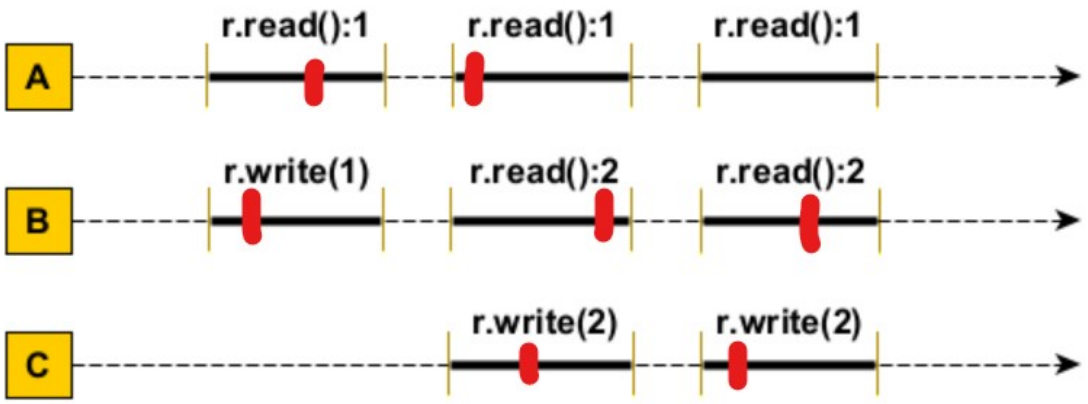
Tema TPM

Aprofirei Adrian - Nastase Valentin - Teodorescu Calin

1.



Ca o secventa sa fie linearizabila, trebuie ca operatiile din interiorul acesteia

Secventa de executie va fi urmatoarea: B:r.write(1) -> A:r.read():1 -> A:r.read():1 -> C:r.write(2) -> B:r.read():2 -> C:r.write(2) -> B:r.read():2. Ultima instructiune a thread-ului A nu va putea avea loc, intrucat r are valoarea 2 acum. Deci, secventa nu este linearizabila.

Instructiunile pot fi reordonate, astfel incat executia acestora sa aiba sens. Astfel, secventa finală este B:r.write(1) -> A:r.read():1 -> A:r.read():1 -> A:r.read():1 -> C:r.write(2) -> B:r.read():2 -> C:r.write(2) -> B:r.read():2. Prin urmare, secventa este consistent secventiala.

2.

Apelul lock() ar trebui executat inainte de blocul try deoarece daca se arunca o exceptie in metoda lock(), apelul unlock() din blocul finally va fi executat, lucru care ar putea duce la un comportament nedefinit, intrucat unlock() trebuie apelat doar dupa un apel lock() care s-a terminat cu succes.

3.

```
1           public void enq(T x) {
2                lock.lock();
3                try {
4                        while(count == items.length)
5                        {
6                           try { full.await(); }
7                           catch (InterruptedException e) { }
8                           finally { }
9                        }
10                       System.out.println("Enqueuing " + x);
11                       items[tail] = x;
12                       if (++tail == items.length) { tail = 0; }
13                       count++;
14                       if (count == 1) {
15                              empty.signal();
16                       }
17               } finally {
18                   lock.unlock();
19               }
20          }
21
22          public T deq() {
23               lock.lock()
24               try {
25                       while (count == 0)
26                       {
27                          try { empty.await(); }
28                          catch (InterruptedException e) { }
29                          finally { }
30                       }
31                       char x = items[head];
32                       if (++head == items.length) { head = 0; }
33                       count--;
34                       System.out.println("Dequeuing " + x);
35                       if (count == items.length - 1) {
36                              full.signal();
37                       }
38                       return x;
39               } finally {
40                   lock.unlock();
41               }
42          }
```

Pentru a decide daca optimizarea propusa contine vreo problema, vom simula executarea programului avand coada initiala vida, cel putin 2 thread-uri producator si 2 thread-uri consumator

Presupunem ca rulam programul cu 2 thread-uri producator pentru enq si 2 consumator pentru deq. Initial, coada este vida, iar cele doua thread-uri consumator vor accesa lock-ul si vor ajunge pana la linia 27, unde vor astepta un semnal. In continuarea executiei, primul thread producator va obtine lock-ul si va accesa zona critica, va incrementa count-ul (count = 1), va adauga elementul curent in coada (linia 11) si va trimite un semnal unui singur thread deq.

In acest moment al executiei, lock-ul poate fi accesat de oricare dintre cel de-al doilea thread enq si thread-ul deq care tocmai a primit semnal.

Cel de-al doilea thread enq va intra in lock inaintea thread-ului deq care a primit semnal, va adauga urmatorul element in coada si va incrementa count-ul (count = 2). Comparatia de la linia 14 va avea de data aceasta valoare falsa, prin urmare nu va mai trimite semnal thread-ului deq care inca asteapta la linia 27.

In aceasta maniera, cele 2 thread-uri producator si primul thread consumator isi vor fi terminat executia, iar cel de-al doilea thread consumator va ramane blocat in await-ul de la linia 27.

4.

a) Daca coada este plina si un thread vrea sa efectueze o operatie enq, acesta va ramane blocat la linia 9 si nu va elibera lock-ul, ceea ce inseamna ca niciun alt thread nu va putea efectua operatia deq pentru a goli coada. Acest lucru va duce la un deadlock. Similar, aceeasi situatie poate aparea si in cazul operatiei deq, unde un thread blocat la linia 20 nu va elibera lock-ul, impiedicand orice alt thread sa efectueze operatia enq si sa adauge elemente in coada, generand tot un deadlock.

b) In cazul in care doua threaduri incearca sa efectueze operatia enq simultan, ambele threaduri vor ramane blocate la linia 8, deoarece verificarea pentru coada plina va fi adevarata pentru ambii. Daca in acelasi timp un alt thread efectueaza operatia deq si goleste coada, threadurile blocate la enq vor iesi din bucla while si vor incerca sa adauge elemente intr-o coada in care incape un singur element. Acest lucru va duce la o eroare.

5.

```
1  function init() {
2          for (int k = 0; k < n; k++) {
3                  flag[k] = false;
4                  access[k] = false;
5                  label[k] = k + 1;
6          }
7  }
8
9  function lock(i) {
10         flag[i] = true;
11         do {
12                 access[i] = false;
13                 await ( every j != i has ( flag[j] == false || label[j] > label[i] )) {};
14                 access[i] = true;
15         } while ( exists j != i with access[j] == true );
16 }
17
18 function unlock(i) {
19         label[i] = max(label[0],...,label[n-1]) + 1;
20         access[i] = false;
21         flag[i] = false;
22 }
```

a) Presupunem prin reducere la absurd ca 2 threaduri au reusit sa obtina lock-ul. Asta ar insemna ca exista doua pozitii i, j, astfel incat access[i] = true, access[j] = true. Dar, acest lucru nu este posibil datorita liniei 15, care permite unui thread sa iasa din while, doar atunci cand este singurul thread pentru care access este true. Deci, algoritmul propus asigura excluderea mutuala.

b)

Deadlock-free:

Presupunem ca n threaduri incearca sa dea lock simultan. Prin urmare, vom avea n - 1 threaduri care vor ramane blocate la linia 13 si un thread care are label-ul cel mai mic, care va obtine acces la lock. Odata ce acesta da unlock, va primi un label mai mare decat maximul celorlalte n - 1 threaduri. Apoi, unul din cele n - 1 threaduri va obtine acces la lock, deoarece va avea label-ul minim. Acest proces se repeta, ceea ce garanteaza proprietatea de deadlock-free a algoritmului.

Starvation-free:

Presupunem ca n threaduri incearca sa obtina lock-ul simultan. Acestea vor avea label-urile 1, 2, .. ,n. Initial, thread-ul cu label n este cel mai putin prioritar. Dar, pe masura ce celelate n - 1 threaduri dau unlock, acestea vor primi label-uri mai putin prioritare decat al threadului n (n + 1, n + 2, .., n + (n - 1)). Deci, dupa ce celelalte n - 1 threaduri au dat unlock, threadul n va fi cel mai prioritar si va primi acces la zona critica. Astfel, orice thread va primi la un moment dat acces la zona critica, de unde reiese faptul ca algoritmul este starvation-free