

Aurs 1

- Moore's law
- Problema : printura numerelor prime de la 1 la 10^{10}

↳ idei: 1) împărțirea în segmente egale
*erie: numerele nu sunt distribuite exact egale,
numerele mari necesită mai mult timp

2) shared Counter

- atomic steps = etape posibile imposabile

↳ există o soluție hardware
↳ există synchronized în Java - face mutual exclusion
pentru threaduri. Doar unul poate fi acția la unice moment dat.

- proprietăți: → safety : nothing bad ever happens (M. Exclusion)

↳ liveness : smth good happens eventually
(no starvation)
no deadlock

- protocols
 - cell phone (doesn't work) - comunicare între procese
 - can protocol (doesn't work) - interruperi
 - flag { waits } - one bit shared variables that can be read or written

- producer / consumer problem
 - functionator cu can-uri
 - functionator cu flag-uri
- pentru ca o soluție să fie corectă, e necesar să fie sigură și ai că safety și livemess.
- așteptările (delays / waiting) = ~~common~~ common & unpredictable
- readers / writers ~~protocol~~ problem
 - ↳ poate fi rezolvată cu mutual exclusion → duse la execuție securității
 - ↳ poate fi rezolvată și fără mutual exclusion
- legea lui Amdahl :

(self formulat) $S = \frac{1}{1-p + \frac{P}{m}}$

$\frac{1 - \text{thread exec time}}{m - \text{thread exec time}}$
 \uparrow (nu e minus, e plus)

$m = \text{number of threads}$

$P = \text{parallel fraction}$

$1-p = \text{sequential fraction}$

$S = \text{de cate ori e mai rapid}$

Cursul 2

- evenimente (events) \rightarrow imstante
 \hookrightarrow nu sunt simultane
- thread = sevență de evenimente a_0, a_1, a_2, \dots pentru threadul A
" $a_0 \rightarrow a_1$ " indică frecvență de la a_0 la a_1
și ordine
- interval = timpul trăit între evenimentele a_0 și a_1
 $A_0 = (a_0, a_1)$
 - \hookrightarrow se pot intercala
 - \hookrightarrow pot fi diferențe
- precedență întrevalelor $A_0 \rightarrow B_0$
- - $\hookrightarrow A \rightarrow A$ numit fals (ineflexiv)
 - $\hookrightarrow A \rightarrow B$ implica $B \rightarrow A$ (anti-simetrică)
 - $\hookrightarrow A \rightarrow B$ și $B \rightarrow C$ atunci $A \rightarrow C$ (transitivă)
 - \hookrightarrow pentru orice $A \neq B$, avem fie $A \rightarrow B$, fie $B \rightarrow A$strict partial order
- cum rezolvăm mutual exclusion - net scalable & impractical,
but good for understanding:
 - \hookrightarrow locks \rightarrow critical section concept
- deadlock-free \rightarrow dacă un thread nu returnează lock-ul, sistemul
niciun avansat
- starvation-free \rightarrow threadurile au progres și (eventual) returnează
lock-ul.
- deadlock poate fi și pe execuție sequentială

- algoritmul lui Peterson

- ↳ se selecțează o victimă pt. a avea înfrântă
- ↳ și selecțează de "victimă" pt. a genera coda prioritățea
- ↳ este deadlock-free (din analiza reacțiunii critice, slide 62)
- ↳ este starvation-free

- bounded waiting

- ↳ lock(s) este împărțit în două părți:
 - (mn. limită de pos.)
doar un interval (Δ_A)
 - (mn. medie limită de pos.)
waiting interval (V_A)

- n-bounded waiting

- ↳ B nu poate intra în cureauța lui A de mai mult de 'n' ori
- ↳ n=0 înseamnă FCFS (First Come, First Served)

- Lamport's Bakery Algorithm

- ↳ FCFS pentru N threads
- ↳ ca un număr și astupă pâine când toate celelalte numere strict mai mici decât time care sunt satisfăcute

- Java memory model - câteva informații generale

- ↳ ce înseamnă "sincronizare corectă" (slide 95)
- ↳ optimizări: → writes within threads
 - relation
- ↳ happens before relationship

Cursul 3

- exemplu cu lock-based queue
- costrucția și progres → trebuie specificat ce înseamnă safety și liveness
- obiect secvențial → fiecare obiect are o stare, date de (slide 29)
 - multime de componenți/atribute.
 - un obiect are un set de metode (singura modalitate de a manipula starea curentă)
- liniarizabilitate
- invocation metăfizică:
$$\underbrace{\text{thread}}_{\text{ }} \cdot \underbrace{\text{object}}_{\text{ }} \cdot \underbrace{\text{method}(\text{args})}_{\text{ }} \quad \left. \begin{array}{l} \text{generating} \\ \text{invocation} \\ \text{(slide 102)} \end{array} \right\} \text{well-formed histories}$$
- response metăfizică:
$$\underbrace{\text{thread}}_{\text{ }} , \underbrace{\text{object}}_{\text{ }} : \underbrace{\text{result}}_{\text{ }}$$
- exemplu de istorii secvențiale echivalente
↳ teorema compunerii (slide 130)
- punctele de liniarizabilitate → sunt astăzi când lock-urile sunt elibile.
Answers
- coherență secvențială → nu putem redonda metode ale aceluiași thread
↳ putem redonda metode care nu se suprapun din thread-urii definite

- progress conditions \rightarrow deadlock-free = um anumé thread ajuung sa-
(slide 162) $\qquad\qquad\qquad$ definite lock-out condition.
- (slide + 163)
nice graph
 - \rightarrow starvation-free = toutes threads ajuung sa definite
lock-out intr-un final.
 - \rightarrow lock-free = un thread ce apelaser o metoda
va returna condusa.
 - \rightarrow wait-free = fiecare thread ce apela o metoda
va returna condusa

Cursul 4

- spin / busy-waiting = un proces care testează în continuu o anumită condiție
- give up lock = maybe goes to sleep?
- Spin lock implementation - TAS (test and set)
 - ↳ cunoscut în Java ca `getAndSet()`,
 - ↳ lock = free : TAS = false {acquisizione}
 - ↳ lock = taken : TAS = true
 - ↳ release : TAS (false)
- TTAS (Test and test and set lock)
 - ↳ slide 21
- deasemenea threadurile folosesc cache, testează mai rapid, de aici rezultând creșterea în ~~de~~ performanță timp
- "volatile" keyword in Java
- conceptul de "delay" pentru spin lock:
 - ↳ simple delay/back off
 - ↳ exponential back off
- back off TTAS

• Anderson Queue Lock

↳ se poate a realiza de flaguri cu un singur monitor
proces cand se permanc

Cursul 5

- CLH Lock
 - FCFS order
 - small, constant size overhead per thread
 - conceptul de ccNUMA și xNUMA machines
 - asigură până când producerul este FALSE
- MCS Lock
 - FCFS order
 - small overhead
 - beneficiar când există high contention
(multe thread-uri vor să ia lock-ul)
- Ce facem în situația în care vrem să nu mai asigurăm după un mod? Avem nevoie de o soluție "băndă" (gracefully)
 - marșează nodul curent ca final skipable
 - felicită la queue based locks

Cursul 6

- invariant = proprietate care este menită adevărată
- exemplu relevant referat de obiect parsoajat: set with add
remove
contains

- ① coarse-grained lock → simplu (pun lock pe tot obiectul)
 - ↳ suflare de hotspot + bottleneck.
- ② fine-grained lock → mai complex
 - ↳ metoda care lucraște pe segmente / partiuni diferite nu trebuie să se exclude
 - ↳ idee: punne lock
 - idee pentru remove: punne lock pe precedesor și machul către stoc.
 - ↳ punct de lățăriabilitate → fie când starea listei s-a schimbat (elementul a fost găsit)
 - ↳ fie când am pus ultimul lock înaintea de a returna.
 - ↳ pentru add, lock pe predecessor + succesor
- ③ optimistic - grained lock → punne lock-uri doar când fac operații.
 - ↳ mergi prin lista fără locks ⇒ găsești elementul ⇒ punne lockuri ⇒ verifică stare
 - ↳ pentru add: ab/c intre b și d
 - ↳ verifică că head → b ①
 - verifică că b → d ② (să fie 113)

avantaj
să fie 113

Cursul 7

① Lazy list = arhitectură cu implementarea optimistă
+ contains() nu se folosește de lockuri.

↳ rezolvă (sau încearcă!) problema remove()-ului, prin
două mai concrete: logical delete și physical delete

- avantaje pentru lock free data structures (slide 38)
 - au garantat progressul

② Lock free list → folosește Compare And Set

↳ fucking hard :-)

Cursul 8

- monitorenă
 - ↳ synchronized f) se blochează pe o conditie Reentrant Locks f) înlocuiește spinning
- coda
 - ↳ bounded
 - astăptării cînd coada e plină pt. add()
 - ↳ unbounded
 - astăptării cînd coada e goală pt. remove()
 - ↳ aruncă eroare cînd dai remove() pe coadă goală
- exemplu : Bounded Queue
 - ↳ sentinel node
 - ↳ head
 - ↳ tail
 - ↳ degLock
 - ↳ engLock
 - ↳ size
- pentru eng :
 - punte lock pe engLock
 - face getAndIncrement pe size
 - dacă size era 0, face notifyAll pe degLock
 - dacă size este maxim, așteptă
- pentru deg : același metode
- problema cu "last wake-ups"
- Lock-free queue [generalizat.]
 - ↳ eng (slide 68)

Cursul 9

- ABA problem - când nu are Garbage Collector și unei să nefiește modulile exemplu de pe wikipedie cu pointeri
- Atomic Stamped Reference
 - stiva - Stack free stack
 - ↳ try Push, operatii de push
 - ↳ pop
 - ↳ utilizare CompareAndSet
 - ↳ în idee, stiva este mai dignăba "sequentială"
 - ↳ ideea de elimination - backoff: un push și un pop se anulează unul re altul.
 - afel numită și "linearizable stack"

- lock-free queue - eng (slide 68)

Causal 10

- hashes — coarse grained locking: lock per table or hash
 - ↳ fine grained locking: lock per frequent bucket, following lists simple implementation (fine locks)
- magic numbers
- sorters, reversed bits, splits
- closed (chained) hashing
- open address hashing (slide 68)
- linear probing
- quadratic hashing

~~QUESTION~~

Cursul 11

- Cum rezolvăm cîndultarea matricelor pe thread-uri?
(discuție)
- thread pool
- Parallel Fibonacci - cîneșteori, folosit ca exemplu
- Cum măsurăm eficiența unui program paralel
- Work Law: $T_p \geq \frac{T_1}{P}$ (cînd-um pas, nu pot. să facem mult de P trăsai)
- Critical Path Law: $T_p \geq T_{\infty}$ (nu pot. bate rezurse infinite)
- Cum făcî procesorul să munca mai ușor că la cînd-ai.

Laboratoriu

- Laboratorul 1 : synchronized, Reentrant Lock, volatile
- Laboratorul 2 : semafor (signal, wait) - semafor binar
 - ↳ un fil de lock implementat astfel diferență semafor vs lock
- Laboratorul 3: tipuri de date atomice în java
- Laboratorul 4: Thread Local
 - algoritmul lui Peterson revizuit
 - linerezabilor $\xrightarrow{\text{implicit}}$ consistent sequential
- Laboratorul 5: monitor
 - condition
 - wait
 - notify
 - notify All
 - await
 - signal
 - signal All

⇒ au synchronized
⇒ au lock-uri

Atunci când un thread e suspendat, își pierde lock-urile.
când e reactivat, el recăștează
- Laboratorul 6: lock-uri readers / writers
- Laboratorul 7: —
- 8: —
- 9: —
- 10: coda pentru eliminarea lock - based / lock - free
- 11: