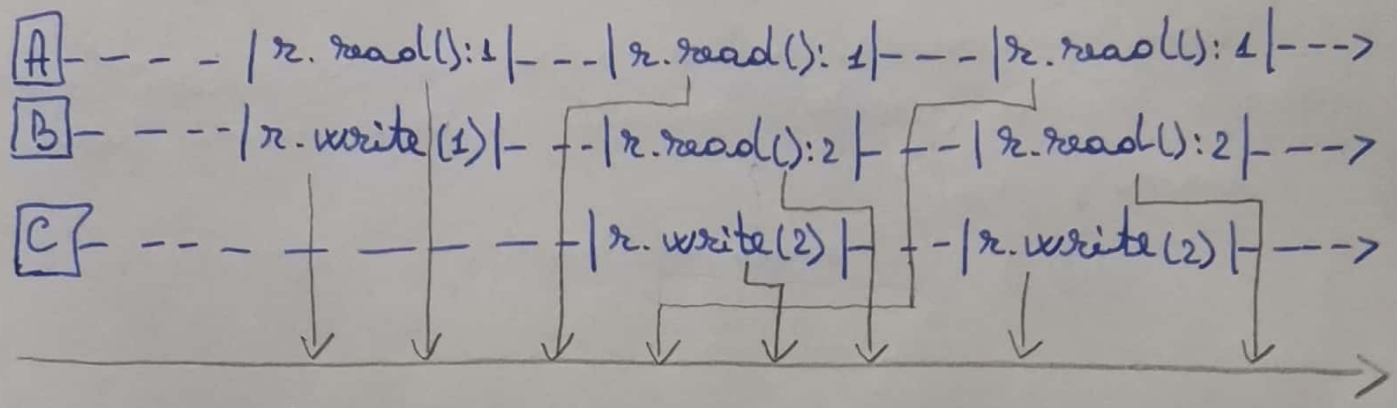
TEMĂ

Exercițiul 1

Inițial $x = 0$



Secvența este consistent secvențială, deoarece putem găsi o permutare a tuturor operațiilor astfel încât rezultatele să fie aceleași ca și cum ar fi fost efectuate într-o singură secvență de operații.

Secvența nu este liniarizabilă, deoarece nu putem găsi o ordine liniară astfel încât să se respecte ordinea în timp a operațiilor.

Demonstrație:

caz 1: inițial $x = 0$

dacă A citește 1 $\Rightarrow$ nu se poate, deoarece niciun thread nu a scris încă 1.

caz 2: inițial $x = 0$

B scrie 1
A citește 1
A citește 1
C scrie 2
B citește 2

iar acum am ajuns în punctul în care A va fi nevoit să citească, la un moment dat, 1, dar în variabila r se va găsi doar valoarea 2.

Exercițiul 2:

Există mai multe motive pentru care se preferă ca apelul lock() să fie executat înainte de try, unul fiind:

```
avem: try {
          someLock.lock();   <-
          ---
      }
      finally {
          someLock.unlock();
      }
```

în cazul de mai sus, someLock.lock() ar putea arunca o excepție, deci nu se pune lacătul, iar în blocul finally va încerca să dea unlock, în condițiile în care nu a fost făcută blocarea =>
=> IllegalMonitorStateException.

Așadar, prima variantă este preferată (cu lock() înainte de blocul try), deoarece asigură că lock-ul este întotdeauna eliberat, indiferent de ceea ce se întâmplă în blocul try, și pentru că convențiile de programare, conform cărora: blocul try este utilizat pentru a încerca o acțiune ce poate arunca o excepție, iar blocul finally este utilizat pentru a asigura că anumite resurse sunt eliberate, indiferent de ce se întâmplă în blocul try.

Exercițiul 3

Există o problemă în abordarea prezentată în cadrul exercițiului, și anume:

dacă coada este plină (count == tail.length), producătorul așteaptă până când un loc devine disponibil. După ce un element este adăugat, coada devine nevidă și se trimite semnal către consumatori.

Înainte ca vreun consumator să poată răspunde la semnalul anterior, un al doilea producător adaugă încă un element în coadă, dar coada fiind deja cu mai mult de 1 element, nu va mai trimite mesaj.

Un consumator va reacționa la semnalul trimis de primul producător și elimină primul element pus în coadă.

Problema apare la al doilea consumator, deoarece acesta nu va primi niciun semnal.

Așadar, problema apare atunci când doi producători adaugă, în paralel, elemente în coadă, astfel trimițându-se un singur semnal, ce va ajunge la un singur consumator.

Exemplu: doi producători și doi consumatori:

P_1: adaugă element$_1$ ⇒ coada devine nevidă ⇒ trim. semnal

P_2: adaugă element$_2$: coada era deja nevidă, deci nu se trimite mesaj

C_1: elimină element 1 : deoarece a primit semnal de la P_1.

C_2: așteaptă, deoarece nu a primit niciun semnal.

Exercițiul 4

a) În urma analizei secvenței de cod, am observat o problemă în implementarea propusă:

- deoarece nu există o logică de așteptare atunci când coada este plină : while (tail - head == QSIZE) {}; , sau goală : while (tail == head) {}; firele de execuție vor rămâne blocate.

- de asemenea putem întâlni și fenomenul de starvation : în momentul în care un fir încearcă să adauge un element în coada plină, rămâne blocat în bucla while, chiar dacă un consumator va scoate un element din coadă, thread-ul blocat nu va fi notificat.

Exemplu :

am implementat programul, iar în urma unei rulări cu 4 producători și 4 consumatori, QSIZE = 5, rezultatul este următorul :

P_1 adaugă 10
P_2 adaugă 20
P_3 adaugă 0
P_4 adaugă 30

C_1 elimină 10
C_2 elimină 20
C_3 elimină 0

C_4 elimină 30

P_3 adaugă 31

P_1 adaugă 11

P_2 adaugă 21

P_4 adaugă 1

C_2 elimină 11

C_3 elimină 31

C_1 elimină 21

C_4 elimină 1

P_1 adaugă 12

P_4 adaugă 2

P_2 adaugă 22

P_3 adaugă 32

P_3 adaugă 33

iar aici firele rămân blocate.

b) În urma analizei programului, am observat o problemă în implementare, și anume:

- verificarea dacă coada este plină / goală se face înainte de enqlock.lock() / deqlock.lock(), ceea ce permite mai multor thread-uri să treacă de verificările: while (tail - head == QSIZE) {}; respectiv while (tail == head) {};

- presupunem că 2 thread-uri încearcă simultan să adauge în coadă, unul dintre ele obține lacătul, adaugă elementul, iar celălalt așteaptă ca lacătul să fie eliberat; dacă coada devine

plină, la obținerea lacătului, cel de-al doilea thread va suprascrie un element din coadă, deoarece era deja trecut de condiția din while.

Exemplu:

Pp. că avem: qsize = 3

→ în momentul curent avem 1 singur loc în coadă (deci nr. de elemente în coadă este 2) și avem 2 thread-uri ce au trecut de while(--).

pas 1: while(tail-head == qSIZE){};
← T_1, T_2

englock.lock();

pas 2: while(----){};
englock.lock() ← T_1 (obține lacătul)
T_2 (așteaptă)

pas 3: try {
items[2] = element_pus_de_T_1
tail = 3;
} finally { englock.unlock(); }

pas 4: englock.lock() ← T_2 primește și el lacătul

pas 5: try {
items[3%3] = element_pus_de_T_2
(observăm că T_2 suprascrie items[0])

Exercițiul 5:

a) Presupunem prin reducere la absurd că algoritmul nu asigură excludere mutuală.
Vom considera două thread-uri: T_1 și T_2 care încearcă să intre în secțiunea critică.
Dacă proprietatea de excludere mutuală nu este asigurată, atunci access[T_1] = access[T_2] = true, ceea ce este imposibil, deoarece conform instrucțiunii: await(every j != i has (flag[j] == false || label[j] > label[i])), threadul cu eticheta mai mică va putea depăși acest await și va seta flagul access corespunzător la true, iar celălalt va face spin în instrucțiune.

Astfel, presupunerea făcută a dus la o contradicție, deci este falsă. ⇒ am demonstrat că algoritmul asigură excludere mutuală.

b) Deadlock-free:

Algoritmul evită blocajele prin utilizarea etichetelor pentru a ordona cererile de acces la resurse.
Dacă un proces este în stare de așteptare (flag[i] = true), acesta va intra în secțiunea critică dacă:
restul proceselor au label-uri mai mari (ceea ce înseamnă că au avut acces la resurse, motiv pt. care label-ul a crescut) sau dacă procesele cu

label mai mic nu încearcă să intre în secțiunea critică.

Acest lucru asigură că întotdeauna există cel puțin un proces ce poate intra în secțiunea critică.

Starvation-free:

O situație de starvation ar putea apărea dacă există măcar un thread care să nu obțină acces la resurse. Datorită etichetării și a condiției await, fiecare proces va avea o șansă să ajungă în secțiunea critică. Presupunem că există un thread t ce dorește să intre în secțiunea critică și toate celelalte au o prioritate mai mare față de el. Pe măsură ce aceste thread-uri vor intra în secțiunea critică, label crește, iar prioritatea scade și astfel la un moment dat restul thread-urilor vor ajunge să aibă prioritate mai mică față de thread-ul t ce dorea să intre în secțiunea critică, astfel thread-ul t va primi acces la resursă.

Așadar, nu poate exista situație de starvation.