Tehnici Programare Multiprocesor
-Tema 2-

Enia Vlad Ieftimie
Găină Robert-Adrian
Viziteu David-Andrei

1.

a). Punctul de linearizare specific unei **adăugări cu succes in listă** este atunci când efectul operației de adăugare este vizibil în lista inițială (i.e. lista inițială este alterată), mai exact atunci când în campul *next* al predecesorului este pus nodul cel nou (i.e. linia 64: pred.next = node)

b). Punctul de linearizare specific unei **adăugări eșuate în listă** este atunci când se face lock asupra listei, întrucât ea rămâne neschimbată în urma unei astfel de operație.

c). Punctul de linearizare specific unei **ștergeri cu succes din listă** este atunci când efectul operației de ștergere este vizibil în lista inițială (i.e. lista inițială este alterată), mai exact atunci când în campul *next* al predecesorului este pus nodul succesor al nodului eliminat (i.e. atunci când practic "se sare" peste nodul ce trebuie liminat din lista - linia 91: pred.next = current.next).

d). Punctul de linearizare specific unei **ștergeri eșuate din listă** este atunci când se face lock aspura listei, întrucât ea rămâne neschimbată în urma unei astfel de operație.

2.

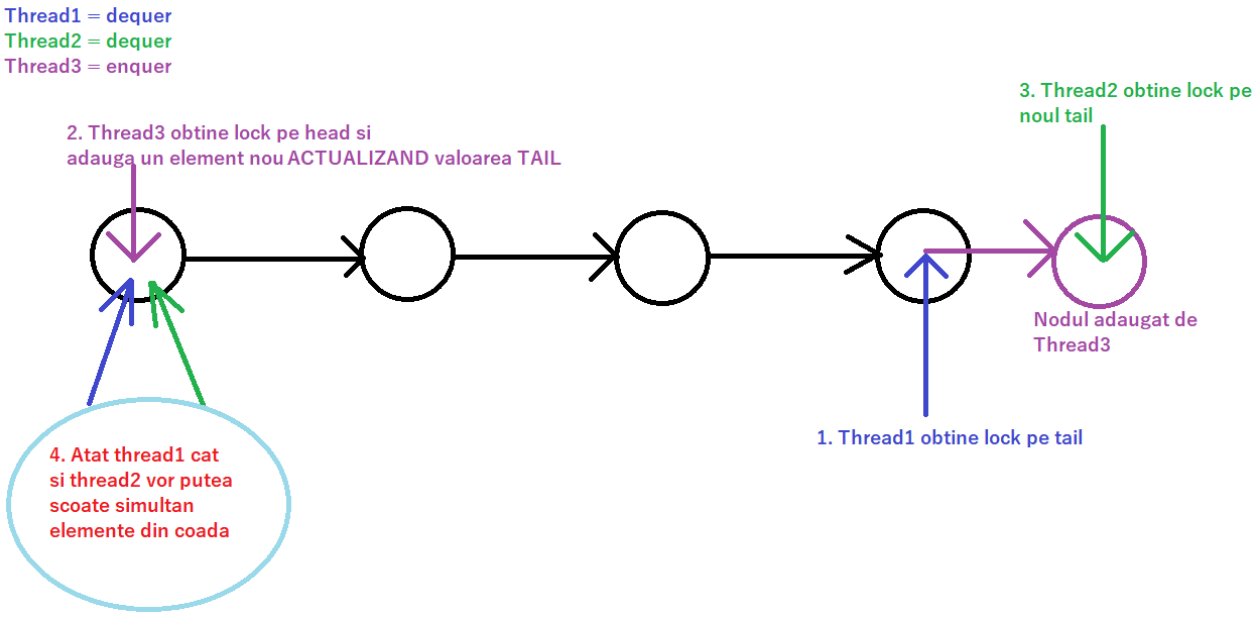
a). Pentru metoda *deq()* variabila size este folosită doar in situația în care coada este plină. Un DEQUER consumă un element și anunță ENQUERII că pot adăuga elemente in coadă.

Dacă am scoate if-ul cu funcția *size.getAndDecrement()* în afara secțiunii protejate de *deqLock*, un alt DEQUER ar putea consuma un element înainte ca atributul *size* să fie actualizat. Acest lucru, însă, **nu ne afectează funcționalitatea**, întrucât condiția folosită pentru a verifica existența elementelor in coadă este ***head.next == null***, nu *size.get() == 0*.

Întrucât operațiile sunt protejate de *deqLock* sau sunt atomice, avem garanția că cel puțin un thread DEQUER va vedea faptul că *size.getAndDecrement()==capacity*. Dacă, de exemplu, am avea toți ENQUERII blocați deoarece atributul *size.get()==capacity* și N thread-uri care vor consuma toate elementele dintr-o coadă de dimensiune N, fără să apuce sa actualizeze atributul *size* (și, implicit, să trezească ENQUERII), atunci cel de-al N+1-lea DEQUER va vedea coada goală (*head.next==null,* chiar dacă *size == capacity*) și va face release la lock. Unul dintre cei N DEQUERI anteriori va primi accesul la lock și va putea actualiza atributul *size,* notificând astfel ceilalți ENQUERI.

În concluzie, **putem scoate metoda *size.getAndDecrement()* în afara sectiunii protejate** de *deqLock,* cât timp numărul de DEQUERI este finit.

b). Să presupunem că avem de a face cu o coadă, pe care operează mai mulți DEQUERI decât ENQUERI. Atunci putem ajunge în situația descrisă mai jos:



Dacă nu ajung Thread1 și Thread2 să efectueze în același timp *head = head.next*, atunci ***deq()* va funcționa corect, fără a se afecta caracterul FIFO** al cozii, deoarece, oricare dintre cele două thread-uri face primul eliminarea, va face totodată și actualizarea *head*-ului, deci celălalt va elimina următorul element firesc din coadă. **Dar operația se poate face simultan => functia deq() nu va funcționa corect.**

c). În cazul în care avem o simplă operație de *spinning*, **algoritmul va funcționa corect**, întrucât tot codul funcției se află in blocul protejat de *enqLock,* respectiv *deqLock* și astfel nu mai suntem nevoiți să notificăm ENQUERII, respectiv DEQUERII.

Deoarece tot codul unei funcții este protejat de lock, nu vom rămâne cu cod neexecutat de către un alt thread după ce face release la lock, deci avem garanția că dimensiunea și elementele cozii vor fi modificate corect.

De asemenea, chiar dacă scade performanța cozii, totuși nu vom mai avea situații în care notificările se realizează cu întârziere.

Pentru a doua întrebare, există o situație care rezultă un **deadlock**:

Presupunem că avem 1 ENQUER *Enquer* și 2 DEQUERI *Dequer1* și *Dequer2.*

1. *Dequer1* așteaptă în zona *if(head.next == null),* iar *Dequer2* așteaptă în zona *while(size.get()==0)*; *Dequer2* are lock-ul;

2. *Enquer*-ul adaugă un element **(1)** in coadă**, însă nu trimite încă semnalul către dequeri.**

3. *Dequer2* consumă elementul adaugăt și se întoarce in *if(head.next==null);*
Dequer1 și *Dequer2* așteaptă acum în *if(head.next==null)*;
Dequer2 cedează lock-ul, iar *Enquer*-ul îl va prelua;

4. *Enquer*-ul trimite semnalul **(1)** către dequeri;

5. *Dequer2* iese din blocul *if* și se blochează in *while(size.get()==0)*, **având lock-ul;**

6. *Enquer*-ul adaugă un element în coadă, **însă nu trimite încă semnalul către dequeri;**

7. *Dequer2* consumă elementul adăugat și se intoarce în *if(head.next==null),* **cedând lock-ul;**

8. *Dequer1* iese din await (deoarece a primit semnalul **(1)** de la *Enquer*) și rămane blocat în while **cu lock-ul luat**;

9. Acum când *Enquer*-ul va încerca să preia lock-ul pentru a da signal către DEQUERI, nu vaputea, întrucat lock-ul este la *Dequer1*. => **DEADLOCK**

În concluzie, se poate ajunge la o situație de deadlock în care un DEQUER rămane cu lock-ul in bucla *while*, alt dequer așteaptă semnal, fiind blocat in instrucțiunea *if*, coada este goală iar ENQUER-ul nu poate umple coada pentru că așteaptă lock-ul pentru a trimite notificarea catre DEQUERI.

d). Să presupunem că avem o coadă a cărei metoda *deq()* nu are verificarea de coadă nevidă într-o secțiune protejată de lock. Vom analiza cazul în care, la un moment dat, avem un singur element în coadă, iar asupra ei au acces 2 thread-uri.

Dacă cele două apelează metoda *deq()* în același timp și ajung ambele în același timp la condiția *if(head.next == null)* (i.e. ambele verifică în același timp dacă coada e goală), atunci ambele vor trece mai departe de condiția *if*, fiindcă în coadă încă este un element întrucât niciun thread nu a ajuns încă în punctul de linearizare (i.e. linia 28:*head = head.next*).

Așadar, se va ajunge în situația în care se vor face mai multe operații *deq()* decât sunt elemente în coadă.

3.
Am implementat versionarea folosind o variabilă atomic long pentru un obiect de tip nod astfel:

- Variabila este inițializată cu 0
- O valoare pară indică faptul că nimeni nu scrie în prezent.
- O valoare impară indică faptul că un cititor este în proces de actualizare a datelor.

Protocol pentru cititor (ideea generală):

1) Se citește și se stochează numărul versiunii.
2) Se citesc datele.
3) Se citește din nou numărul versiunii și se compară cu valoarea citită la pasul 1). Aici avem mai multe cazuri posibile:
   i) Valorile citite la pașii 1 si 3 sunt egale si pare. În acest caz, știm că datele citite sunt consistente.
   ii) Valoarea citită la pasul 1 este impară. În acest caz, este irelevantă valoarea citită la pasul 3, deoarece un scriitor este/a fost in procesul de actualizare a resursei, în timp ce cititorul a accesat resursa.
   iii) Valoarea citită la pasul 3 este impara. În acest caz este irelevantă valoarea citită la pasul 1, deoarece un scritor a început procesul de actualizare a resursei, în timp ce cititorul a accesat resursa.

Protocol pentru scriitor (ideea generală):

1) Se citește și se stochează numărul versiunii. (Dacă acesta este impar, înseamnă că un alt scriitor este deja în proces de actualizare a datelor, așa că se repetă acest pas până se citește un număr par.)
2) Se incrementează numărul versiunii al resursei și se accesează resursa
3) Se incrementează numarul versiunii al resursei (acesta devine din nou par)

Această abordare are sens (este eficientă) doar când cea mai frecventă operație este cea de citire, deoarece cititorii trebuie să recitească o resursa dacă aceasta a fost modificată de un scriitor, pentru a fi siguri că datele citite sunt consistente. Dacă operația de scriere este predominantă, numarul de operații de recitire efectuate de cititori va crește, ducând astfel la timpi de execuție mai mari.

Descriere implementare:

- Particularități pentru **funcția validate**: primește 2 parametri suplimentari: versiunea inițială a nodului pred și versiunea inițiala a nodului curent. Dacă una din valorile inițiale este impară, returnează false. Dacă versiunile inițiale (cele primite ca parametru) ale nodurilor pred si curent sunt egale cu versiunile curente ale acestora și dacă versiunea lui pred.next este egală cu verisunea lui curent, funcția returnează true.

- Particularități pentru **funcția add**: salvează versiunea inițială ale nodurilor pred și next înainte de a aplica lacătul pe ele. Dacă funcția de validare returnează true, se

incrementează versiunile, se adaugă noul nod, iar la sfârșit se incrementează din nou versiunile nodurilor pred și next și se deblochează lacătele aplicate.

- Particularități pentru **funcția remove**: salvează versiunea inițială ale nodurilor pred și next înainte de a aplica lacătul pe ele. Dacă funcția de validare returnează true, se incrementează versiunile, se scoate din listă nodul curent, iar la sfârsit se incrementează din nou versiunile nodurilor pred și next și se deblochează lacatele aplicate.

- Particularități pentru **funcția contains**: la fel ca inainte, doar că se salvează versiunile nodurilor pred si current înainte de a aplica lacătul pe ele, aceste valori fiind folosite ca parametri suplimentari pentru noua funcție validate.

Comparație (5 rulari):

Funcție	Nod versionat	Nod neversionat (varianta originala)
add	102.188 ns	222.160 ns
remove	3.055 ns	5.369 ns
validate	**128 ns**	**204.567 ns**
Contains	93.997 ns	180.635 ns

După cum se poate observa în tabel, **introducerea unei versionari reduce timpul operațiilor aproape la jumătate fața de cel original** în acest context. Cea mai mare diferența se poate observa la funcția validate, care este de 100 de ori mai rapida.

Specificatii PC: Ryzen 9 5900HS, 8 cores, 16 threads, 16gb RAM, ssd.