

# Structuri de date concurente

---

CURS NR. 7

# Structuri de date concurente

---

## Context și motivație

- Răspândirea sistemelor multiprocesor cu memorie partajată (ex. sisteme multi-core)
  - Execută multiple threaduri care comunică și se sincronizează prin intermediul structurilor de date aflate în memoria partajată

## Provocări

- Proiectarea (și testarea) structurilor de date concurente este **dificilă** (în comparație cu structurile secvențiale)
  - Problema este dată de **concurență**
    - execuția threadurilor care accesează structurile de date în citire/scriere pot fi **intercalate** în moduri variate și cu urmări diferite și posibil neașteptate
      - Datorită planificatorului, a page-fault-urilor, a întreruperilor, etc.
      - → operațiile threadurilor distincte trebuie considerate ca fiind complet asincrone
  - Necesitatea de a proiecta structuri de date **scalabile** și **corecte** (obiective care adesea sunt greu de maximizat împreună)
    - Care pot beneficia de îmbunătățirile care apar în sistemele care permit execuția paralelă a tot mai multor threaduri concurente
- Structura de date concurentă necesită modalitate aparte de stocare și gestiune a structurii de date pentru a permite acces concurent din partea mai multor threaduri, oferind în același timp performanță și corectitudine
  - Acces în citire și în scriere/modificare

# Exemplu: Incrementarea unui contor partajat

- Contorizarea este una din **acitivitățile de bază** efectuate de calculator
- În context concurent, apare **difficultatea proiectării unui algoritm de contorizare concurent** care să se scaleze bine în sisteme multicore/multiprocesor cu memorie partajată

- Problema**

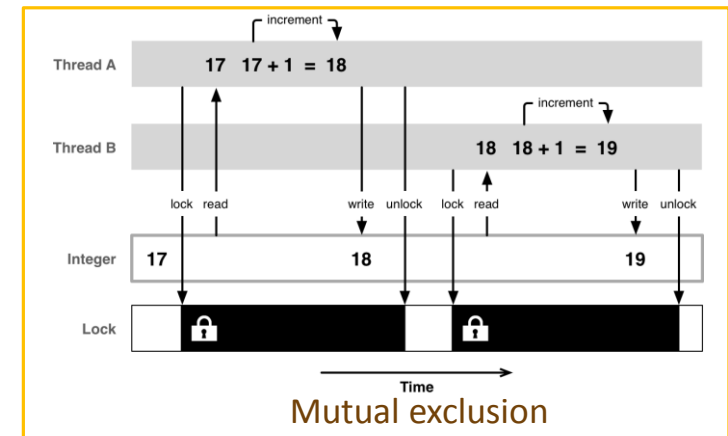
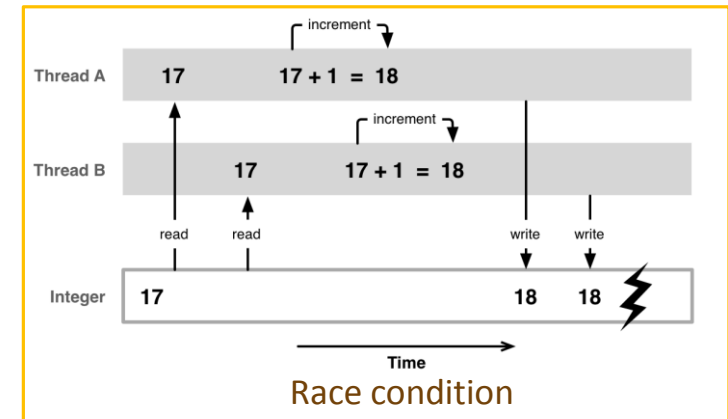
- Operația de incrementare **nu este atomică** (load, inc, store)
  - Execuția threadurilor se poate **intercala** aî. se ajunge la rezultate inconsistente

- Soluția clasică: **excludere mutuală** folosind **lacăte**

```
oldval = X;
X = oldval + 1;
return oldval;

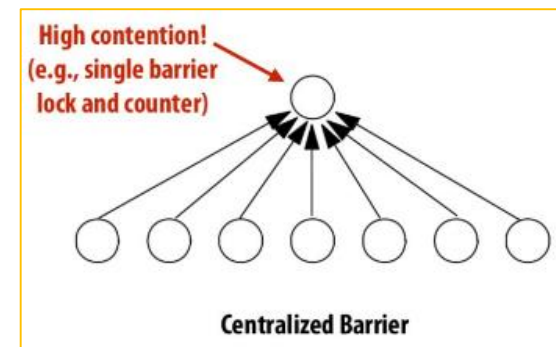
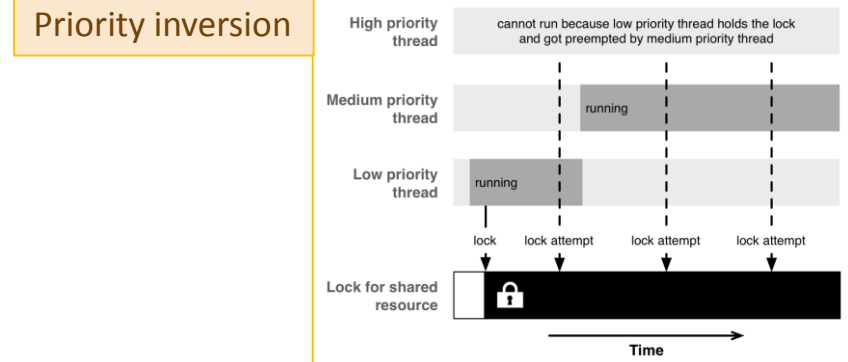
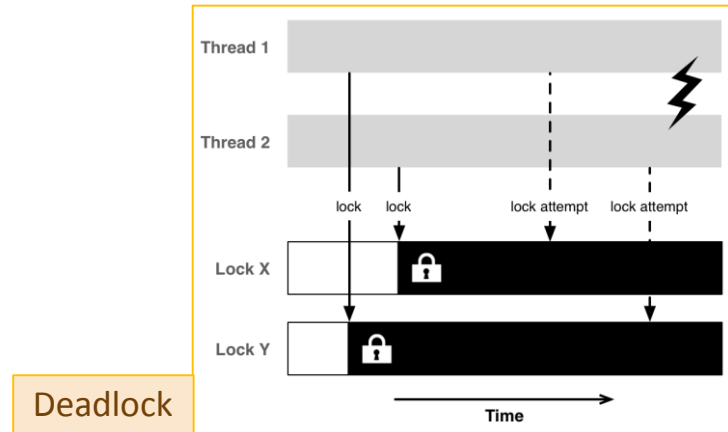
acquire(Lock);
oldval = X;
X = oldval + 1;
release(Lock);
return oldval;
```

fetch-and-inc secvențial (stg.) și bazat pe lacăte (dr)



# Exemplu: Incrementarea unui contor partajat

- Probleme introduse de **excluderea mutuală** folosind **lacăte**
  - Se evită intercalările rele prin excluderea tuturor intercalărilor
  - Prin blocare se introduc o serie de probleme legate de performanță și ingineria software
    - Deadlock** (interblocare)
    - Înfometare**
    - Inversarea priorităților**
    - Întârzierea** tuturor threadurilor care necesită acces la contor
    - Lock contention**
      - Mai multe threaduri se întrec în accesul la lacăt pentru blocarea acestuia  
→ toate threadurile vor acces la aceeași resursă partajată: lacătul
    - Impact negativ** asupra paralelismului și **scalabilității**
      - În regiunea critică activitățile sunt secvențiale – se pierde avantajul concurenței



# Concepte

- **Speedup**

- Raportul dintre timpul de execuție pe 1 procesor și timpul de execuție pe P procesoare
- Măsura eficienței în utilizarea sistemului pe care rulează aplicația (Ideal – am dori speedup liniar)

- **Scalabilitate**: Structurile de date a căror speedup crește cu P sunt scalabile

- **Gâtuire secvențială** – poate reduce speedup-ul în mod drastic

- Excluderea mutuală introduce gâtuire
- Efectul secvențializării asupra speedup-ului
  - Considerând că
    - Programul necesită **1** unitate de timp pentru execuție pe 1 procesor
    - Partea secvențială pe **1** și pe **P** procesoare necesită **b** unități de timp
    - Partea concurentă pe P procesoare necesită **(1 - b)/P** unități de timp în cazul cel mai bun
- Observație: dacă 10% din cod este secvențial, speedup-ul pe o mașină cu P = 10 este limitat la aprox 5.3
  - Aplicația rulează la aproximativ jumătate din capacitatea sistemului



Speedup-ul este:

$$1 / ( b + (1 - b)/P )$$

- Concluzie

- Trebuie redusă lungimea secțiunii de cod secvențial → reducerea numărului de lacăte, și reducerea numărului de instrucțiuni din regiunea critică (granularitate blocării trebuie redusă)

# Concepte (2)

---

- **Memory contention**

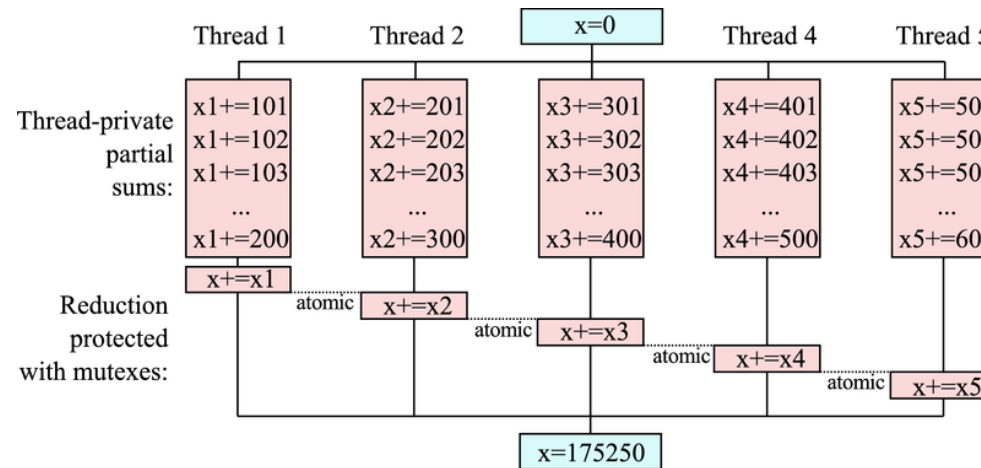
- Încărcare introdusă de accesul simultan din partea mai multor threaduri la aceeași locație de memorie partajată
  - Ex. Dacă lacătul este într-o zonă de memorie partajată, pentru a obține lacătul, toate threadurile încearcă să modifice în mod repetat valoarea acestuia (să o blocheze pentru a intra în regiunea critică)
  - Așteptare datorată asigurării coerenței cache-urilor, etc.

- **Blocare**

- Dacă threadul care deține lacătul este reținut în execuție (planificatorul alege alte threaduri spre execuție), atunci toate threadurile care așteaptă eliberarea lacătului sunt “blocate” (întârziate)
- Soluție: utilizarea unor algoritmi neblocați – dacă un thread este întârziat, să nu provoace întârzierea (blocarea) altor threaduri
  - Acești algoritmi nu pot utiliza lacăte – ca mecanisme de sincronizare (se numesc lock-less, sau lock-free)

# Exemplu: Incrementarea unui contor partajat

- Alte soluții – **permit evitarea sincronizării** (în situații speciale)
  - Utilizarea structurilor locale threadurilor concurente**
    - În locul accesării unei resurse partajate – threadurile folosesc date stocate local
      - Fiecare thread lucrează cu o copie locală a resursei – vede doar această copie și nu știe că alte threaduri concurente pot folosi simultan copiile lor locale
    - Când se poate folosi? - Doar când ordinea reală în care fiecare thread accesează resursa nu este importantă!



Însumare folosind containere private  
 $x = 101 + 102 + \dots + 600$

- Distribuirea în timp a accesărilor (Back-off)**
  - Când se poate folosi? – Când ordinea accesării este importantă dar, accesările pot fi distribuite în timp - unele accesări pot fi amânate în siguranță → Actualizările se fac în intervale de timp distincte

# Structuri de date concurente

---

Cum se construiesc structurile concurente? Ce tehnici de blocare sunt utilizate?

- Cu **granularitate mare** în blocarea accesului la regiunile critice
  - Toată regiunea unde se accesează data partajată (întreaga structură) este protejată de un singur lacăt
- Cu **granularitate fină** în blocarea accesului la regiunile critice mai restrânse
  - Folosesc lacăte de granularitate fină pentru protejarea diferitelor părți (zone) ale structurii de date
    - Operațiile necesită blocarea unuia sau a mai multor lacăte pentru a putea accesa (citi, scrie) data partajată
  - Poate furniza performanțe mult mai bune decât varianta precedentă, însă persistă problemele specifice utilizării lacătelor
    - Pot apare probleme de interblocare, înfometare, preempție, inversarea priorităților, etc.
    - Necesită o strategie adecvată pentru eliberarea resurselor partajate în cazul terminării neprevăzute a threadului
- **Fără blocare** (fără blocarea altor threaduri) în accesul la regiunile critice
  - Granularitatea cea mai fină ce poate fi obținută în accesul sincronizat la resurse partajate → fără lacăte clasice
  - Folosește operații atomice (test-and-set, compare-and-swap, etc.) pentru a menține datele partajate în stări consistente
  - Poate furniza performanțe foarte bune, însă
  - Proiectarea, implementarea și verificarea corectitudinii poate fi foarte dificilă
- Folosind **memorie tranzacțională**
  - Mai multe operații sunt grupate împreună formând o tranzacție care se efectuează în mod atomic



# Tehnici blocante cu granularitate fină

## Context:

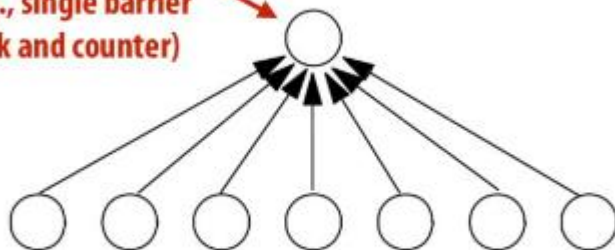
- Soluția cu granularitate mare - cu un singur lacăt care protejează toată structura - reduce semnificativ performanța
- Soluția cu containere private threadurilor este scalabilă la operații de update, dar nu și la operații frecvente de citire
  - Citirea presupune calcularea valorii globale iterând prin valorile parțiale ( $O(n)$ )

**Scop:** scalabilitate - efectuarea paralelă a operațiilor concurente care nu accesează aceeași parte a structurii

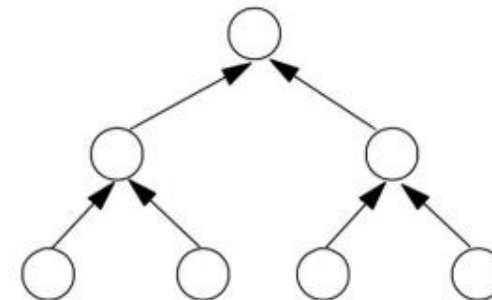
**Principiu:** Mai multe lacăte de granularitate fină protezează diferitele părți ale structurii de date

- Pentru unele structuri abordarea aceasta este naturală (ex. Lista, Tabela de hashing – cu câte un lacăt per bucket)
- Pentru altele se folosesc tehnici speciale
  - Ex. Arbore de combinări (combining tree) pentru incrementarea scalabilă a unui contor partajat

High contention!  
(e.g., single barrier  
lock and counter)



Centralized Barrier



Combining Tree Barrier

# Tehnici blocante cu granularitate fină

## Arbore de combinări (combining tree) pentru incrementarea scalabilă a unui contor partajat

- Arborele de combinări este un **arbore binar complet**, în care
  - Fiecărui thread îi corespunde o frunză
    - Fiecare nod poate avea maxim 2 threaduri asignate
  - În rădăcină este valoarea reală a contorului
  - Nodurile interne se folosesc pentru coordonarea accesului la rădăcină
- **Principiu**
  - Threadurile urmăresc să escaladeze arborele de la frunze către rădăcină, și pe drumul lor se "combină" cu alte operații concurente
  - Dacă două threaduri ajung aproximativ în același timp la un nod, se realizează "combinarea" operațiilor
    - Primul thread este threadul ACTIV (winner)
      - Își combină cererea proprie de incrementare cu cel al threadului PASIV și duce cererea combinată la următorul nivel către rădăcină
    - Al doilea thread este threadul PASIV (loser)
      - Așteaptă la acest nod ca threadul ACTIV să revină cu rezultate
      - Un thread activ de la un nod poate deveni thread pasiv la nivelul următor
  - Threadul activ care ajunge la rădăcină va incrementa valoarea contorului și va trimite valoarea veche în jos în arbore (distribuire)

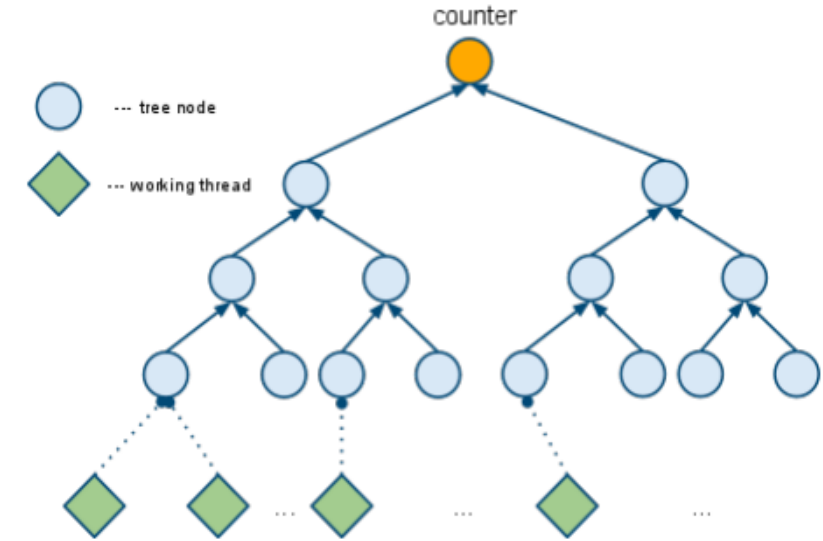


Figure 3-1: a combining tree with 8 leaf nodes

# Tehnici blocante cu granularitate fină

## Arbore de combinări (combining tree) pentru incrementarea scalabilă a unui contor partajat

- 4 faze

- **Pre-combinare**

- Threadul pornește de la o frunză și avansează în sus până când ajunge la rădăcină sau devine pasiv
    - Dacă se oprește la un nod intern → blochează nodul: indică faptul că va reveni cu rezultate combinate

- **Combinare**

- Threadul activ avansează mai sus cu cererea combinată (și cererea threadului pasiv)
    - Dacă nodul este blocat, threadul trebuie să aștepte deblocarea pentru a putea efectua combinarea și apoi poate avansa în sus

- **Operație**

- Threadul pasiv își pune cererea proprie (acumulată pe calea pe care a avansat) în nod și deblochează nodul – așteptând aici revenirea threadului activ cu rezultate în urma incrementării conotorului din rădăcină

- **Distribuire**

- Threadul activ (după ce efectuează incrementarea contorului) va coborâ în arbore (până ajunge la noduri frunză), către nodurile unde threadul pasiv așteaptă distribuirea rezultatelor

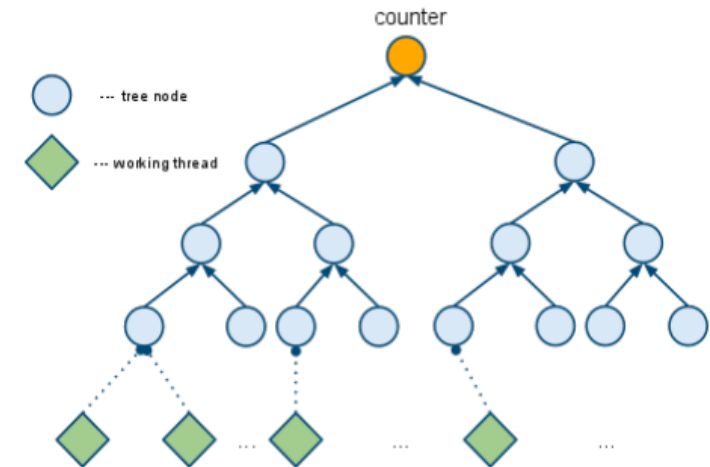


Figure 3-1: a combining tree with 8 leaf nodes

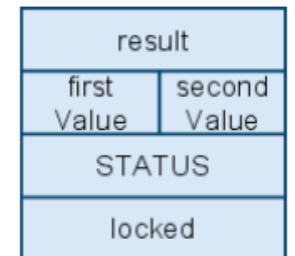


Figure 3-2: structure of a tree node

# Tehnici blocante cu granularitate fină

---

## Arbore de combinări (combining tree) pentru incrementarea scalabilă a unui contor partajat

- **Avantaje**

- Reduce competiția pentru accesul la aceeași locație de memorie
- Reduce traficul prin spinning local
- Performanța se scalează cu numărul de threaduri concurente – mult mai bine decât varianta cu un singur lacăt
- Considerăm un arbore de lățimea  $P$  – unde threadurile reușesc în mod repetat să facă combinare
  - Permite ca  $P$  threaduri să returneze  $P$  valori în timp  $O(\log P)$  → speedup:  $O(P / \log P)$

- **Dezavantaje**

- Trebuie să cunoaștem limitele pentru numărul de threaduri care accesează contorul
- Implică cost în spațiu de stocare –  $O(P)$
- Deși oferă debit mai bun dacă încărcarea este mare, la număr mic de threaduri performanța este precară

- **Concluzie**

- Structurile de date blocante pot oferi o soluție de implementare eficientă dacă se ajunge la un echilibru între a folosi suficiente blocări pentru a avea corectitudine în timp ce se minimizează blocările pentru a permite executarea operațiilor concurente în paralel

# Tehnici neblocante

---

**Scop:** asigurarea că suspendarea nelimitată sau terminarea precoce a unui thread nu va întârzia alte threaduri în progresul lor (ca și grup) – aceștia pot progresa prin propriile operații neblocante

- elimină problemele create de utilizarea lacătelor

**Principiu:** accesarea în mod thread-safe a resurselor partajate, fără utilizarea primitivelor clasice de sincronizare (ex. Lacătele)

- Necesită suport din partea hardware-ului
- Condiții de progres – în ordinea descrescătoare a tăriei condiției
  - **Wait-freedom**
    - finalizarea este garantată după un număr finit de pași proprii (independent de temporizarea altor operații)
  - **Lock-freedom**
    - executarea unei operații (oarecare – poate fi al altui thread) este garantată după un număr finit de pași proprii
  - **Obstruction-freedom**
    - finalizarea este garantată după un număr finit de pași proprii, după ce nu mai apar interferențe cu alte operații
- Asigurările oferite de condițiile mai tari sunt atractive, dar implementările cu mai puține asigurări sunt de regulă mai simple, mai eficiente și mai ușor de proiectat și verificat
  - Compensarea pentru un progres mai redus se poate face cu tehnici precum backoff, și altele.

# Exemplu: Incrementarea lock-free a contorului partajat

- Problema: folosind doar operații load și store nu se poate furniza sincronizarea neblokantă a incrementării contorului partajat

- Load și store sunt operații distincte – nu formează o instrucțiune atomică!!!
- Intercalarea execuției threadurilor multiple poate duce la date inconsistente

**Instrucțiunile hardware atomice sunt universale:**

În sistemele care suportă aceste instrucțiuni, orice structură de date concurentă poate fi implementată lock-free

- Soluție: folosirea instrucțiunilor hardware atomice care grupează operațiile load și store

- Exemplu: Compare-and-swap

```
oldval = X;
X = oldval + 1;
return oldval;

acquire(Lock);
oldval = X;
X = oldval + 1;
release(Lock);
return oldval;
```

fetch-and-inc secvențial (stg.) și bazat pe lacăte (dr)



```
bool CAS(L, E, N) {
    atomically {
        if (*L == E) {
            *L = N;
            return true;
        } else
            return false;
    }
}
```

Semantica compare\_and\_swap

- Incrementare: fetch-and-inc → load + CAS
  - CAS poate eșua doar dacă valoarea contorului s-a schimbat între load și CAS – caz în care se poate trece la reîncercare
  - Abordarea este lock-free, dar nu și wait-free
- Unele dezavantaje ale abordării cu un singur lacăt persistă și aici
  - Gâtuirea secvențială și competiția pentru accesul la aceeași resursă
  - Soluțiile pentru eliminarea acestor probleme sunt mult mai dificil de integrat în abordări neblocante

# Listă simplu înlănțuită concurrentă

---

(ABORDĂRI ÎN IMPLEMENTAREA UNEI  
LISTE ORDONATE SIMPLU ÎNLĂNȚUITE CONCURENTE )

# Fără sincronizare

- Ce probleme pot apare dacă mai multe threaduri concurente efectuează operații pe listă?

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
};

void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }

    n->next = cur;
    prev->next = n;
}
```

```
void delete(List* list, int value) {

    // assume case of deleting first element is
    // handled here (to keep slide simple)

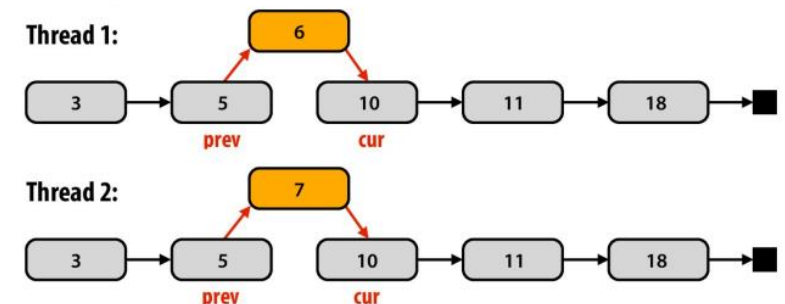
    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }

        prev = cur;
        cur = cur->next;
    }
}
```

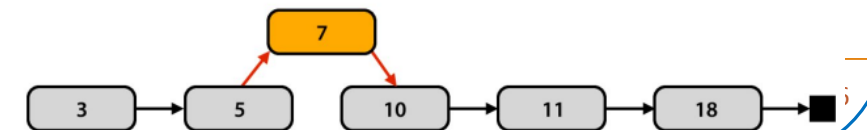
Inserare simultană

Thread 1 attempts to insert 6  
Thread 2 attempts to insert 7



Thread 1 and thread 2 both compute same prev and cur.  
Result: one of the insertions gets lost!

Result: (assuming thread 1 updates prev->next before thread 2)





# Blocare cu granularitate mare (1)

O soluție: **Sincronizare folosind un singur lacăt pentru protejarea întregii liste**

```
struct Node {  
    int value;  
    Node* next;  
};  
  
struct List {  
    Node* head;  
    Lock lock; ← Per-list lock  
};
```

Unde inserăm blocarea /  
deblocarea lacătului ?

```
void insert(List* list, int value) {  
  
    Node* n = new Node;  
    n->value = value;  
  
    // assume case of inserting before head of  
    // of list is handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value > value)  
            break;  
  
        prev = cur;  
        cur = cur->next;  
    }  
  
    n->next = cur;  
    prev->next = n;  
}  
  
void delete(List* list, int value) {  
  
    // assume case of deleting first element is  
    // handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value == value) {  
            prev->next = cur->next;  
            delete cur;  
            return;  
        }  
  
        prev = cur;  
        cur = cur->next;  
    }  
}
```

# Blocare cu granularitate mare (2)

O soluție: **Sincronizare folosind un singur lacăt pentru protejarea întregii liste**

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
    Lock lock; ← Per-list lock
};

void insert(List* list, int value) {
    Node* n = new Node;
    n->value = value;

    lock(list->lock);

    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }
    n->next = cur;
    prev->next = n;
    unlock(list->lock);
}

void delete(List* list, int value) {
    lock(list->lock);

    // assume case of deleting first element is
    // handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            unlock(list->lock);
            return;
        }

        prev = cur;
        cur = cur->next;
    }
    unlock(list->lock);
}
```

## Avantaj

- Simplitate în implementarea corectă a excluderii mutuale

## Dezavantaje

- Operațiile pe structura de date sunt serializate
- Poate reduce semnificativ performanța

# Blocare cu granularitate fină (1)

Căutăm o soluție mai bună: **Reducerea granularității la blocare**

- Nu trebuie să blocăm toată lista, doar "zona" în care lucrăm → threaduri concurente care accesează zone diferite pot lucra simultan

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
};

void insert(List* list, int value) {
    Node* n = new Node;
    n->value = value;

    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }

    prev->next = n;
    n->next = cur;
}

void delete(List* list, int value) {
    // assume case of deleting first element is
    // handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }

        prev = cur;
        cur = cur->next;
    }
}
```



## Blocarea unui singur nod

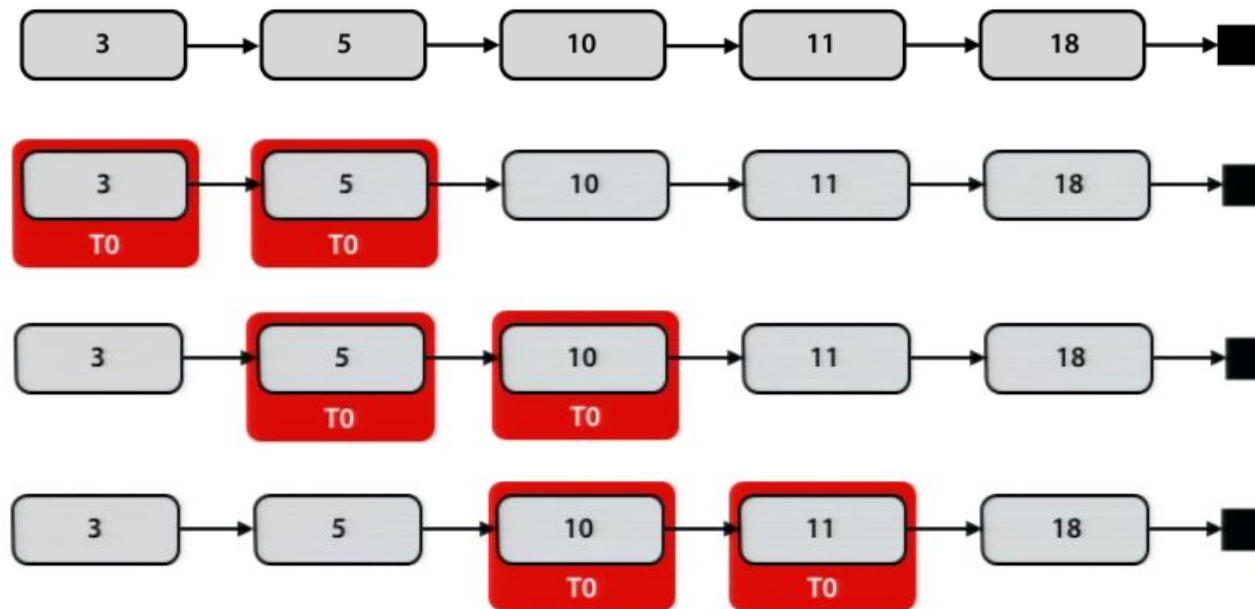
- Este suficientă? (inserare, ștergere)
- Ce probleme pot apare?

## Exemplu: ștergerea nodului 11

- Simultan cu inserarea unui nod după 10

# Blocare cu granularitate fină (2)

- **Hand-over-hand locking** (blocare mână-peste-mână)



Blocarea succesivă a două noduri consecutive

- Blochez primul
- Blochez al doilea
- Eliberez primul
- Blochez al treilea, etc.

Invariant

- pentru a modifica nodul M, trebuie să blocăm M și precedentul lui M

Exemplu: ștergerea nodului 11

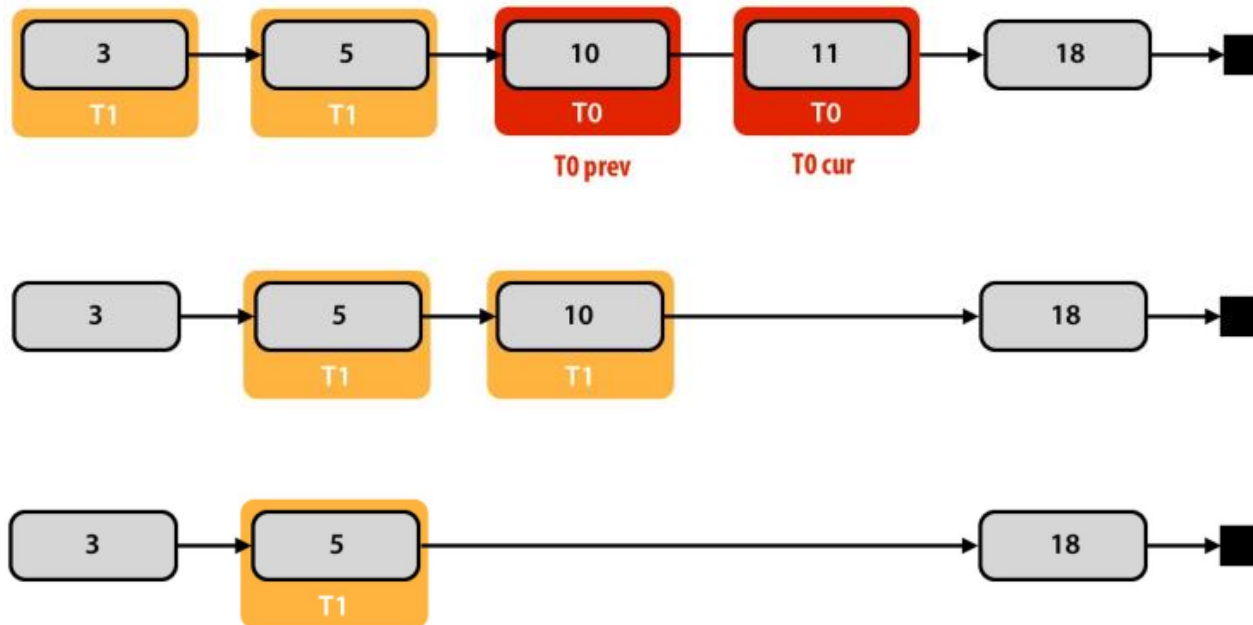
- Lacăt pe 10 și pe 11 (ambele se modifică)
- Între timp nu poate un alt thread să insereze după 10 (ar trebui să-l blocheze pe 10) sau să-l șteargă pe 18 (ar trebui să-l blocheze pe 11)

# Blocare cu granularitate fină (3)

- **Hand-over-hand locking** (blocare mână-peste-mână)

**Thread 0: delete(11)**

**Thread 1: delete(10)**



Exemplu: ștergere concurentă

- T1 trebuie să aștepte pt că T0 deține nodul 10
- T0 are toate resursele necesare pt ștergere – poate progresa și elibera nodul 10
- T1 are acces la nodul 10 și îl poate șterge – apoi deblochează nodul 5

# Blocare cu granularitate fină (4)

```
struct Node {
    int value;
    Node* next;
    Lock* lock;
};

struct List {
    Node* head;
    Lock* lock;
};

void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of insert before head handled
    // here (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    unlock(list->lock);
    if (cur) lock(cur->lock);

    while (cur) {
        if (cur->value > value)
            break;

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }

    n->next = cur;
    prev->next = n;

    unlock(prev->lock);
    if (cur) unlock(cur->lock);
}
```

```
void delete(List* list, int value) {

    // assume case of delete head handled here
    // (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    unlock(list->lock);
    if (cur) lock(cur->lock);

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            unlock(prev->lock);
            unlock(cur->lock);
            delete cur;
            return;
        }

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }

    unlock(prev->lock);
}
```

## Observații

- pentru a modifica nodul M, trebuie să blocăm nodul M
  - Pentru ca alt thread să nu-l șteargă în timp ce îl accesăm
- La ștergere, blocăm M și nodul precedent lui M
  - Pointerul next din nodul precedent se modifică
- La inserare nu trebuie blocat și nodul următor – se modifică doar pointerul la acel nod
  - E în regulă și dacă alt thread are nodul următor blocat
    - Pt. că nu îl poate șterge
    - Ar avea nevoie și de nodul pe care l-am blocat deja
- List-lock – protejează primul nod

## Exercițiu

- Încerați să găsiți o abordare mai puțin conservatoare pt insert !

# Blocare cu granularitate fină (5)

---

## Concluzii

- Avantaje
  - Permite efectuarea paralelă a operațiilor asupra structurii de date – în zone distincte
  - Reduce competiția pentru accesarea unui singur lacăt global
- Provocări
  - Asigurarea corectitudinii este mai dificilă
    - Trebuie identificate situațiile în care excluderea mutuală este necesară
- Costuri
  - Încărcare suplimentară la traversare - la fiecare pas trebuie să blocăm / deblocăm lacăte
    - Mai multe instrucțiuni
    - Traversarea implică scrieri în memorie (modificarea stării lacătului)
  - Crește dimensiunea nodului
    - Stocarea lacătului – în fiecare nod
- Pentru a reduce din costuri se pot aborda
  - Metode cu granularitate intermediară (zone cu un număr mai mare de noduri)
  - Lacăte Reader/Writer pentru a permite mai mulți cititori simultan

# Tehnici neblocante (lock-free)

---

## Să ne reamintim...

- Un **algoritm blocant** permite ca un thread să **împiedice un timp nelimitat** un alt thread în a-și efectua operațiile asupra detelor partajate
- Diferența dintre **busy-waiting** și **blocare**
  - Aici ambele sunt considerate abordări blocante
- Dacă folosim lacăt – implicit avem o abordare blocantă
  - Indiferent dacă se folosește spinning sau preempție
  - Doar un thread poate fi în regiunea respectivă de structură în care a intrat threadul
  - Chiar dacă granularitatea este fină ( și blocarea este la nivelul unui singur nod – ca în exemplul anterior)
- **Motivația**
  - La blocare poate interveni interblocarea, inversarea priorităților, scăderea performanței etc. și din motive care nu țin doar de implementare (eșuare, swapping, etc.)

## Algoritm / structură de date neblocantă

- Se garantează că un thread (cel puțin) face progres
  - Nu se poate preemta un thread la un moment inoportun și restul sistemului nu îl va împiedica în progres
- Atenție: nu se elimină înfometarea!



# Coadă concurentă neblocantă

---

(ABORDĂRI ÎN IMPLEMENTAREA UNEI  
COZI CONCURENTE LOCK-FREE)

# Tehnici nebloccante (lock-free)

Exemplu – Producător / Consumator cu **coadă circulară finită (vector de dimensiune fixă)**

```
struct Queue {  
    int data[N];  
    int head;    // head of queue  
    int tail;    // next free element  
};
```

```
void init(Queue* q) {  
    q->head = q->tail = 0;  
}
```

```
// return false if queue is full  
bool push(Queue* q, int value) {  
  
    // queue is full if tail is element before head  
    if (q->tail == MOD_N(q->head - 1))  
        return false;  
  
    q->data[q->tail] = value;  
    q->tail = MOD_N(q->tail + 1);  
    return true;  
}
```

```
// returns false if queue is empty  
bool pop(Queue* q, int* value) {  
  
    // if not empty  
    if (q->head != q->tail) {  
        *value = q->data[q->head];  
        q->head = MOD_N(q->head + 1);  
        return true;  
    }  
    return false;  
}
```

## Cu câte un singur producător și consumator !

Ce se întâmplă la push și pop concurent

- Un thread face push iar celălalt face pop concurent
- Când coada este goală pop eșuează
- Când coada este plină push eșuează
- Este thread-safe? (obs: 1 prod / 1 cons !!!)
  - DA
- Fără sincronizare și fără așteptare
  - Lock-free – abordare nebloccantă
  - Producătorul modifică doar tail-ul
  - Consumatorul modifică doar head-ul

# Tehnici nebloccante (lock-free)

Exemplu – Producător / Consumator cu **coadă circulară dinamică (vector dinamic)**

```
struct Node {
    Node* next;
    int value;
};

struct Queue {
    Node* head;
    Node* tail;
    Node* reclaim;
};

void init(Queue* q) {
    q->head = q->tail = q->reclaim = new Node;
}
```

```
void push(Queue* q, int value) {
    Node* n = new Node;
    n->next = NULL;
    n->value = value;

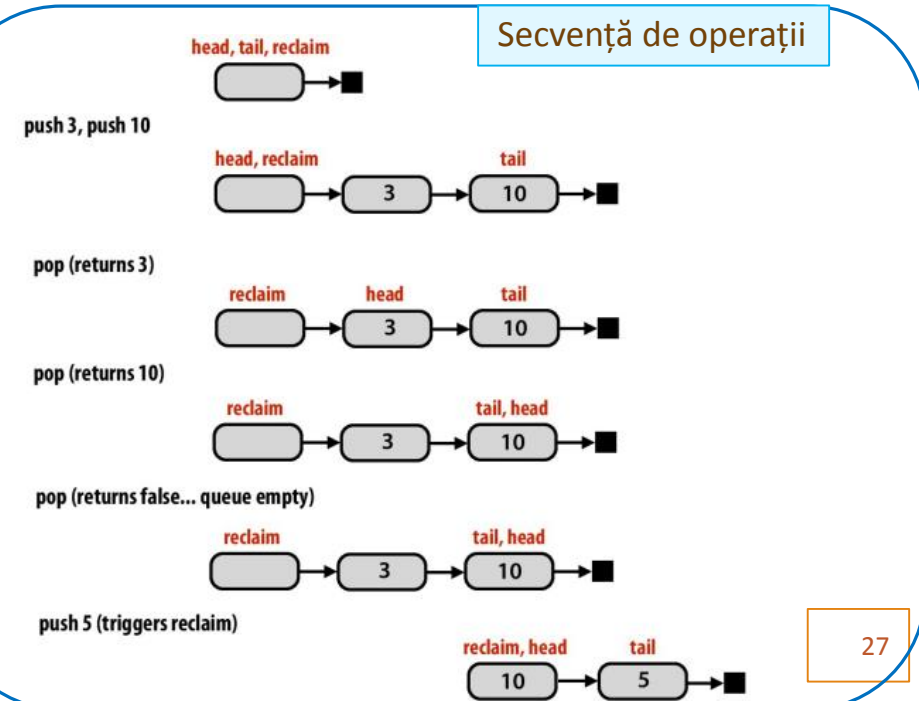
    q->tail->next = n;
    q->tail = q->tail->next;

    while (q->reclaim != q->head) {
        Node* tmp = q->reclaim;
        q->reclaim = q->reclaim->next;
        delete tmp;
    }
}
```

```
// returns false if queue is empty
bool pop(Queue* q, int* value) {
    if (q->head != q->tail) {
        *value = q->head->next->value;
        q->head = q->head->next;
        return true;
    }
    return false;
}
```

## Cu câte un singur producător și consumator !

- Alocarea și eliberarea memoriei se efectuează de un singur thread – Producătorul
- Tail indică ultimul element adăugat
- Head indică nodul dinaintea începutului de coadă
- Reclaim
  - Pointer la primul element care nu mai este în coadă dar care încă nu a fost șters
  - Asigură că pop nu va accesa un pointer nul



# Stivă concurentă neblocantă

---

(ABORDĂRI ÎN IMPLEMENTAREA UNEI  
STIVECONCURENTE LOCK-FREE)

# Tehnici neblocante (lock-free)

## Stivă neblocantă

- O primă abordare

```
struct Node {  
    Node* next;  
    int value;  
};  
  
struct Stack {  
    Node* top;  
};
```

```
void init(Stack* s) {  
    s->top = NULL;  
}
```

```
void push(Stack* s, Node* n) {  
    while (1) {  
        Node* old_top = s->top;  
        n->next = old_top;  
        if (compare_and_swap(&s->top, old_top, n) == old_top)  
            return;  
    }  
}
```

```
Node* pop(Stack* s) {  
    while (1) {  
        Node* old_top = s->top;  
        if (old_top == NULL)  
            return NULL;  
        Node* new_top = old_top->next;  
        if (compare_and_swap(&s->top, old_top, new_top) == old_top)  
            return old_top;  
    }  
}
```

## Stiva implementată cu listă simplu înlănțuită

Ne asigurăm că nici un alt thread nu a făcut pop la vârf, respectiv că nu s-a inserat un nou nod în vârful stivei

- Este vârful același cu ce a fost când am început procesul de adăugare?
  - Ca și pointer (adresă), nu ca și valoare
- Dacă DA, atunci înseamnă cu între timp nimeni nu a modificat stiva, deci operația este safe
- Analogic pentru pop
- Principiu
  - Dacă nici un alt thread nu a modificat stiva, threadul își poate efectua operația

## Comparați cu varianta în care folosim blocare cu granularitate fină

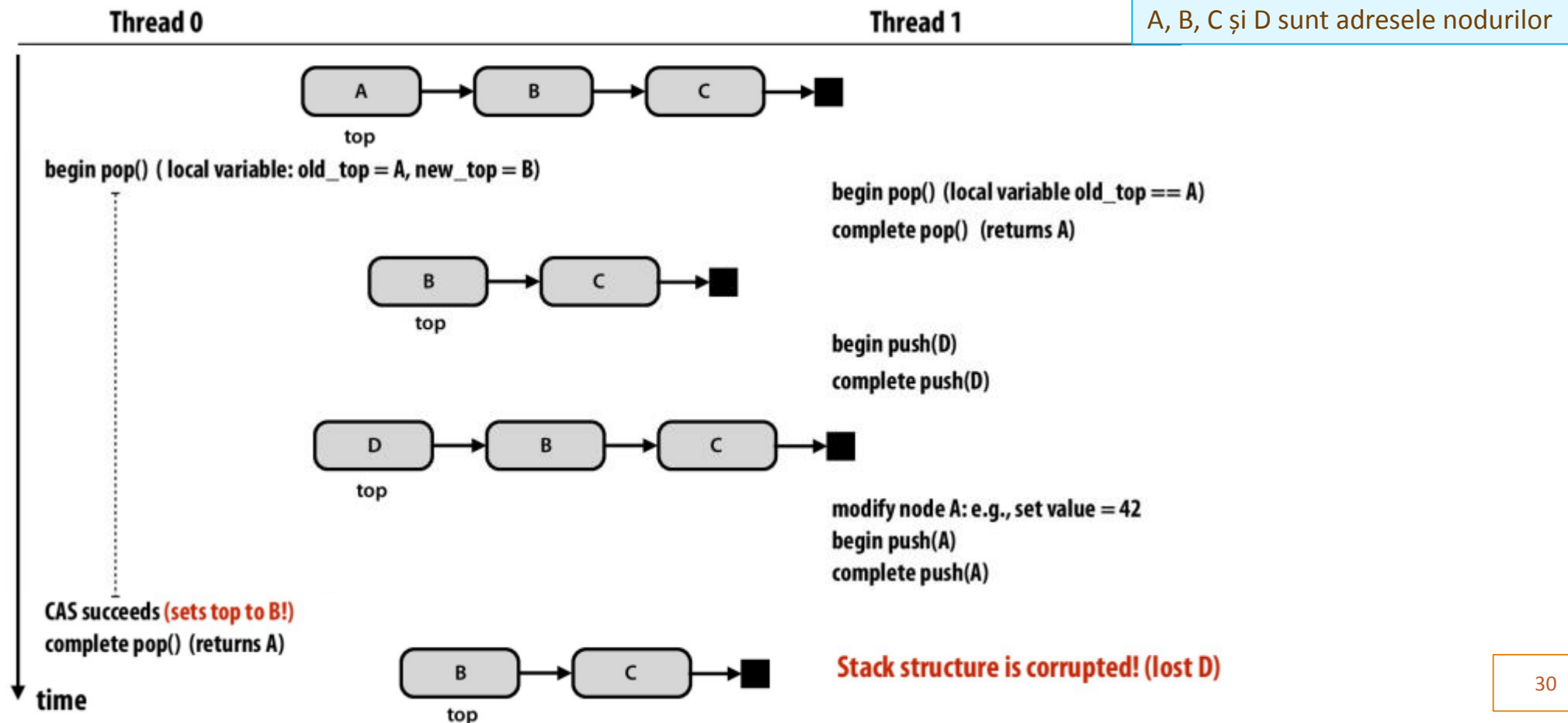
- Câte un lacăt pentru fiecare zonă a structurii
- Aici nu se folosește nici un lacăt!
  - Ce se întâmplă la swapping?
  - Accesul nu este exclusiv!
  - Atomicitatea operației de modificare este garantată de instrucțiunile atomice furnizate de hardware (compare\_and\_swap)

Sesizați vreo problemă?

# Tehnici nebloccante (lock-free)

## Problema ABA

- Un thread citește resursa partajată – este A. Un alt thread vine și modifică valoarea în B și apoi pune A înapoi. Primul thread crede că nu s-au produs modificări asupra resursei!



# Tehnici nebloccante (lock-free)

## Stivă nebloccantă

- Rezolvarea problemei ABA cu contorizarea pop-urilor


```
struct Node {  
    Node* next;  
    int value;  
};  
  
struct Stack {  
    Node* top;  
    int pop_count;  
};
```

```
void init(Stack* s) {  
    s->top = NULL;  
}
```

```
void push(Stack* s, Node* n) {  
    while (1) {  
        Node* old_top = s->top;  
        n->next = old_top;  
        if (compare_and_swap(&s->top, old_top, n) == old_top)  
            return;  
    }  
}
```

```
Node* pop(Stack* s) {  
    while (1) {  
        int pop_count = s->pop_count;  
        Node* top = s->top;  
        if (top == NULL)  
            return NULL;  
        Node* new_top = top->next;  
        if (double_compare_and_swap(&s->top, top, new_top,  
                                    &s->pop_count, pop_count, pop_count+1))  
            return top;  
    }  
}
```

test to see if either have changed (in this example: return true if no changes)



- Contorizarea operațiilor pop
- Necesită suport pentru
  - double\_compare\_and\_swap sau doubleword CAS
- Coluții alternative pentru problema ABA
  - Politici de alocare a nodurilor

# Listă înlănțuită concurrentă neblocantă

---

(ABORDĂRI ÎN IMPLEMENTAREA UNEI  
LISTE CONCURENTE LOCK-FREE)



# Tehnici nebloccante (lock-free)

## Liste înlanțuite nebloccante

- **Inserarea** este relativ simplă
  - – dacă avem doar operații de inserare concurente
- Comparativ cu abordarea cu blocare de granularitate fină
  - Nu avem overhead pentru blocarea lacătelor la fiecare nod
  - Nu avem cost suplimentar în spațiul de stocare

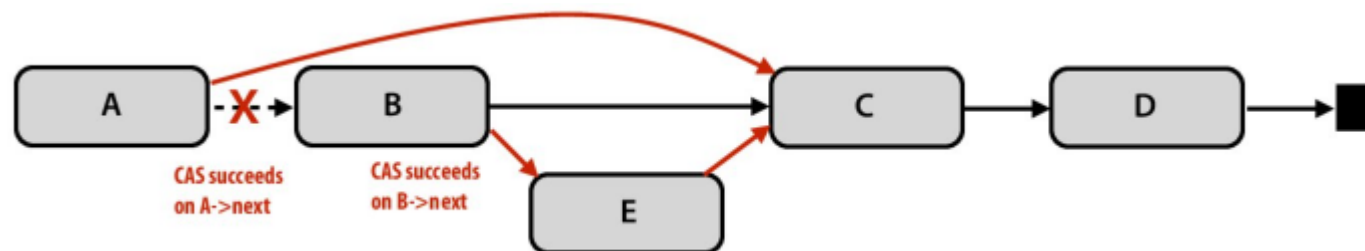
```
struct Node {  
    int value;  
    Node* next;  
};  
  
struct List {  
    Node* head;  
};  
  
// insert new node after specified node  
void insert_after(List* list, Node* after, int value) {  
  
    Node* n = new Node;  
    n->value = value;  
  
    // assume case of insert into empty list handled  
    // here (keep code on slide simple for class discussion)  
  
    Node* prev = list->head;  
  
    while (prev->next) {  
        if (prev == after) {  
            while (1) {  
                Node* old_next = prev->next;  
                n->next = old_next;  
                if (compare_and_swap(&prev->next, old_next, n) == old_next)  
                    return;  
            }  
        }  
        prev = prev->next;  
    }  
}
```

# Tehnici nebloccante (lock-free)

## Liste înlănțuite nebloccante

- **Ștergerea** este dificilă cu metode nebloccante
  - Structura devine complicată
- Exemplificarea unei probleme
  - Ștergerea lui B simultan cu inserarea lui E după B
  - B pointează către E, dar B nu mai este în listă!

- **Soluție:**
  - Ștergerea se desface în două operații atomice CAS
  - folosirea flagului de ștergere asupra nodului de șters
    - Ștergere logică
  - La ștergere mai întâi se verifică dacă flagul este setat
    - Dacă da, se încearcă mai întâi ștergerea fizică și apoi inserarea



# Verificarea corectitudinii

---

- În cazul structurilor de date secvențiale operațiile se execută câte unul pe rând în ordine
  - Corectitudinea este ușor de verificat – operațiile fiind seriale
- În cazul structurilor de date concurente operațiile nu respectă neaparat o ordine totală
  - Apelul operațiilor se pot intercala, rezultând în efecte diverse – și date posibil inconsistente
  - Condițiile de corectitudine impun o anumită ordine totală – ex. Linearizabilitate

## Linearizabilitate (Condiția de corectitudine al lui Herlihy și Wing)

- Standardul de-facto în verificarea corectitudinii structurilor concurente
- **Principiu:** Întregul efect observabil al fiecărei operații asupra structurii concurente se realizează instantaneu (indivizibil)
  - adică efectul fiecărei operații este atomic
  - Ordinea totală a operațiilor trebuie să respecte ordinea de timp real
- O structură concurentă este linearizabilă dacă este "echivalentă " cu o execuție secvențială legală
- Este o proprietate **locală**: un sistem este linearizabil dacă fiecare obiect individual este linearizabil
  - Spre deosebire de alte condiții de corectitudine (alternative) – consistența secvențială sau serializabilitatea
  - → Permite **modularitate** și concurență – obiectele pot fi implementate și verificate individual
- Performanța și scalabilitatea pot fi îmbunătățite prin asigurarea unor condiții alternative mai slabe
- Deocamdată nu există o metodă generală (independentă de orice proprietăți ale structurii de date ) pentru a demonstra linearizabilitatea

# Concluzii

---

Structurile de date concurente pot fi implementate folosind abordări variate

- Cele mai importante dpdv al performanței:
  - Blocare cu granularitate fină
  - Tehnici neblocante (lock-free)

Blocarea cu granularitate fină este utilizată pentru creșterea paralelismului (și a scalabilității)

- Dar, poate prezenta probleme legate de performanță
  - În timp ce threadul este în regiunea critică intervine un page fault, o schimbare de context, un swap
  - Poate crea interblocare, înfometare, inversarea priorităților, etc.
- În multe situații, structura de date implementată folosind blocare de granularitate fină – în mod corespunzător – poate avea performanțe la fel de bune (sau mai bune) decât abordarea lock-free
  - Granularitatea fină introduce complexitate și încărcare suplimentară

Tehnicile lock-free sunt soluția neblocantă pentru evitarea overhead-ului caracteristic lacătelor

- nu sunt întotdeauna soluția optimă
  - Adesea implică efort semnificativ și complexitate considerabilă pentru asigurarea corectitudinii
  - Nu elimină competiția pentru accesul concurent la date partajate
    - Compare\_and\_swap poate eșua dacă numărul de threaduri concurente crește foarte mult – necesită spinning

# Materiale de studiu

---

- Concurrent Counting using Combining Tree, [http://www.cs.nyu.edu/~lerner/spring11/proj\\_counting.pdf](http://www.cs.nyu.edu/~lerner/spring11/proj_counting.pdf)
- Mark Moir and Nir Shavit, Sun Microsystems Laboratories, Concurrent Data Structures, <https://www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf>
- Parallel Computer Architecture and Programming, Course materials <http://15418.courses.cs.cmu.edu/spring2016/lectures>