Temă Tehnici de Programare Multiprocesor [Hohleva Catalina, Ungureanu Ana-Maria, B4]

### 1. Se da următoarea secvența (istorie) de execuție de mai jos. Este aceasta linearizabila? Dar consistent secvențială? Se considera valoarea inițială r = 0.

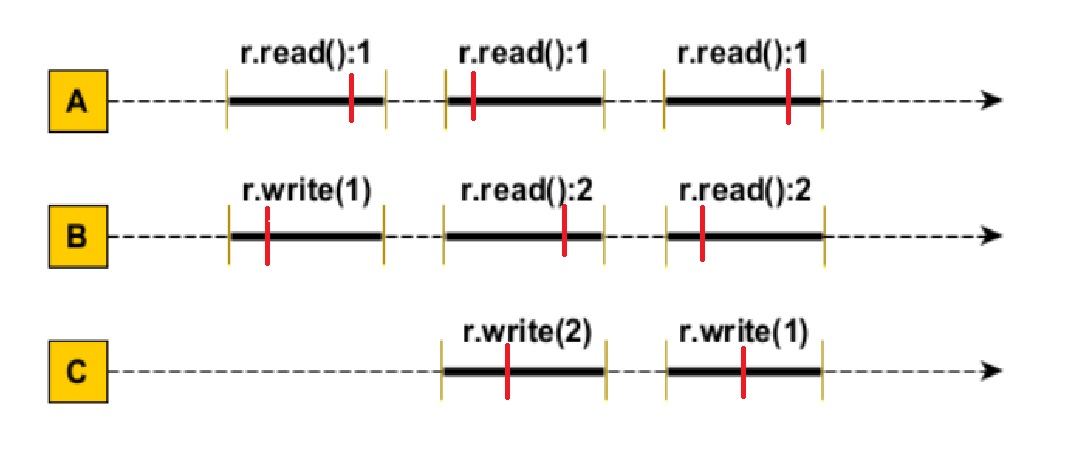
### Argumentați răspunsul oferind explicațiile (eventual secvența istoriei de execuție) și/sau o diagrama cu punctele de linearizare după caz.

### Răspuns:

* Valoarea inițială a lui r este 0.
* Thread-ul B efectuează r.write(1), deci valoarea lui r devine 1.
* Thread-ul A efectuează două operații de read, fiecare returnând valoarea 1.
* Thread-ul C efectuează r.write(2), deci valoarea lui r devine 2.
* Thread-ul B efectuează două operații de read, fiecare returnând valoarea 2.
* Thread-ul C efectuează r.write(1), deci valoarea lui r devine 1.
* Thread-ul A efectuează o operație de read ce returnează valoarea 1.

Prin urmare, această secvență este **linearizabilă**, deoarece rezultatul oricărei citiri

respectă ordinea în care au avut loc scrierile (putem ordona operațiile pentru a corespunde unei singure istorii atomice).

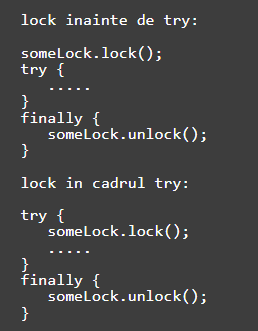


De asemenea, este și **consistent secvențială**, deoarece putem construi o ordine în care fiecare citire returnează ultima valoare scrisă (în conformitate cu ordinea scrierilor care preced acea citire).

O astfel de ordine poate fi cea enunțată în cazul linearizabilitatii sau în exemplul de mai jos:

* Inițializare: r = 0
* Thread B: r.write(1) ⇒ r = 1
* Thread A: Citiri succesive care returnează 1 (corect, deoarece ultima valoare scrisă este 1)
* Thread C: r.write(2) ⇒ r = 2
* Thread B: Citiri succesive care returnează 2 (corect, deoarece ultima valoare scrisă este 2)
* Thread C: r.write(1) ⇒ r = 1
* Thread A: Citire finală care returnează 1 (corect, deoarece ultima valoare scrisă este 1)

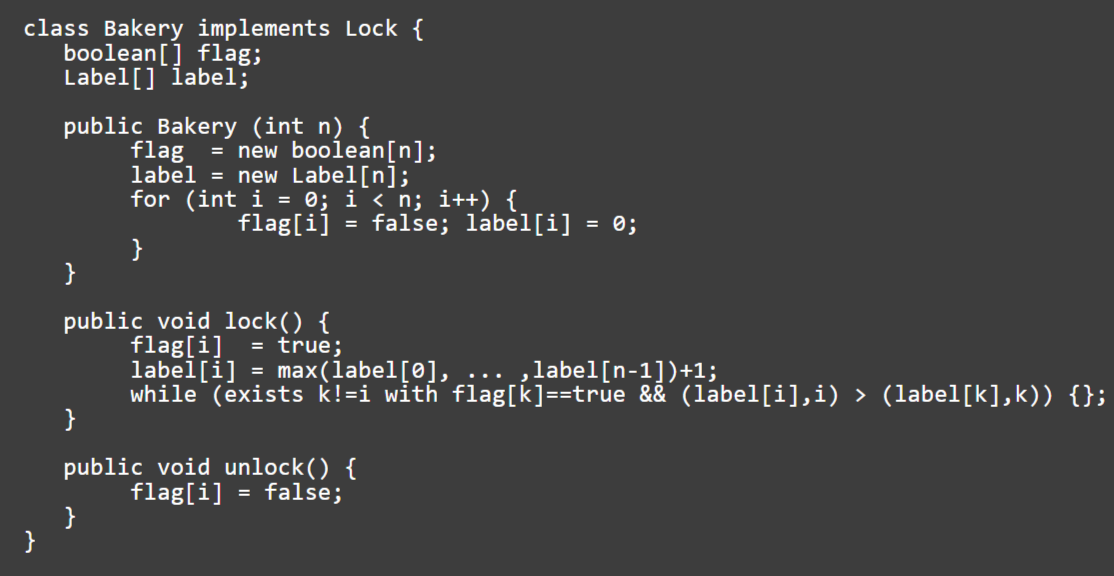
### 2. De ce în mod obișnuit în utilizarea unui lock se prefera ca apelul lock() să fie executat înainte de blocul try, și nu în cadrul acestuia (prima varianta de mai jos și nu a doua)? Argumentați.



Răspuns:

Utilizarea unui lock în afara blocului try este preferată pentru a se asigura că blocarea este garantată înainte de a intra în secțiunea critică de cod. În cazul în care apelul **lock()** ar fi plasat în interiorul blocului try, și ar avea loc o excepție înainte de a se ajunge la apelul **lock()**, atunci codul din finally va încerca să deblocheze un lock care nu a fost niciodată blocat, ceea ce poate duce la erori de execuție. De asemenea, exista posibilitatea ca atunci când apelăm **lock() în** blocul try să aibă loc o excepție, astfel se ajunge în finally și se încearcă deblocarea unui lock care nu a fost de fapt blocat. Plasarea **lock()** în afara blocului try garantează că blocarea este realizată înainte de orice operațiune și că **unlock()** este sigur să fie apelat în finally.

3. De ce în algoritmul Bakery (discutat în cursul 2, al cărui pseudocod e reamintit mai jos), în comparatia tuplelor din metoda lock (label[i],i) > (label[k],k) nu este utilizata doar comparatia etichetei (label) sau doar a indexului ce identifica threadul, ci e necesara comparatia tuplelor formate din ambele? Argumentați răspunsul descriind o situatie concreta pentru doua thread-uri care ar folosi doar etichetele sau indecsii în comparatia respectivă.



Răspuns:

**De ce nu este utilizata doar comparatia etichetei (label):**

while (exists k!=i with flag[k]==true && label[i] > label[k]) {};

Să presupunem că avem două thread-uri, Thread 0 și Thread 1, care încearcă simultan să acceseze secțiunea critică. Toate etichetele sunt inițial 0 (label[0] = 0; label[1] = 0), toate flag-urile - false (flag[0] = false; flag[1] = false).

Ce se întâmplă în acest caz?

1. Ambele thread-uri intră în metoda lock() în exact același moment și setează flag[i] = true pentru a semnala că doresc acces la secțiunea critică: flag[0] = true; flag[1] = true;
2. Ambele thread-uri simultan își calculează etichetele ca fiind max (label[0], label[1]) + 1

Toate etichetele sunt inițial 0,deci:

label[0] = max (label[0], label[1]) + 1 = max (0, 0) + 1 = 1

label[1] = max (label[0], label[1]) + 1 = max (0, 0) + 1 = 1

3. Ambele thread-uri verifică dacă există un thread în secțiunea critică cu o prioritate mai mare, comparând doar etichetele.

Pentru Thread 0:

∃k != 0 with flag[1] == true? Da: k == 1

Comparăm etichetele:

label[0] > label[1] ?

1 > 1? - False ⇒ Thread 0 intră în secțiunea critică.

Pentru Thread 1:

∃k != 1 with flag[0] == true? Da: k == 0

Comparăm etichetele:

label[1] > label[0] ?

1 > 1? - False ⇒ Thread 1 tot intră în secțiunea critică.

Astfel, ambele thread-uri intră simultan în secțiunea critică ⇒ algoritmul bazat doar pe comparația etichetelor (label[i] > label[k]) nu poate asigura excluderea mutuală în cazul în care thread-urile încearcă să acceseze secțiunea critică în exact același moment.

**De ce nu este utilizata doar comparatia indexului:**

while (exists k!=i with flag[k]==true && i > k) {};

Utilizarea exclusivă a indexului în comparație introduce următoarea problemă: thread-urile cu index-uri mai mici au întotdeauna prioritate față de cele cu index-uri mai mari, indiferent de momentul real al sosirii lor, ceea ce împiedică respectarea ordinii corecte de acces la secțiunea critică și duce la situații, în care thread-urile sosite mai târziu (cu index-uri mai mici) accesează secțiunea critică în mod repetat, în timp ce cele sosite mai devreme (cu index-uri mai mari) rămân blocate.

Să presupunem că avem trei thread-uri, Thread 0 (i = 0), Thread 1 (i = 1) și Thread 2 (i = 2).

1. Thread-ul 2 ajunge primul la metoda lock:
   * Setează flag[2] = true și își calculează eticheta (dar aceasta nu contează, deoarece se compară doar indexurile).

Thread-ul 1 ajunge imediat după Thread-ul 2:

* + Setează flag[1] = true și își calculează eticheta.

Thread-ul 0 ajunge după Thread-ul 1:

* + Setează flag[0] = true și își calculează eticheta.

2. Thread-ul 2 trebuie să aștepte, deoarece:

* + Pentru Thread 0 (i = 0): i > k ⇒ 2 > 0 ⇒ True.
  + Pentru Thread 1 (i = 1): i > k ⇒ 2 > 1 ⇒ True.

3. Thread-ul 0 intră în secțiunea critică, finalizează și setează flag[0] = false.

4. Apoi, Thread-ul 1 are prioritate față de Thread-ul 2 (i = 1 < i = 2) și intră.

5. Thread-ul 0 revine înainte ca Thread-ul 2 să intre:

* + Dacă Thread-ul 0 reia execuția înainte ca Thread-ul 2 să intre, acesta are din nou prioritate (i = 0), menținând Thread 2 blocat.

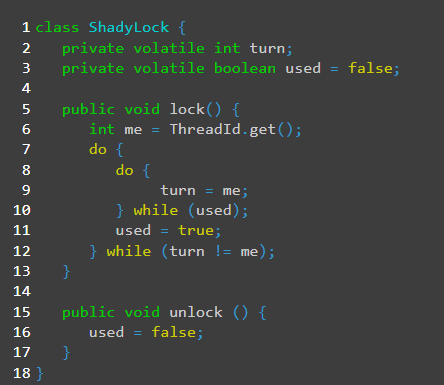
Astfel, Thread-ul 2 (cu cel mai mare index) poate rămâne blocat pe tot parcursul programului dacă utilizăm exclusiv indexul în comparație.

**Concluzie:**

Compararea tuplelor (label[i], i) permite algoritmului Bakery să asigure atât excluderea mutuală, cât și corectitudinea ordinii de acces la secțiunea critică. Această abordare combină ordinea de sosire (prin etichete) cu un criteriu secundar unic (indexul thread-ului), eliminând ambiguitatea în caz de egalitate a etichetelor. Astfel, algoritmul previne atât accesul simultan, cât și situațiile de starvation sau inversări incorecte ale priorității.

### 4. a) O echipa de programatori a dezvoltat algoritmul de lock prezentat în pseudocodul următor. ThreadId se considera a fi o clasa ce furnizează un id unic pozitiv fiecărui thread.

### Într-o execuție concurenta a n > 1 thread-uri, este acest algoritm deadlock-free? Argumentați. În argumentație se pot include și eventuale trace-uri demonstrative pentru execuția unor thread-uri sau opțional o implementare exemplificativa dacă este cazul.



Răspuns:

În acest algoritm, variabila **turn** permite unui thread să revendice accesul la secțiunea critică, iar **used** este utilizată pentru a indica dacă secțiunea critică este ocupată. Deși un thread poate fi forțat să aștepte din cauza altor thread-uri care concurează pentru lock, există situații în care un thread poate fi blocat permanent, în funcție de ordinea execuției.

Problema apare din faptul că un thread poate seta **turn = me**, dar un alt thread poate suprascrie imediat această valoare. Dacă acest lucru se repetă ciclic, un thread ar putea să nu mai ajungă niciodată să îndeplinească condițiile pentru a intra în secțiunea critică. În astfel de cazuri, execuția poate ajunge într-o stare în care unul sau mai multe thread-uri sunt blocate definitiv.

Prin urmare, acest algoritm **nu** este **deadlock-free.**

### **Analiză detaliată cu trace pentru deadlock-free:**

Vom urmări un scenariu cu trei thread-uri: **T1**, **T2** și **T3**.

1. **Thread T1 intră în secțiunea critică:**

* T1 setează **turn = 1**, **used = true**, și intră în secțiunea critică.

1. **Thread T2​ intervine:**

* T2​ setează imediat **turn = 2**. În acest moment, T1​ vede că **turn != 1** și este forțat să reia bucla.

1. **Thread T3​ intervine:**

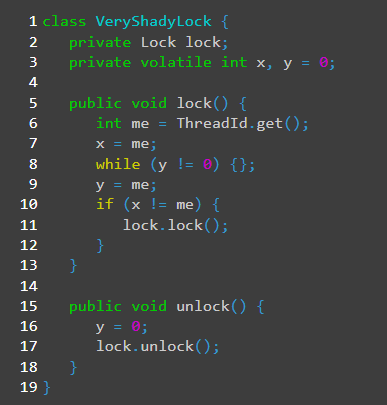
* T3​ setează imediat **turn = 3**, blocând astfel și T2.

1. **Thread-urile rămân blocate într-o stare ciclică:**

* Fiecare thread continuă să rescrie **turn**, iar niciunul dintre ele nu reușește să îndeplinească simultan condițiile **turn == me** și **used = false**. Toate thread-urile rămân blocate în bucla interioară, fără să progreseze.

### b) O alta echipa de programatori a dezvoltat algoritmul de lock prezentat în pseudocodul următor ce incapsuleaza un alt lock oarecare. Se considera ca lock-ul incapsulat asigura corect excluderea mutuala și este starvation-free. De asemenea lock-ul incapsulat permite un apel unlock fără excepție și fără efect chiar dacă nu a existat un apel lock. ThreadId se considera a fi o clasa ce furnizează un id unic pozitiv fiecărui thread.

### Într-o execuție concurenta a n > 1 thread-uri, asigura acest algoritm o garanție de fairness (niciun thread nu poate accesa o secțiune critică protejata de lock mai des ca altele)? Argumentați. În argumentație se pot include și eventuale trace-uri demonstrative pentru execuția unor thread-uri sau opțional o implementare exemplificativa dacă este cazul.



Răspuns:

### **Analiză detaliată cu trace pentru fairness:**

Să presupunem că avem 3 thread-uri care concurează pentru secțiunea critică: **T1**, **T2** și **T3**.

#### **1. T1 începe:**

* T1 setează **x = 1** și intră în bucla de așteptare **while (y != 0)**.
* La început, **y = 0**, deci T1 iese din bucla de așteptare și setează **y = 1** (adică T1 este acum în secțiunea critică).
* T1 poate accesa secțiunea critică.

#### **2. T2 încearca sa intre in sectiunea critica:**

* T2 setează **x = 2** și intră în bucla de așteptare.
* La acest moment, **y = 1** (deoarece T1 a intrat în secțiunea critică), așa că T2 așteaptă ca y să devină 0.
* T2 este blocat și nu poate continua până când T1 nu iese din secțiunea critică.

#### **3. T3 încearca sa intre in sectiunea critica:**

* T3 setează **x = 3** și intră și el în bucla de așteptare.
* Similar cu T2, T3 așteaptă ca y să devină 0 (deoarece T1 este încă în secțiunea critică).

#### **4. T1 iese din secțiunea critică:**

* T1 finalizează secțiunea critică și apelează **unlock()**, setând **y = 0**.
* Acum, **T2 și T3** pot încerca să intre în secțiunea critică.

#### **5. Conflict intre T2 si T3:**

* Atat **T2**, cat și **T3** detectează că **y = 0** și concurează pentru a seta **y**.
* Dacă **T3** reușește să seteze **y = 3** înaintea lui **T2**, **T3** intră în secțiunea critică, deși **T2** aștepta de mai mult timp.

#### **6. T3 iese din secțiunea critică:**

* **T2** finalizează secțiunea critică și apelează **unlock()**, setând **y = 0**.
* Acum, **T1 și T2** pot încerca să intre în secțiunea critică.

**7. T1 poate accesa din nou secțiunea critică:**

* **T1** poate încerca să intre în secțiunea critică și poate reuși sa seteze **y = 1** înaintea lui **T2** sau **T3**.

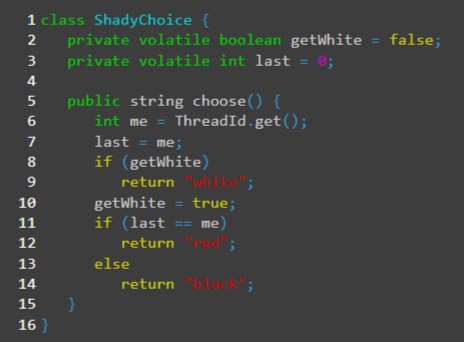
Algoritmul **nu asigură fairness**, deoarece:

* Nu exista o coada care sa asigure ca thread-urile sunt servite în ordinea în care au solicitat lock-ul.
* Orice thread care detectează **y = 0** poate încerca sa seteze **y**, iar ordinea depinde de alți factori externi (timing, viteza CPU).
* Thread-urile care finalizează rapid pot reuși sa acceseze secțiunea critică mai frecvent decat cele care așteaptă, ceea ce încalcă noțiunea de **fairness**.

Acest algoritm **nu asigură fairness absolut**. Într-un scenariu concurent, un thread poate accesa secțiunea critică de mai multe ori înaintea altor thread-uri, iar ordinea de acces nu este garantată.

### c) O a treia echipa de programatori s-a plictisit de programat lacate, si a implementat clasa din pseudocodul următor. Considerăm un context de execuție concurenta a n > 1 thread-uri, în care fiecare thread apelează metoda choose. ThreadId se considera a fi o clasa ce furnizează un id unic pozitiv fiecărui thread.

### Demonstrati ca maxim un thread poate obține valoarea "red" și maxim n-1 thread-uri pot obține valoarea "white".



Răspuns:

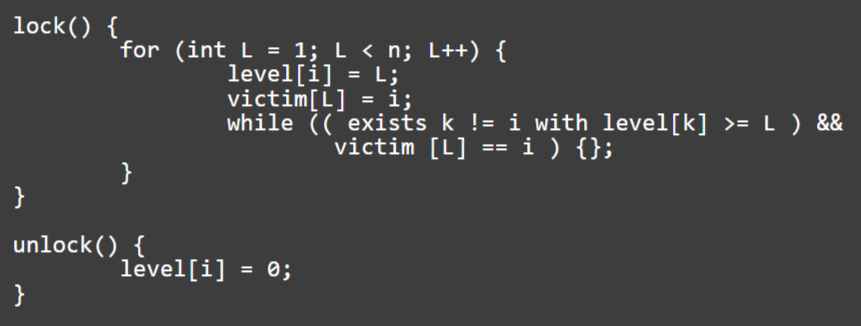
#### **1. Maxim un thread poate obține "red":**

* Primul thread care rulează metoda **choose** va obține "**red**" deoarece acesta setează **last = me**, **getWhite = true**, și verifică condiția **last == me**, care este adevărată în cazul primului thread, deoarece last a fost setat la valoarea sa imediat înainte de verificare.
* Datorită setării **getWhite = true**, niciun alt thread nu va mai putea ajunge la această condiție pentru a obține "**red**".

#### **2. Maxim n−1 thread-uri pot obține "white":**

* Odată ce **getWhite** este setat la **true** de primul thread, toate thread-urile care apelează metoda ulterior vor returna "**white**" conform condiției **if (getWhite)**.
* Într-un context cu n thread-uri, dacă unul dintre ele obține "**red**", atunci restul de n−1 thread-uri vor obține "**white**".

5. Considerăm algoritmul lui Peterson de excludere mutuala ce oferă o generalizare pentru n thread-uri, ale cărui metode sunt redate în pseudocodul de mai jos. Ideea acestuia este de a trece fiecare thread printr-un filtru de n-1 nivele pana la accesul la secțiunea critică. i poate fi considerat ca identificator al unui thread iar L ca număr al nivelului. Tabloul level asociat nivelelor reține nivelul curent pentru fiecare thread, iar tabloul victim reține identificatorul fiecărui ultim thread ce a avansat la respectivul nivel.



Găsiți o varianta de îmbunătățire a algoritmului pentru a asigura ca și în aceasta forma valoarea r ce cuantifica așteptarea este 0 (0-bounded waiting), sau altfel zis algoritmul asigura fairness.

Argumentați îmbunătățirea adusă și implementați algoritmul în acest mod pentru a demonstra practic aceasta proprietate.

Odată pornite n thread-uri niciunul dintre acestea nu ar trebui să poată accesa secțiunea critică protejata de lock mai des ca celelalte.

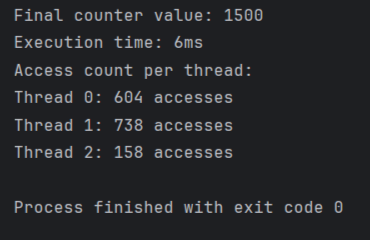
Nota: Implementarea ar trebui sa imbunatateasca algoritmul dat, adică sa pastreze logica acestuia în scopul de locking, nu sa o transforme printr-un alt lock care este fair (ex., folosind Bakery).

Răspuns:

Google Drive Link - Programul, care implementează algoritmul lui Peterson de excludere mutuala de pe poza de mai sus:

<https://drive.google.com/file/d/1mL4YrFT9zzn88arZZbu5EZRImkOB9WVa/view?usp=sharing>

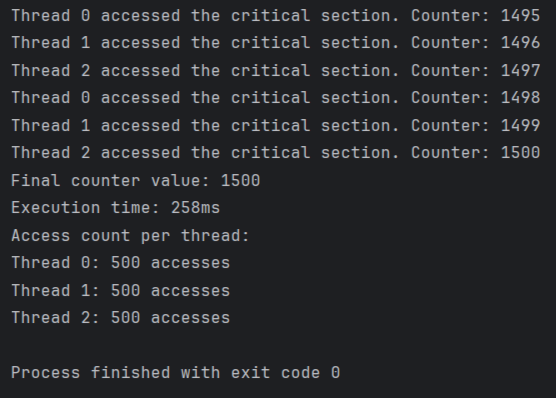
Output:



Google Drive Link - Programul, care implementează algoritmul lui Peterson îmbunătățit, ce asigura fairness:

<https://drive.google.com/file/d/1XR_tSkrCRTDhGovhRtBBGC6lhe5JQqJJ/view?usp=sharing>

Output:



Pseudocodul programului îmbunătățit:

lock() {

// Așteaptă să fie rândul thread-ului i conform fairAccessIndex

while (fairAccessIndex % n != i) {};

for (int L = 1; L < n; L++) {

level[i] = L;

victim[L] = i;

while (( exists k != i with level[k] >= L ) &&

victim [L] == i ) {};

}

}

unlock() {

level[i] = 0;

fairAccessIndex++; // Incrementare pentru a permite următorului thread să intre

}

Cum fairAccessIndex rezolvă problema de fairness:

fairAccessIndex determină ordinea strictă în care thread-urile accesează secțiunea critică. În fiecare iterație, doar thread-ul al cărui index corespunde cu fairAccessIndex % n poate intra. După ce finalizează execuția, acesta incrementează fairAccessIndex, transferând controlul următorului thread din rotație. În acest mod, fiecare thread primește exact o șansă pe rând să intre în secțiunea critică, evitând accesul preferențial, blocajele și riscul de starvation.

În cod, valoarea fairAccessIndex este incrementată după fiecare ieșire din secțiunea critică (în metoda unlock). Apoi, în metoda lock, fiecare thread verifică dacă este rândul său de a intra, folosind următoarea condiție:

*while (fairAccessIndex.get() % n != i) {*

*}*

Condiția fairAccessIndex % n != i asigură că un thread poate intra în secțiunea critică doar când fairAccessIndex este un multiplu al valorii sale de index (i). Astfel, accesul la secțiunea critică este strict ordonat.

*De exemplu:*

*Dacă există 3 thread-uri (n = 3), iar fairAccessIndex = 4, atunci:*

*fairAccessIndex % n = 4 % 3 = 1 → Este rândul thread-ului cu index 1 să intre.*

*După ce thread-ul 1 finalizează execuția și fairAccessIndex este incrementat la 5:*

*fairAccessIndex % n = 5 % 3 = 2 → Este rândul thread-ului cu index 2 să intre.*

*Și astfel mai departe, rotația continuă.*

Dacă fairAccessIndex nu corespunde indexului thread-ului curent (i), acesta va intra într-un busy wait până când alte thread-uri termină și variabila fairAccessIndex este incrementată.

După ce un thread finalizează execuția în secțiunea critică și iese (prin metoda unlock), fairAccessIndex este incrementat:

*fairAccessIndex.incrementAndGet();*

Acest pas transferă controlul către următorul thread din ordine.

Întrucât valoarea fairAccessIndex determină rândul fiecărui thread, niciun thread nu poate intra de două ori consecutiv în secțiunea critică dacă există alte thread-uri care așteaptă rândul lor. Acest lucru elimină riscul ca un thread să acceseze prea des sau să blocheze accesul altora.