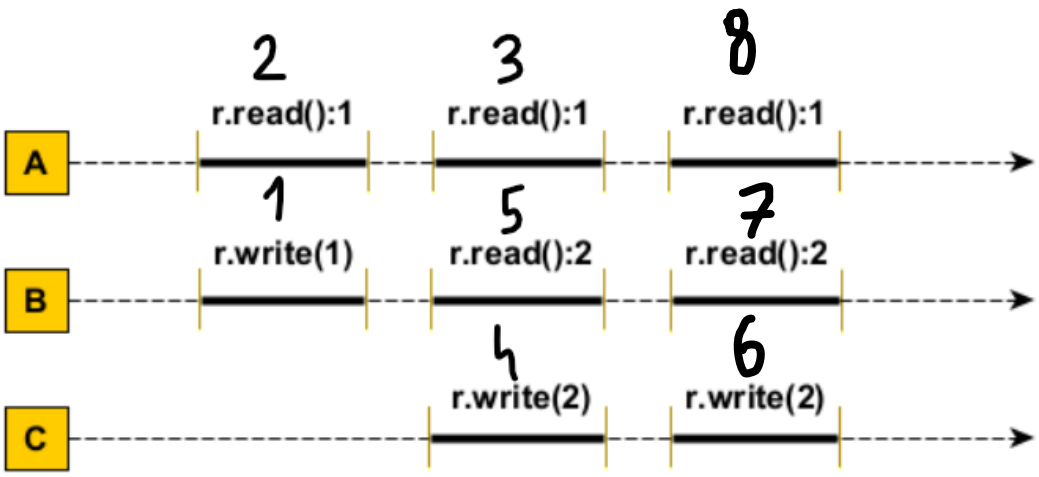
Tema TPM

Roman Lidia Lavinia
Briulung Bianca Ioana
Grigore Valerian

1.



Diferenta cheie intre linearizabilitate si consistenta secventiala este ca linearizabilitatea trebuie sa respecte ordinea in timp real iar consistenta secventiala nu.

In exemplul de mai sus operatiile nu respecta ordinea in timp real.

De exemplu:

B scrie 1 la momentul 1, dar A inca citeste valoarea veche (1) la momentele 2 și 3. Aceasta este o incalcare a ordinei in timp real, deoarece o citire ulterioara nu ar trebui sa returneze o valoare mai veche decat cea scrisa anterior. Prin urmare, acest sistem nu este linearizabil.

Operatiile respectă ordinea programului pentru fiecare proces individual.

De exemplu:

B scrie 1 la momentul 1, apoi citeste 2 la momentul 5 si 7, ceea ce este in conformitate cu ordinea programului pentru procesul B. A citeste 1 la momentele 2, 3 si 8, ceea ce este in conformitate cu ordinea programului pentru procesul A. C scrie 2 la momentele 4 și 6, ceea ce este in conformitate cu ordinea programului pentru procesul C.

Cu toate acestea, este secventiala deoarece ordinea de executie intre diferite procese este nedefinita, ceea ce este permis în consecventa secventiala.

2.

In general se prefer asezarea apelului look() in afara blocului try pentru a evita situatia in care o exceptie ar putea sa scape din blocul try fara a fi gestionata si fara a debloca sursa.

Daca ai plasa lock() in interiorul blocului try si o exceptie ar fi aruncata inainte de aplelul lock(), atunci blocul finally nu ar fi executata si resursa ar fi blocata. Aceasta ar putea duce la

problem de concurenta, in care resursa ramane blocata si nu poate fi accesata corect de catre alte thread uri.

Plasand lock() in afara blocului try, te asigurri ca resursa nu ramane blocata inainte de inceperea executiei blocului try. Astfel, chiar daca o exectie apare in blocul try, finally este garantat sa fie executat si sa elibereze resursa prin apelul unlock.

Acest mod de gestionare a lock-urilor contribuie la prevenirea problememlor legate de concurenta si la asigurarea unui utilizari corecte a resurselor partajate intre thread-uri.

3.

Da, exista o problema in aceasta abordare, si anume potentialul pentru o situatie de blocare (deadlock). Problema apare atunci cand sunt implicate mai multe fire de executie (thread-uri) si se petrec anumite evenimente in ordinea specificata. In particular, exista o problema de sincronizare intre semnalele si verificarile conditiilor in `enq`.

Sa analizam o situatie specifica in care coada este initial vida si exista cel putin doua fire de executie producatoare (adaugare) si doua fire de executie consumatoare (eliminare).

1. Firul de executie P1 (Producator 1) incearca sa adauge un element. Coada este vida (`count == 0`), si P1 trece de verificarea initiala `while(count == items.length)`.

2. P1 adauga un element in coada si incrementeaza `tail`, apoi verifica daca `tail` a atins capatul circular al cozi (`tail == items.length`).

3. In cazul in care `tail == items.length`, P1 incrementeaza `tail` si continua, apoi semnalizeaza `empty` pentru a anunta consumatorii ca coada nu mai este vida (`if (count == 1) { empty.signal(); }`).

4. Intre timp, firul de executie C1 (Consumator 1) observa ca coada nu este vida (`count == 1`) si trece de verificarea `while (count == 0)`.

5. C1 incepe sa extraga un element din coada, iar `head` este incrementat.

6. Intre timp, firul de executie P2 (Producator 2) incearca sa adauge un element. Coada nu este vida (`count == 1`), si P2 trece de verificarea initiala `while(count == items.length)`.

7. P2 adauga un element in coada si incrementeaza `tail`.

8. P2 observa ca `tail` a atins capatul circular al cozi (`tail == items.length`) si semnalizeaza `empty` (`if (count == 1) { empty.signal(); }`).

9. Firul de executie C2 (Consumator 2) incearca sa extraga un element. Coada este vida (`count == 0`), si C2 trece de verificarea initiala `while (count == 0)`.

10. C2 asteapta semnalul de la producatori (`try { empty.await(); }`), deoarece P1 a semnalat anterior ca coada nu mai este vida, dar C1 inca nu a terminat de extras elementul.

11. P1 asteapta semnalul de la consumatori (`try { full.await(); }`), deoarece P2 a semnalat anterior ca a adaugat un element si a eliberat un loc in coada.

12. Blocare: P1 asteapta sa elibereze `full`, dar P2 a semnalat `empty`. In acelasi timp, C2 asteapta sa elibereze `empty`, dar C1 nu a semnalat inca.

Rezultatul este un deadlock, deoarece P1 si P2 asteapta semnale de la consumatori, iar C1 si C2 asteapta semnale de la producatori, iar niciunul dintre aceste semnale nu va fi emis, deoarece fiecare producator asteapta semnalul celuilalt si fiecare consumator asteapta semnalul celuilalt.

Pentru a evita aceasta problema, semnalele ar trebui sa fie emise independent de operatiile specifice (cum ar fi verificarea `count == 1` sau `tail == items.length`), si ar trebui sa fie emise intotdeauna atunci cand o conditie este indeplinita, indiferent de operatia care a condus la indeplinirea conditiei.

4.

a) Problema este ca, atunci cand mai multi producatori sau mai multi consumatori incearca sa acceseze coada simultan, acestia se blocheaza reciproc, reducand astfel eficienta si avantajul de a avea mai multe fire de executie.

Cu un singur lock pentru intreaga coada (ReentrantLock), fiecare producator sau consumator trebuie sa obtina si sa elibereze acest lacat inainte si dupa ce acceseaza coada. Acest lucru face ca operatiile de enq si deq sa fie blocante pentru celelalte thread-uri, chiar daca acestea ar putea fi in alt capat al cozii si nu ar afecta operatia curenta.

b) Aceasta varianta de generalizare este corecta, avand in vedere:

-Evitarea Deadlock-ului: Prin folosirea a doua lacate separate, se evita potentialul deadlock intalnit in varianta anterioara cu un singur lacat. Deoarece enq si deq au lacatele lor distincte, un thread care a obtinut un lacat nu va bloca operatiile celuilalt thread.

-Concurenta intre Operatii: Operatiile enq si deq pot fi realizate concurent de catre diferite thread-uri. Daca un thread realizeaza o operatie de enq, nu va bloca un alt thread care incearca sa realizeze o operatie de deq si invers.

5.

a) Demonstratie pentru excluderea mutuala:

Invarianta la intrare: Inainte de a intra in bucla, thread-ul seteaza `flag[i]` la `true`, semnalizand dorinta de a intra in sectiunea critica.

Invarianta in bucla: Bucla do-while asigura ca thread-ul poate intra in sectiunea critica doar daca:

- Niciun alt thread nu doreste sa intre (`flag[j] == false`), indicand ca sectiunea critica este disponibila.

- Sau daca thread-ul curent are o eticheta mai mare (`label[j] > label[i]`), indicand ca este prioritizat si are permisiunea de a intra.

Invarianta la iesire: Dupa ce iese din bucla, `access[i]` este setat la `true`, indicand ca thread-ul are acces la sectiunea critica.

Aceste invarianțe asigura ca un thread poate intra in sectiunea critica numai daca niciun alt thread nu a intrat in aceeasi perioada si ca ordinea de acces este determinata de etichetele thread-urilor. Astfel, algoritmul propus asigura excluderea mutuala, deoarece un singur thread are acces la sectiunea critica intr-un moment dat, iar accesul este gestionat in functie de etichetele si starea flag-urilor thread-urilor.

b) Demonstratie pentru deadlock-free:

- Thread-ul va astepta in bucla pana cand toate celelalte thread-uri fie nu doresc sa intre in sectiunea critica, fie au o eticheta mai mare.

- Cel putin un thread va obtine acces la sectiunea critica, deoarece bucla se va incheia atunci cand thread-ul curent are cea mai mica eticheta sau nu exista alti thread-uri care doresc sa intre.

Demonstratie pentru starvation-free:

- Fiecare thread care solicita acces la sectiunea critica va obtine in cele din urma accesul.

- Etichetele sunt atribuite intr-o maniera ordonata si justa, iar thread-ul cu eticheta cea mai mica va avea prioritate. Astfel, niciun thread nu va fi blocat permanent.

Algoritmul propus asigura atat lipsa blocarii (deadlock-free), cat si asigura ca fiecare thread va obtine accesul la sectiunea critica la un moment dat (starvation-free). Etichetele si utilizarea lacatelor contribuie la aceste garantii, asigurand ordinea si corectitudinea procesului de acces la resursa comuna.

c) Implementarea modificata pentru a evita cresterea nelimitata a valorilor din label:

Adaugam un prag (THRESHOLD) in functia unlock(i) pentru a verifica daca label[i] a depasit pragul.

Daca label[i] depaseste pragul, resetam toate valorile din matricea label la valorile initiale.

Astfel, evitam cresterea nelimitata a valorilor din label.

```
function unlock(i) {
   label[i] = max(label[0], ..., label[n-1]) + 1;
   if (label[i] > THRESHOLD) {
      for (int k = 0; k < n; k++) {
         label[k] = k + 1;
      }
   }
   access[i] = false;
   flag[i] = false;
}
```

Dezavantajul: Adaugarea acestei verificari poate afecta complexitatea timpului de executie, deoarece resetarea matricei label implica parcurgerea intregii matrice.

Observatie: Pragul (THRESHOLD) poate fi setat la numarul total de thread-uri, daca dorim sa resetam label dupa ce fiecare thread a trecut prin sectiunea critica o data.