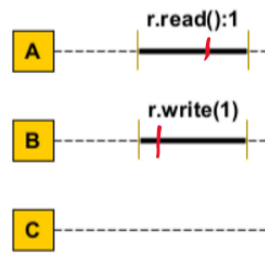


Figure 3: Sectiunea 1

### 1.1.2 Sectiunea 2

În sectiunea a doua, analizam urmatoarele comportamente:

- Se intra cu valoarea  $r = 1$  în sectiune, deci Thread-ul A poate sa citeasca 1 direct, totusi Thread-ul B are nevoie sa citeasca 2 deci cineva trebuie sa scrie aceasta valoare în  $r$ , lucru pe care poate sa-l faca Thread-ul C.
- Indiferent de momentul în care are loc scrierea din partea Thread-ului C, variabila  $r$  va avea valoarea 2 la finalul acestei sectiuni

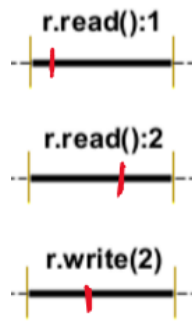


Figure 4: Sectiunea 2

### 1.1.3 Sectiunea 3

În sectiunea a treia, comportamentele observate sunt urmatoarele:

- Se intra cu valoarea  $r = 2$  în sectiune, deci Thread-ul B poate sa citeasca 2 direct de la inceput, totusi Thread-ul A are nevoie sa citeasca 1 deci cineva trebuie sa scrie aceasta valoare în  $r$ , lucru pe care poate sa-l faca Thread-ul C.
- Indiferent de momentul în care are loc scrierea din partea Thread-ului C, variabila  $r$  va avea valoarea 1 la finalul acestei sectiuni

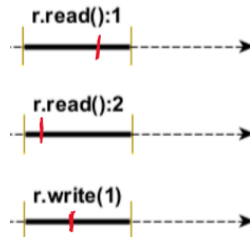


Figure 5: Sectiunea 3

#### 1.1.4 Istoria Secvenței de Executie

Pentru a demonstra ca aceasta secvență este liniarizabila, putem ordona operațiile astfel încât acestea să reflecte o ordine secvențială realistă a lor:

- B scrie valoarea 1 — Secțiunea 1 — r.write(1)
- A citește valoarea 1 — Secțiunea 1 — r.read(1)
- A citește valoarea 1 — Secțiunea 2 — r.read(1)
- C scrie valoarea 2 — Secțiunea 2 — r.write(2)
- B citește valoarea 2 — Secțiunea 2 — r.read(2)
- B citește valoarea 2 — Secțiunea 3 — r.read(2)
- C scrie valoarea 1 — Secțiunea 3 — r.write(1)
- A citește valoarea 1 — Secțiunea 3 — r.read(1)

## 1.2 Consistența Secvențială

**Este secvența consistentă secvențial?**

**Da, secvența de execuție este consistentă secvențial** deoarece liniarizabilitatea este o formă mai strictă a consistenței secvențiale.

Prin definiție, dacă o secvență este liniarizabilă, aceasta respectă ordinea secvențială și, în plus, asociază fiecărei operații un moment "instantaneu", ceea ce reprezintă o constrângere în plus.

Încât secvența este liniarizabilă ea verifică pe lângă toate cerințele de consistență secvențială și altele în plus.

Consistența secvențială permite o ordine secvențială fără a necesita instantaneitate de care vorbeam anterior, în timp ce liniarizabilitatea impune atât respectarea ordinii cât și un punct specific în timp în care fiecare operație pare să se execute.

## 2 Exercițiul 2

### 2.1 Clean-Up Matches Acquisition

Apelul de `unlock()` trebuie executat doar atunci când `lock()`-ul corespunzător lui a fost cerut și obținerea acestuia s-a efectuat. În varianta în care `lock`-ul este obținut în interiorul `try`-ului, `unlock()` se execută chiar și când obținerea lacatului nu s-a realizat.

### 2.2 Clean Code

Obținerea lacatului înainte de blocul `try` indică clar că blocarea are loc înainte de intrarea în secțiunea de cod protejată. Astfel, codul se păstrează curat, are un flow natural, și nu prezintă posibile erori de logică pentru cineva care interacționează cu acesta.

### 2.3 Certitudinea Obținerii / Eliberării Lacatului

#### 2.3.1 Cazul I

În prima versiune, `lock()` este apelat înainte de blocul `try`, lucru ce asigură faptul că atunci când intrăm în interiorul secțiunii lacatul a fost deja obținut. Astfel, finaly va debloca întotdeauna `lock`-ul, indiferent dacă execuția codului a fost făcută cu succes sau o excepție a fost aruncată în blocul de cod delimitat de `try`.

#### 2.3.2 Cazul II

În a doua versiune, dacă o excepție apare în interiorul secțiunii delimitate de `try` înainte de obținerea lacatului acesta nu va fi obținut însă instrucțiunea `finally` va încerca să apeleze `unlock` pe acesta fapt ce va conduce la o posibilă eroare depinzând de limbajul de programare folosit dar și de implementarea lacatului respectiv.

#### 2.3.3 Exemplificare Lacat din Java

Când încerci să dai `unlock` la un lacat în Java care nu a fost blocat anterior, o excepție de tipul `IllegalMonitorStateException` va fi aruncată. Această excepție este definită exact pentru a semnala faptul că metoda de `unlock` a fost apelată fără să existe anterior un apel al funcției `lock()` pe acel lacat.

`ReentrantLock` de exemplu folosește un contor intern pentru a verifica dacă `lock()` a fost apelat și cine deține acel lacat. Dacă se încearcă deblocarea sau fără ca el să fi fost blocat mai întâi sau dacă un alt fir de execuție încearcă să-l deblocheze, excepția va fi aruncată.

Alte tipuri de excepții care pot apărea sunt `IllegalArgumentException` sau `IllegalStateException` acestea apar în cazuri specifice de lacate personalizate dar cum noua nu ni s-a prezentat tipul de lacat folosit trebuie să luăm în calcul și această opțiune.

## 2.4 Probleme de Sincronizare / Deadlocks

### 2.4.1 General

În prima versiune, logica este bine definită, blocarea începe înainte de secțiunea încadrată în `try` și este eliberată în `finally`. În a doua versiune este posibil să apară deadlock-uri sau erori de sincronizare din cauza unui flux de control complicat

### 2.4.2 Despre Sincronizare

Dacă `lock()` este în interiorul blocului `try`, există o mică posibilitate ca o excepție să apară imediat după începerea rularii `try`-ului și în același timp înainte de `lock()` astfel încât intrăm în secțiunea protejată fără ca `lock`-ul să fie obținut. Acest lucru ar duce la o eroare de sincronizare, alte thread-uri pot accesa simultan secțiunea critică, încălcând proprietățile de consistență și sincronizare.

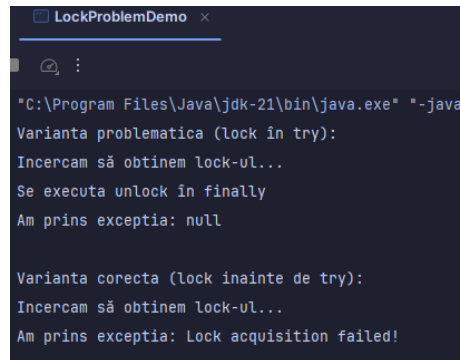
### 2.4.3 Despre Deadlock uri

Dacă obținerea lacatului este în interiorul blocului `try`, în special într-un cod mai complicat, există riscul să apară deadlock-uri dacă ordinea apelurilor de `lock` nu este consistentă lucru pe care putem să-l exemplificăm succint astfel. De exemplu, dacă există alte blocuri `try` și `lock`-uri înainte de `someLock.lock()`, o excepție aruncată acolo ar putea lăsa codul într-o stare inconsistentă sau cu mai multe `lock`-uri blocate.

Dacă folosim prima versiune a implementării, ne asigurăm că lacatul este obținut atunci când dormim acest lucru înainte de oricare thread, și nu este influențat de eventualele excepții din blocul `try`, minimizând astfel riscul deadlock-ului.

## 2.5 Rezultatul executiei codulu

Implementarea codului se poate regasi in repository-ul de GitHub al temei in folder-ul asociat Exerciitiului 2



```
LockProblemDemo x
:
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaa
Varianta problematica (lock în try):
Incercam să obținem lock-ul...
Se executa unlock în finally
Am prins exceptia: null

Varianta corecta (lock înainte de try):
Incercam să obținem lock-ul...
Am prins exceptia: Lock acquisition failed!
```

Figure 6: Demonstratie rulare defectuasa

## 2.6 Concluzii

- Asigura ca obtinerea lacatului este obtinuta corect si complet înainte de sectiunea critica.
- Evita exceptii neasteptate în blocul finally care distrug flow-ul aplicatiei.
- Reduce riscul de race conditions si deadlock.
- Codul mai clar d.p.d.v al logicii si mai usor de intretinut si refactorizat.
- Codul din versiunea a 2 a se poate schimba slightly pentru a fi mult mai sigur prin adaugarea unui flag care ne spune daca obtinerea lacatului a fost facuta cu succes

```
boolean locked = false;
try {
    someLock.lock();
    locked = true;
    ...
}
finally {
    if (locked) {
        someLock.unlock();
    }
}
```

## 3 Exercitiul 3

### 3.1 Introducere

În aceasta analiza, vrem sa vedem de ce în algoritmul Bakery e important sa comparăm perechile de tipul  $(label[i], i)$  și  $(label[k], k)$ , și nu doar etichetele sau doar indexurile.

### 3.2 Contextul Problemei

Algoritmul Bakery asigura exclusivitate folosind o structura de tipul urmator:

```
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;

    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false;
            label[i] = 0;
        }
    }

    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (exists k!=i with flag[k]==true && (label[i],i) > (label[k],k)) {};
    }

    public void unlock() {
        flag[i] = false;
    }
}
```

### 3.3 Analiza Cazurilor Problematic

#### 3.3.1 Cazul 1: Compararea Doar a Etichetelor (label)

Daca am compara doar etichetele  $label[i] > label[k]$ , s-ar putea crea o situatie de deadlock. Sa luam ca exemplu doua thread-uri:



Timp	Thread 1	Thread 2	Observatii
$t_1$	$flag[1] = \text{true}$	–	Thread 1 începe
$t_2$	$label[1] = 5$	–	Thread 1 ia numarul 5
$t_3$	–	$flag[2] = \text{true}$	Thread 2 începe
$t_4$	–	$label[2] = 5$	Thread 2 ia acelasi numar
$t_5$	blocat	blocat	DEADLOCK

Table 1: Exemplu de deadlock folosind doar etichete

În acest caz:

- Ambele thread-uri obtin aceeași eticheta ( $label = 5$ ).
- Fiecare observa ca celalalt are  $flag = \text{true}$ .
- Compararea doar a etichetelor  $label[i] > label[k]$  nu le poate departaja.
- Rezulta deadlock, deoarece ambele thread-uri rămân blocate așteptând unul după altul.

### 3.3.2 Cazul 2: Compararea Doar a Indexurilor

Dacă am compara doar indexurile  $i > k$ , am putea ajunge la probleme de tipul starvation:

Timp	Thread 1	Thread 2	Observatii
$t_1$	$flag[1] = \text{true}$	–	Thread 1 începe
$t_2$	$label[1] = 5$	–	Thread 1 ia numarul 5
$t_3$	–	$flag[2] = \text{true}$	Thread 2 începe
$t_4$	–	$label[2] = 3$	Thread 2 ia numarul 3
$t_5$	intra în SC	blocat	STARVATION

Table 2: Exemplu de starvation folosind doar indexuri

În acest caz:

- Thread-urile cu index mai mic ar avea mereu prioritate.
- Thread-urile cu index mai mare ar putea fi blocate la nesfârșit.
- Algoritmul nu mai asigură fairness, adică șanse egale pentru toți.

## 3.4 Soluția: Compararea Tuplelor

Comparatia perechilor  $(label[i], i) > (label[k], k)$  rezolvă ambele probleme, astfel:

1. **Ordonare Primara:** Compararea pe baza etichetelor ( $label$ ) asigură regula FIFO (First-In-First-Out), deci thread-urile sunt deservite în ordinea în care solicită accesul.

2. **Ordonare Secundara:** Compararea pe baza indexurilor ( $i$ ) ne scapa de situatiile în care etichetele sunt egale, prevenind astfel deadlock-ul si asigurând o ordine clara.

## 4 Exercițiul 4

### 4.1 Subiectul a

#### 4.1.1 Directia Demonstratiei

Vom demonstra ca algoritmul de lock prezentat nu este starvation-free intr-un sistem cu  $n > 1$  thread-uri.

#### 4.1.2 Cod Analizat

```
1 class ShadyLock {
2     private volatile int turn;
3     private volatile boolean used = false;
4
5     public void lock() {
6         int me = ThreadId.get();
7         do {
8             do {
9                 turn = me;
10            } while (used);
11            used = true;
12        } while (turn != me);
13    }
14
15    public void unlock () {
16        used = false;
17    }
18 }
```

#### 4.1.3 Demonstratie prin Trace

Vom prezenta un scenariu concret cu doua thread-uri care demonstreaza posibilitatea de starvation.

#### 4.1.4 Starea Initiala

- `used = false`
- `turn = 0`

#### 4.1.5 Secventa de Executie

1. Thread-ul A ( $ID = 1$ ) apeleaza `lock()`:

- `me = 1`
- `turn = 1`
- `used = true`

- Thread-ul A intra in sectiunea critica
2. Thread-ul B (ID = 2) apeleaza `lock()`:
    - `me = 2`
    - `turn = 2`
    - B asteapta in bucla `while(used)` deoarece `used = true`
  3. Thread-ul A termina executia sectiunii critice si apeleaza `unlock()`:
    - `used = false`
  4. Thread-ul A apeleaza imediat `lock()` din nou:
    - `turn = 1`
    - `used = true`
    - A reintra in sectiunea critica inainte ca B sa poata progresa

#### 4.1.6 Analiza

Acest trace demonstreaza doua probleme fundamentale care fac ca algoritmul sa nu fie starvation-free:

1. Lipsa Mecanismului de ordonare: Nu exista niciun mecanism care sa asigure ordinea in care thread-urile acceseaza sectiunea critica de exemplu un queue, spre deosebire de algoritmi precum cel al lui Peterson.
2. "Reobtinerea Imediata a lacatului" se intampla deoarece un thread poate sa reachizitioneze imediat lacatul dupa ce il elibereaza, impiedicand alte thread-uri sa progreseze. Acest lucru se intampla pentru ca:
  - Dupa `unlock()`, variabila `used` devine `false`
  - Thread-ul care tocmai a eliberat lock-ul poate executa `lock()` inainte ca alte thread-uri sa aiba sansa sa progreseze
  - Nu exista nicio restrictie care sa impiedice acelasi thread sa obtina lock-ul de mai multe ori consecutiv
  - In acelasi timp exista optimizari pe CPU care fac ca un thread care asteapta mult intr-o bucla while sa faca check uri mai rar cu cat nu se observa nicio schimbare in acel fir de executie deci asta permite sanse mai mari ca thread-ul A sa reintre in SC

#### 4.1.7 Concluzie

Algoritmul nu este starvation-free deoarece:

1. Un thread poate monopoliza accesul la SC prin reachizitionarea continua a lacatului

2. Nu exista niciun mecanism care sa garanteze ca thread-urile care asteapta vor primi eventual acces
3. Modificarea conflictuala a variabilelor **turn** si **used** poate duce la situatii in care unele thread-uri nu progreseaza niciodata

## 4.2 Subpunctul b

### Descrierea algoritmului

Mai întâi, sa înțelegem ce face algoritmul:

- Foloseste doua variabile volatile  $x$  si  $y$  initializate cu 0.
- Fiecare thread are un ID unic pozitiv.
- Se încearca o forma de excludere mutuala în doua faze, folosind aceste variabile.
- Exista si un lock încapsulat care e folosit în anumite conditii.

### Scenariu problematic cu doua thread-uri ( $T_1$ si $T_2$ )

Vom analiza un scenariu în care doua thread-uri,  $T_1$  si  $T_2$ , interactioneaza cu variabilele  $x$  si  $y$  dupa cum urmeaza:

Timp	Thread 1 (ID = 1)	Thread 2 (ID = 2)
$t_1$	$x = 1$	
$t_2$		$x = 2$
$t_3$	<b>while</b> ( $y \neq 0$ ) {} ambele trec pentru ca $y = 0$	<b>while</b> ( $y \neq 0$ ) {} ambele trec pentru ca $y = 0$
$t_4$	$y = 1$ suprascriu succesiv $y$	$y = 2$ suprascriu succesiv $y$
$t_5$	<b>if</b> ( $x \neq 1$ ) $x = 2$ , deci $T_1$ va lua lock	<b>if</b> ( $x \neq 2$ ) $x = 2$ , deci $T_2$ nu va lua lock

### Problema critica

În acest scenariu,  $T_2$  a reusit sa intre în sectiunea critica fara sa astepte dupa lock-ul încapsulat:

- $T_1$  a luat lock-ul încapsulat, dar  $T_2$  a trecut oricum.
- Astfel, ambele thread-uri pot fi în sectiunea critica simultan.

## De ce apare problema

Problema apare din urmatoarele motive:

- Variabila  $x$  poate fi suprascrisa înainte ca un thread sa verifice valoarea ei.
- Conditia `while(y != 0)` nu ofera garantii suficiente când mai multe thread-uri o verifica simultan.
- Nu exista sincronizare între scrierea în  $x$  si verificarea valorii lui  $x$ .

## Concluzie

Nu, acest algoritm nu asigura excluderea mutuala. Am demonstrat un scenariu în care doua thread-uri pot intra simultan în sectiunea critica, încalcând princip-iul excluderii mutuale. Problema fundamentala este ca modificarile variabilelor  $x$  si  $y$  nu sunt atomic sincronizate între ele, permitând conditii de cursa (*race conditions*) care duc la esecul protocolului de excludere mutuala.

## 4.3 Subpunctul c

### 4.3.1 Valori posibile pentru fiecare thread

- "white" - daca gaseste `getWhite = true`
- "red" - daca seteaza `getWhite = true` si ( $last = me$ )
- "black" - daca seteaza `getWhite = true` si ( $last \neq me$ )

### 4.3.2 Maxim 1 valoare rosie?

Exista 2 cazuri posibile care se pot intampla in codul dat, unul dintre cazuri este atunci cand toate thread-urile trec de *if(getWhite)* iar al doilea caz este atunci cand doar unul reuseste sa treaca de acest if inaintea celorlalte, vom discuta si despre un caz putin mai special la finalul primului caz care este o combinatie dintre celelalte 2, acum vom exemplifica si discuta mai pe larg cele 2 cazuri in cele ce urmeaza

- Cazul I (toate thread-urile trec de prima instructiune if):  
Daca toate thread-urile apeleaza `choose()` in acelasi timp, chiar daca fiecare linie poate fi citita in acelasi timp de toate thread-urile si variabila `last` va fi modificata secvential de fiecare in parte doar ultimul thread va schimba cu adevarat valoarea variabilei `last` pe termen lung (adica `last` va avea id-ul ultimului thread), ulterior toate pot sa citeasca instructiunea *if(getWhite)* in perioade asemanatoare (inainte ca `getWhite` sa fie modificat) si sa evalueze expresia ca fiind falsa (*getWhite == false*), apoi toate thread-urile trec de if si prima dintre ele modifica valoarea lui `getWhite` in true. Acum toate thread-urile mai putin ultimul vor citi instructiunea

$if(last == me)$  si o vor evalua ca fiind falsa si vor returna "black" iar ultimul va evalua expresia ca fiind adevarata si va returna "red". Acest caz poate fi completat cu cateva cuvinte sa vorbeasca si despre cazul in care doar unele thread-uri trec in acelasi timp de primul if caz in care din nou doar ultimul thread poate returna "red" sau daca intre timp last este modificat niciun thread nu mai poate returna "red" deoarece  $last \neq me$  pentru toate thread-urile din a 2-a parte de cod  
 $\implies 1/0$  valoare(i) "red"

- Cazul II (unul dintre thread-uri trece de if inaintea celorlalte)  
 In acest caz unul dintre thread-uri a fost mai rapid decat celelalte si a trecut de  $if(GetWhite)$  si a setat GetWhite ca fiind True, acum exista inca 2 mini-cazuri
- Cazul II.1 (un alt thread in acelasi timp a modificat last cu id-ul lui)  
 In acest caz primul thread care a trecut de if va returna "black" si restul thread-urilor vor returna "white"  
 $\implies 0$  valori "red"
- Cazul II.2 (niciun alt thread nu a modificat inca last cu id-ul lui)  
 In acest caz primul thread care a trecut de if va returna "red" deoarece el fusese ultimul care trecuse pe linia  $last = me$  si restul thread-urilor ca si in celalalt caz vor returna "white"  
 $\implies 1$  valoare "red"

Astfel am demonstrat ca orice s-ar intampla nu putem returna mai mult de 1 valoarea de "red". Lucru ce are foarte mult sens daca luam problema logic, odata ce un thread modifica GetWhite in true e ca si cum a inchis posibilitatea de a putea returna red deoarece toate nodurile de dupa vor returna white, astfel doar cei care au trecut de if inainte ca aceasta schimbare sa aiba loc pot sa returneze red insa doar unul dintre ei poate face asta deoarece  $last == me$  poate avea loc doar pentru un singur thread care a trecut deja de primul if deoarece asta ar insemna sa se intoarca inapoi sa schimbe valoarea lui last si acest lucru nu este posibil, de asemenea altele nu au cum sa treaca de if deoarece vor returna "white" intre timp

#### 4.3.3 Maxim n-1 valori negre?

Putem folosi un rationament similar pentru acest subpunct ca la cel anterior in care avem 2 cazuri, de asemenea nu mai trebuie sa tratam cele 2 cazuri mini si vom mentiona de ce in urmatoarele momente

- Cazul I (toate thread-urile trec de prima instructiune if):  
 Daca toate thread-urile trec de primul if ultimul thread care a modificat  $last = me$  va returna "red" pe cand restul de n-1 thread-uri vor returna "black", nu ne mai intereseaza cazul in care unele thread-uri trec de if inainte ca GetWhite sa fie modificat deoarece am demonstrat ca daca toate trec maximul de valori black returnat este n-1, dar daca acest lucru

s-ar intampla (doar unele thread-uri apuca sa treaca de primul if dar altele raman inainte) numarul de thread-uri care au trecut (notat cu  $m$ ) va fi  $m \leq n - 1$ , deci pot returna maxim  $m$  noduri "black" daca last este modificat intre timp deci cum  $m \leq n - 1$   
 $\implies$   $n-1$  valori "black"

- Cazul II (unul dintre thread-uri trece de if inaintea celorlalte)  
 Acest caz este usor de demonstrat deoarece daca unul dintre thread-uri trece de primul if acesta va modifica GetWhite astfel incat sa aiba valoarea True si in mod cert restul nu vor returna "white", acum nu ne pasa daca acest thread care a trecut de if returneaza "red" sau "black" deoarece  $1 \leq n - 1$  pentru cazuri de concurenta asa cum este prezentata problema.  
 $\implies$   $1/0$  valori "black"

Astfel pot exista maxim  $n-1$  valori de "black" returnate de  $n$  thread-uri indiferent de momentul apelarii functiei choose()

#### 4.3.4 Concluzii

Maxim un thread poate obtine "red" (prin analiza constructiva) Maxim  $n-1$  thread-uri pot obtine "black" (prin analiza constructiva)



## 5 Exercițiul 5

### 5.1 0-Bounded Waiting

Bounded waiting (sau fairness) este un termen care descrie de câte ori un proces este ocolit de un alt proces care intra in locul sau in sectiunea protejata dupa ce acesta a indicat deja inainte intentia de a intra în sectiunea critica. Merita mentionat faptul ca acesta definitie este una generala si in demonstratiile, exemplificarile si codul prezentat mai jos se va pune accent asa cum ni s-a comunicat de catre profesorul de curs **Emanuel Onica** pe partea de **fairness** pe care trebuie sa o obtinem

In contextul exercitiului se va evidentia asigurarea fairness-ului care se va putea observa mai jos printr-o uniformitate in numarul de accese ale thread-urilor la SC ( +1 acces la toate mai putin 1 thread care vine din faptul ca se verifica ca suma de 300k a fost atinsa, pe cand un singur thread este cel care chiar atinge acesta suma in mod incipient dar aceste accesari nu se iau in calcul pentru exercitiu)

### 5.2 Lock-ul Peterson

În varianta clasica (non-volatila in cazul nostru) a algoritmului Peterson pentru n thread-uri, nu exista siguranta ca toate firele vor avea acces uniform la SC. Thread-urile pot ramâne blocate deoarece altele trec peste acesta in timp ce el asteapta sa intre in SC, aceste lucru se intampla deoarece algoritmul fara optimizarile pe care le vom mentiona in urmatoarele randuri nu tine cont de frecventa intrarilor in SC de catre fiecare thread in parte.

### 5.3 Optimizarea Gasita

Algoritmu optimizat foloseste doua elemente suplimentare fata de cel clasic pentru a asigura proprietatea de 0-bounded waiting.

- `accessCount`: Retine numarul de accesari la SC al fiecarui thread.
- `shouldYield()`: Aceasta verifica, pentru fiecare thread in parte daca acesta ar trebui sa cedeze ("yield") accesul la SC altor thread-uri care au avut mai putine accesari in zona protejata. Daca un thread are un numar de accesari mai mare decat oricare altul, aceasta valoare depasind un prag setat prin variabila `THRESHOLD`, thread-ul curent va permite acestora sa progreseze.

## 5.4 Modificari Lock()/Unlock()

### 5.4.1 Lock()

În while-ul din functia lock(), fiecare thread verifica de fiecare data daca trebuie sa faca yield si sa permita alota thread-uri sa intre in SC prin apelul functiei shouldYield. Daca apelul functiei este unul adevarat thread-ul elibereaza pe moment nivelul in care se afla si da apeleaza yield(), permitând astfel altor candidati sa progreseze. Dupa ce acestia termina executia SC, thread-ul revine la nivelul unde era inainte si încearca din nou sa treaca mai departe catre urmatorul strat.

### 5.4.2 Unlock()

Functia unlock() nu se schimba extrem de mult, insa fiecare thread anunta intrarea in SC prin modificarea conter-ului de accesari asociat acestuia, lucru care mentine un istoric al accesarilor la SC, permitând astfel uniformitatea accesarii prin verificarea diferentei in functia shouldYeld()

## 5.5 Cum suntem siguri ca 0-Bounded Waiting are loc?

- Dominanta accesului la SC: Functia pe care am adus-o de uniformizare a numarului de accesari la SC aka shouldYield actioneaza ca un mecanism de echilibrare al accesarilor si mentinând o diferenta prestabilita intre numarul de accesari, astfel niciun thread nu poate prelua controlul asupra SC pentru o perioada indelungata lucru care nu se poate spune si despre varianta de baza a lock-ului Peterson
- Evitarea starvation-ului: Un thread care constata ca a accesat SC mult mai mult decât altele va ceda accesul sau in favoarea altor thread-uri care doresc intrarea in sectiunea protejata, lucru care conduce la o uniformitate al numarul de accesari.

## 5.6 Implementare

Implementarile se pot gasi de asemenea in folder-ul asociat exercitiului 5 din repository-ul de GitHub. Rularea codului se va putea gasi in sesiunea aferenta 3.7

### 5.6.1 Peterson Clasic

```
1 public class PetersonLock {
2     private final int n;
3     private final AtomicInteger[] level;
4     private final AtomicInteger[] victim;
5     private int sharedCounter;
6     private final int LIMIT = 300000;
7
8     public PetersonLockNonVolatile(int n) {
9         this.n = n;
10        level = new AtomicInteger[n];
11        victim = new AtomicInteger[n];
12        sharedCounter = 0;
13        for (int i = 0; i < n; i++) {
14            level[i] = new AtomicInteger(0);
15            victim[i] = new AtomicInteger(0);
16        }
17    }
18
19    public void lock(int i) {
20        for (int L = 1; L < n; L++) {
21            level[i].set(L);
22            victim[L].set(i);
23            boolean otherThr;
24            do {
25                otherThr = false;
26                for (int k = 0; k < n; k++) {
27                    if (k != i && level[k].get() >= L) {
28                        otherThr = true;
29                        break;
30                    }
31                }
32            } while (otherThr && victim[L].get() == i);
33        }
34    }
35
36    public void unlock(int i) {
37        level[i].set(0);
38    }
```

### 5.6.2 Peterson Optimizat

```

1  public PetersonLockFair(int n) {
2      this.n = n;
3      level = new AtomicInteger[n];
4      victim = new AtomicInteger[n];
5      waiting = new AtomicBoolean[n];
6      accessCount = new AtomicInteger[n];
7      sharedCounter = 0;
8
9      for (int i = 0; i < n; i++) {
10         level[i] = new AtomicInteger(0);
11         victim[i] = new AtomicInteger(0);
12         waiting[i] = new AtomicBoolean(false);
13         accessCount[i] = new AtomicInteger(0);
14     }
15 }
16
17 private boolean shouldYield(int i) {
18     for (int j = 0; j < n; j++) {
19         if (j != i && accessCount[i].get() -
20             accessCount[j].get() >= THRESHOLD) {
21             return true;
22         }
23     }
24     return false;
25 }
26
27 public void lock(int i) {
28     waiting[i].set(true);
29
30     while (waiting[i].get() && shouldYield(i)) {
31         Thread.yield();
32     }
33
34     for (int L = 1; L < n; L++) {
35         level[i].set(L);
36         victim[L].set(i);
37         boolean otherThr;
38
39         do {
40             otherThr = false;
41             for (int k = 0; k < n; k++) {
42                 if (k != i && level[k].get() >= L) {
43                     otherThr = true;
44                     if (shouldYield(i)) {
45                         level[i].set(0);
46                         Thread.yield();
47                         level[i].set(L);
48                     }
49                 }
50             }
51             if (!otherThr) break;
52         } while (true);
53     }
54 }

```

```

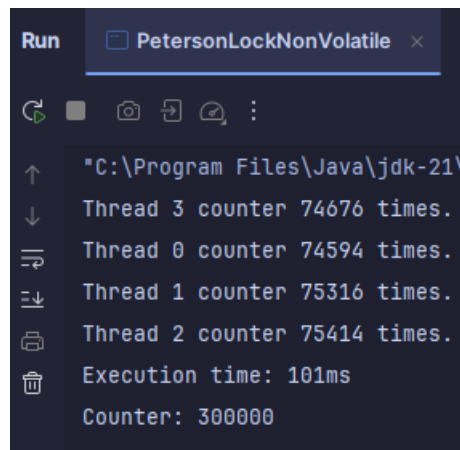
49         }
50     }
51     } while (otherThr && victim[L].get() == i);
52 }
53
54 waiting[i].set(false);
55 }
56
57 public void unlock(int i) {
58     accessCount[i].incrementAndGet();
59     level[i].set(0);
60 }

```

## 5.7 Exemplificare

In urma mai multor teste pentru fiecare algoritm am strans urmatoarele date

### 5.7.1 Peterson Clasic



```

Run PetersonLockNonVolatile x
C:\Program Files\Java\jdk-21\
Thread 3 counter 74676 times.
Thread 0 counter 74594 times.
Thread 1 counter 75316 times.
Thread 2 counter 75414 times.
Execution time: 101ms
Counter: 300000

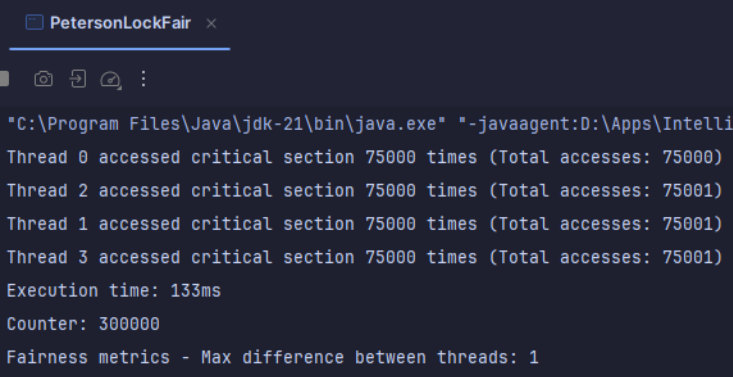
```

Figure 7: Peterson Clasic

$$t_{avg} = 85 \text{ ms}$$

$$n_{accesari} = 75.000 \pm 2.000 \text{ per } 300.000 \text{ operatii}$$

### 5.7.2 Peterson Optimizat



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:D:\Apps\Intelli
Thread 0 accessed critical section 75000 times (Total accesses: 75000)
Thread 2 accessed critical section 75000 times (Total accesses: 75001)
Thread 1 accessed critical section 75000 times (Total accesses: 75001)
Thread 3 accessed critical section 75000 times (Total accesses: 75001)
Execution time: 133ms
Counter: 300000
Fairness metrics - Max difference between threads: 1
```

Figure 8: Peterson Optimizat

$$t_{avg} = 135 \text{ ms}$$

$$n_{accesari} = 75.000 \text{ (constant) per } 300.000 \text{ operatii}$$