

The ultimate

APD

Notebook

(For noobs...)

Exam practice

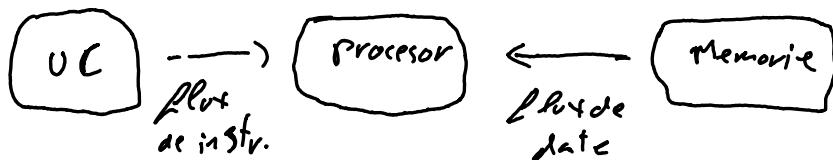
serial :

- algoritmi mai buni
- hardware mai bun

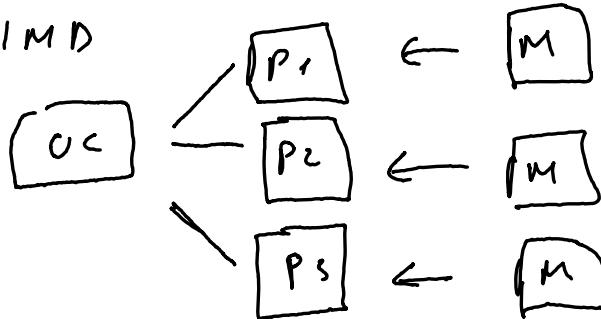
parallel :

- mai multă procesare în același timp

- SIMD (modelul clasic de execuție)
single instruction / single data

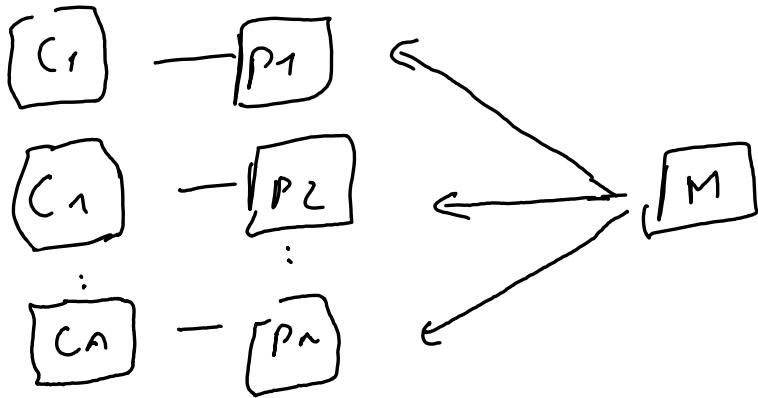


- SIMD



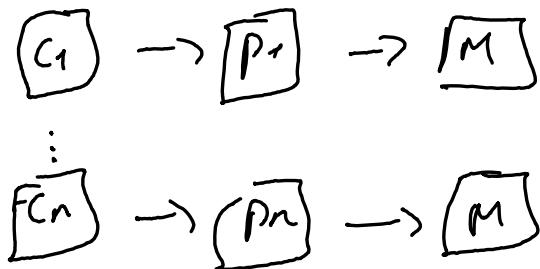
UC dă o comandă (tăi execută Load A),
asteaptă să se termine tăta. Repeat,

- M1 SD



Decriptare: aplici mai multe soluții de decriptare pt același text.

- M1 MD



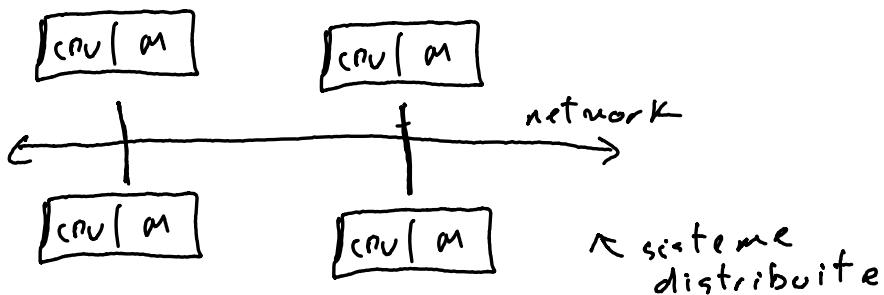
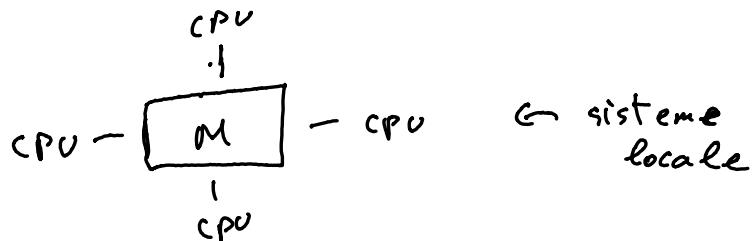
sisteme distribuite \Rightarrow

Shared memory.

- exclusiv \rightarrow se așteaptă procesorul pe rând.
- concurent \rightarrow se citesc fără să se întâlnească

citire CR - 1 pas

citire ER - $\log(N)$



Tipuri de paralelism

- bit level
- instruction level
- task level.

Proces = aplicație serverială.

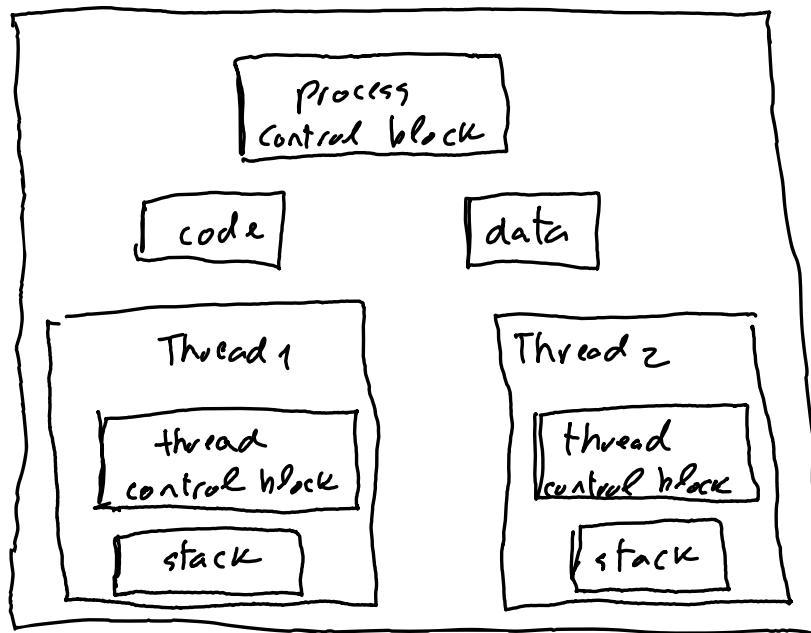
comunicare intra-procese:

- prin sockets
- prin scriere / citire din fiziere.

Notări pseudocod:

- $\text{do } s_1 \parallel s_2 \parallel \dots \parallel s_n \text{ od}$
ex.:
$$\left\{ \begin{array}{l} x=0; \quad y=0; \\ \text{do } x=x+1 \parallel y=y+1 \text{ od} \\ z=x+y; \end{array} \right.$$
- $\text{do } [\text{cuantificator}] \{ s_j \}$
ex: $\text{do } [j=1 \text{ to } n] \{ a[j]=0; \quad b[j]=0; \}$

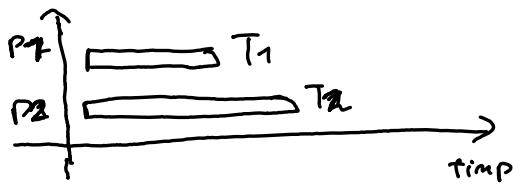
process : control block | code | data | stack



- Aplicații care vor să facă multe task-uri simultan = mai multe threaduri
- Multe procesări pe contitate mare de date = mai multe procese

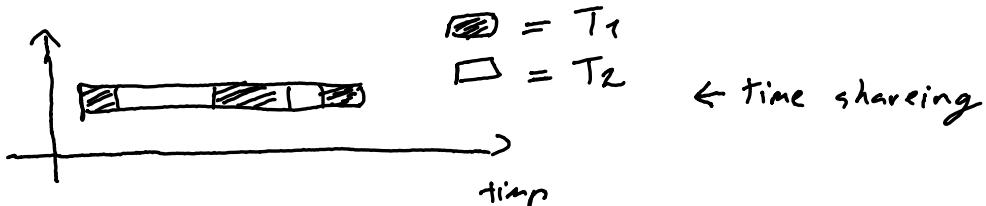
nr proceșe \Rightarrow nr. cores.

???



dacă am mai multe thread-uri decât procese???

O.S. aloca un singur core pt mai multe thread-uri ale mai multor procese (prin scheduler).



Hyperthreading (fără intel) = coreurile pot executa mai multe thread-uri.

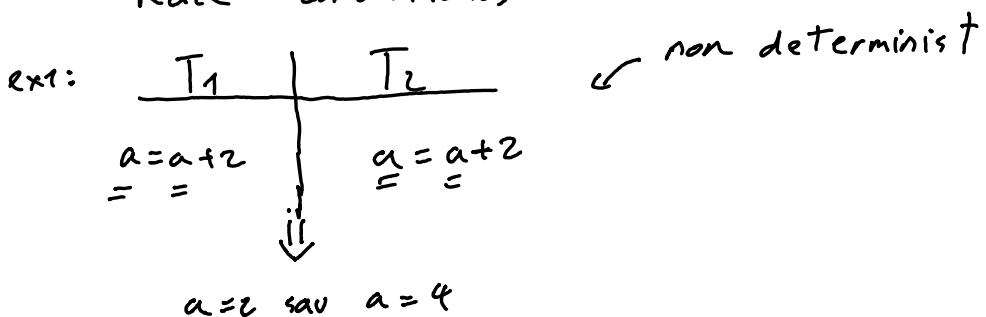
Multi tasking : 1 task pe rând / core.

Multi threading : 1 / mai multe threaduri simultan

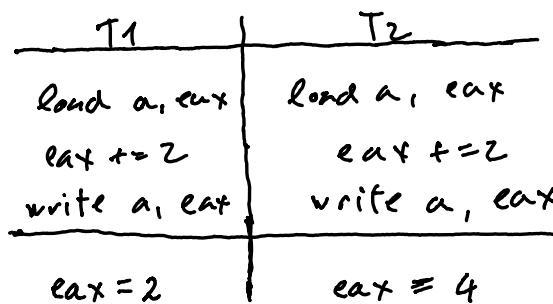
Pot fi acasă mai multe task-uri pe mai multe core - urii.

Pot fi acasă mai multe thread-uri pe un singur core.

Race conditions:



ex2: $a = 0$



Cum rezolvăm race conditions ??

Primitive de sincronizare:

- instrucțiuni atomice:

- actualizarea se execuță complet, neîntrerupt.

- semaforoare:

- mutex == semafor binar.



secțiuni critice == zona de cod se execuță de un thread o dată.

- bariera:

- forțează procesele să se aștepte între ele.

I Atomics :

ex. add 64 biți pe 32 bit registers.

load Aq eax

load Bq ebx

eax += ebx

write Cq, eax

load Aq eax

load Bq ebx

eax += ebx

write Cq, eax

← pot fi întrerupt și să am doar jumătate de C modificat

↓
când un thread începe să scrie C, nu e întrerupt nănă nu scrie și C?

II: Mutex

- Solotia lvi Dekker

```
wants_to_enter [0] = true;  
while (wants_to_enter [1])  
| if (turn != 0)  
|   wants_to_enter [0] = false;  
|   while (turn != 0)  
|     wait;  
|   wants_to_enter [0] = true;
```

// CRITICAL SECTION

turn = 1

wants_to_enter [0] = false;

T0		T1
turn = 0		

want = \exists ein thread will
wrea scritpe.
turn = e random thread will x

want 0	!want 1	\rightarrow T0 execute instructions; schimba turn.
!want 0	want 1	\rightarrow \emptyset
want 0	want 1	\rightarrow T1 want=false; wait; T0 execute; To schimba turn; T1 want=true;

want 0	!want 1	\rightarrow \emptyset
!want 0	want 1	\rightarrow T1 execute instructions; schimba turn.
want 0	want 1	\rightarrow T0 want=false; wait; T0 execute; schimba turn

• Solving the Dijkstra.

L₂: //CRITICAL AREA

$b[i] = \text{true};$ } i no mai urca sau
 $c[i] = \text{true};$ bacă cera.

$s[i] ==$ semnul lui i
 $b[i] ==$ ivrea între dacă $b[i] = \text{false}$.
 $c[i] ==$ o rată randul lui i
 $i ==$ thread id
 $j ==$ thread id
 $k ==$???

• Solutia lui Peterson

`flag [0] = true ;`

PO-gate : turn =1;

// CRITICAL SECTION

flag [0] = false;

- lași călărat în urad
să lucreze

Flag {0} == vowel access.

turn == random cui este.

• Soluția cu asistare HardWare.

```
while (test-and-set(lock));
```

// critical section

lock = 0

Concavità:

Dekker
Dijkstra
Peterson

} solutii prin busy
waiting

Hardware } for busy waiting

Probleme care nu au soluție paralelă

Huffman decoding

DFS

hash of hash of hash ... of string
outer loops of simulations.

P complete problems

Pt calculul start / stop.

ceil vs floor



Potrivit ptc oferă garantă
că trase de pragul limită.

Performanța programelor:

T timp de execuție

P nr de nuclee

G timpul celui mai bun algoritm secvențial

I: S speedup

$$S = \frac{T}{G} = P \text{ (ideal)}$$

$\neq P \text{ (real)} - \text{create/join costă}$

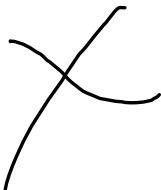
II: E eficiență

$$E = \frac{G}{C} = \frac{G}{TP} = \frac{S}{P}$$

costul

La nivel de threaduri

La nivel de procese



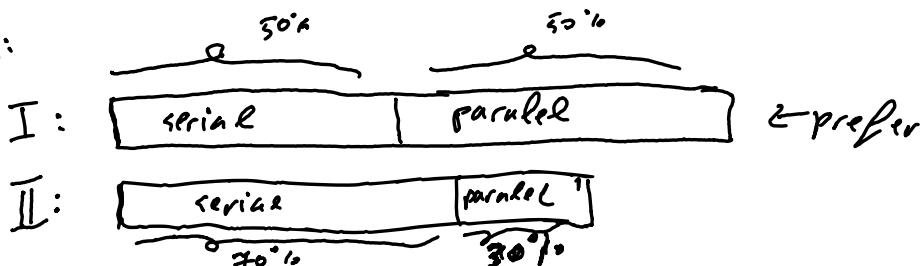
Legea lui Amdahl

În fiecare algoritm este un procent de operații care nu pot fi paralelizate (ρ).

$$\left\{ \begin{array}{l} T = \rho G + (1-\rho) \cdot \frac{G}{P} \\ S = \frac{G}{T} = \frac{G}{\rho G + (1-\rho) \frac{G}{P}} = \frac{1}{\rho + \frac{1-\rho}{P}} \end{array} \right. \leq \frac{1}{\rho}$$

$$S \leq \frac{1}{\rho} \Leftrightarrow T \geq \rho \cdot G$$

E.x.:



prefer I ptc are mai multe operații care pot fi paralelizate, chiar dacă instrucțiunile serial se execute mai rapid la II.

De ce ???

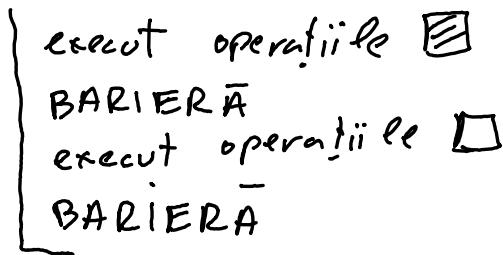
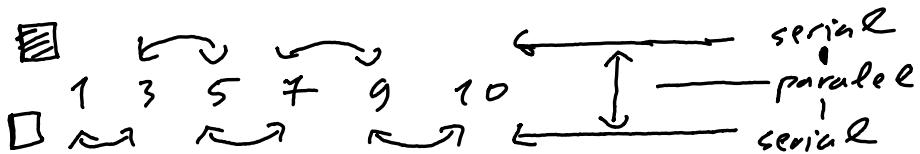
Speedup-ul este limitat de procentul de operații care nu pot fi paralelizate:

$$I: S = \frac{1}{5} \quad \Rightarrow \text{prefer I}$$

$$II: S \leq \frac{1}{7}$$

OETS

Operări care pot fi executate în paralel:



Complexitate: N operații (comparații) \Rightarrow
 $\frac{N}{P}$ iteratii

$$\Rightarrow O\left(\frac{N^2}{P}\right)$$

~~$$S = \frac{N^2}{\frac{N^2}{P}} = P$$~~

Dar $G \neq N^2$; $G = N \log N$
 $(G = \text{timpul celui mai bun - sort})$

$$\Rightarrow S = \frac{\frac{N \log N}{N^2}}{P} = \frac{P \log N}{N}$$

Shear Sort

vectorii cu n^2 el \rightarrow matrici $n \times n$

iterativ :
 - riecare linie pară sortată crescător
 - riecare linie impară sortată descrescător

Paralelizare :

- sortările liniilor nu depind una de alta.
- sortările coloanelor nu depind una de alta.

Complexitate :

$\log N$ operații

2 etape

\sqrt{N} lini/coloane

$N \log N$ sortarea liniiei / coloanei

$$O(\log N \cdot N \cdot \log \sqrt{N}) < O(N \log N)$$

$$\text{decì } G = O(N \log N)$$

Sortez liniile paralel

BARIERĂ

Sortez coloanele paralel

BARIERĂ

$$\text{Complexitate: } O\left(\frac{\log N \cdot N \cdot \log \sqrt{N}}{P}\right)$$

$$S = \frac{P}{\log \sqrt{N}}$$

OETS vs SHEAR

$$\frac{2P}{\log N} \text{ cu } \frac{P \log N}{N}$$

$$N \text{ cu } \log N^2$$

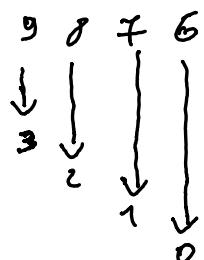
$$2^N \text{ cu } N^2$$

$$2^N < N^2 \Rightarrow$$

Shear Sort > OETS

Rank Sort

rezultă într-un vector către carele elemente din vectorul initial sunt mai mici decât elementul corespondent indexului.



\Rightarrow pozițiile pe care trebuie puse elementele de din vectorul original.

Părere: fiecare element va număra elem. mai mici decât el în paralel.

Counting
 PARIREA
 schimbarea pozitiei

Complexitate: N numere

$N-1$ comparații $\Rightarrow O(N^2)$

N schimbări

paralel: $O\left(\frac{N^2}{P}\right)$

$$S = \frac{N \log N}{N^2} = \frac{P}{N} \log N$$

Sheer Sort \Rightarrow
 OETS =
 Rank Sort



Înmulțirea de matrici $N \times N$

sequential : $O(N^3)$

parallel : pot paraleliza oricare dintre cele 2 permutări superioare (care merg pe linii / coloane). Le impart egal între thread-uri.

Complexitate: $O\left(\frac{N^3}{P}\right)$

$$S = \frac{\frac{N^3}{N^3}}{\frac{N^3}{P}} = P$$

Super-linear speed-up?

$S > P ? \rightarrow$ NU în teorie; P = limită superioară \approx

în practică: timpul de execuție parallel poate să depășească P , cind e raportat la 6.

De ce? cacheing!

Programare concurrentă în Java ::

- Concurență
- Cooperare

Sincronizare:

↳ ofice cacheing și sincronizează la memoria principală.

- fiecare obiect are asociat un lock.
- synchronized == lock pe metoda/sevență de cod.
= zona de cod din metodă devine regiune critică.
- wait = manevrarea lock-ului asociate unui obiect wait pe obiectul m din thread-ului t.
 - se face lock pe m
 - t se adaugă în lista de threaduri blocați a lui m.
- ★ t trebuie să detină lock-ul pt m.
 - t continuă execuția când va fi scos din wait-set-ul lui m:

notify:

- thread u din wait-list-ul m e scos și repos în execuție.

notifyAll:

- toate threads scacse. Numai 1 are lockul
- ★ t trebuie să detină lock pt m.

How it works!



din thread-ul T:

t este blocat ← apelez synchronized pe M
(e pus intr-o mulțime de threaduri)
blocate asociate din thread-ul V:
(cu n)

v este deblocat ← apelez synchronized pe M
(scos din lista)

(\Rightarrow) un thread se pune singur pe wait, până când primește notify (semnal că poate să continue) de la alt thread.

volatile int i;

se actualizează pentru toata thread-urile. Nu ajunge in cache.

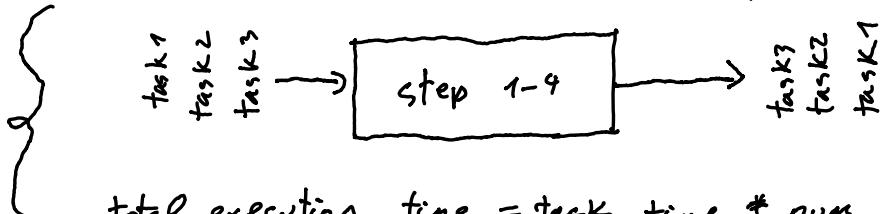
Pornirea / distrugerea threads = costisitor.

↓
folosesc Thread Pooling. = replicated

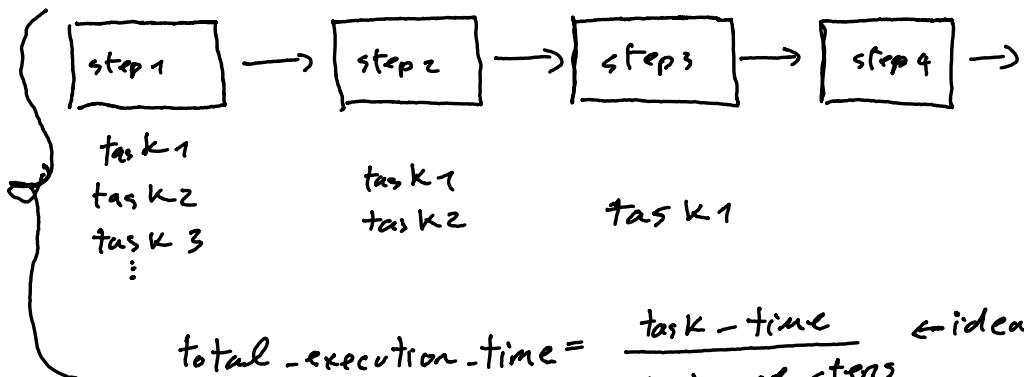
Atomic (\Rightarrow same as C.

Abordare a soluțiilor de paralelizare

I. Modelul Pipeline sf paralelism pt seturi distincte de date.
ex: calcul polinomial.



$$\text{total_execution_time} = \text{task_time} * \text{num_tasks}$$



$$\text{total_execution_time} = \frac{\text{task_time}}{\text{number_of_steps}} \leftarrow \text{ideal.}$$

Ex.: Sorting

trebuie să fie atâtă
pași căreia în vector.

→ timp: $2N - 1$ → N pași introduc
N-1 propagare
până la ultima
pozitie



← n unități de comparație (pași din pipeline)

premăște valoare din stânga.
compară cu valoarea locată.
retine val mai mică.
aruncă în dreapta val mai mare

Valorile sunt ordonate, dar în memoria locală
a pagilor.

1. se pasează la s_3
când s_3 primește, pasează val locală la s_2
.....
 s_1 le scoate.

2. în mom. în care s_1 nu mai are ce să pri-
mească, scoate spre stânga.

— — — s_2 — — —
.....

1. durează $2N - 2$ pagi

?
un procesor întâi primește val locală
apoi o aruncă în stânga $\Rightarrow 2(N-1)$

2. durează $N - 1$

\Rightarrow Total =

$$1. 2N - 1 + 2N - 2 = 4N - 3$$

$$2. \underbrace{2N - 1}_{\text{la primire}} + \underbrace{N - 1}_{\text{la scoatere}} = 3N - 2$$

N Queens Problem

Iterativ: Back Tracking

$\begin{matrix} 0 & 1 & 2 & 3 \end{matrix}$ ← liniiile din matrice
 $\begin{matrix} 1 & 3 & 0 & 2 \end{matrix}$ ← coloanele pe care punem damele
încercăm toate posibilitățile cu BT.

Paralel:

1. Replicated Workers

task_{1-n}: poziționarea primei dame

workerii iau task-urile 1-n

încearcă poziționarea damei pe următoarea col.
daca este valid, pune noul task în coadă.

Binary Search

Iterativ : LAMBDA - -

Parallel :

1. spargi vectorul ℓ în subintervale
compari cu capetele fiecărui subinterval
când găsești subintervalul bun,
ți împart pe el în $N/2$ threads. BARIERĂ.
Nu las celelalte threads să caute inutil,
ci le dai noi subintervale.

$$\mathcal{O}(\log_p N)$$

Obs: dacă împart în $p+1$ intervale o să
aveam doar o singură comparație, în
loc de 2 (capetele)

$$\mathcal{O}(\log_{p+1} N)$$



$g = 1$ // capăt stânga interval curent

$r = n$ // capăt dreapta interval curent

$x, k = \infty$ // $x = \text{căutat}$; $k = \text{poziția lui}$

$\theta = \sup (\log(n+1) / \log(p+1))$ // nr pasi

enun sens {stânga, dreapta} // sens de căutare: st \rightarrow dr

sens $\in [o : p+1]$ // $C[i] == +$ e la st / dr procesului i

$C[o] = \text{dreapta}$; $C[p+1] = \text{stânga}$

$j \in [o : p]$ // poziția din sir inspectată de procesului

To Be Continued ...

Producer - Consumer

1. 1 producători } 1 buffer, partajat
2. 1 consumator

1. Produce date și le pun în date
2. Consumă date din buffer

Cefac când nu am ce lăua / pune din în buffer?

Semafóare!

sem empty = 1 ; full = 0;

buff[1..k];

sem empty = k ; full = 0;

1. while (1)

produce (val);

P(empty)

buff = a;

V(full)

last = 1;

while (1)

produce (val);

P(empty);

buff[last] = val;

last = last % k + 1;

V(full);

2. while (1)

P(full)

w = buff[i]

V(empty);

take & (buff);

first = 1;

while (1)

P(full)

w = buff[first];

first = first % k + 1;

V(empty)

take & (w);

1. N producători } 1 buffer, partajat
 2. M consumatori

mutex1, mutex2;
 first = last = 1
 buf[k];
 sem empty = k ; full = 0;

put in shared
memory so
everybody sees
it.

1. process producer (1...N)

```

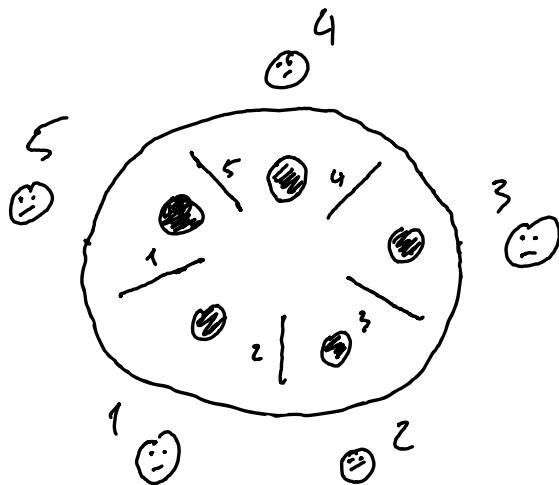
  while (1)
    produce (val);
    P (empty);
    mutex.lock();
    { buf[last] = val; } zonă critică ptc
    { last = last % k + 1; } N producători fac
    mutex.unlock();          operații asupra unei
    V (full);               zone comune de
                            memorie
  
```

2. process consumer (1...M)

```

  while (1)
    P (full)
    mutex.lock();
    { w = buf[first];
      first = first % k + 1;
    }
    mutex2.unlock();
    V (empty)
    take (w);
  
```

Problema Filosofilor



proces Filozof { $i = 1 \dots 5$ }

```
while (true)
    {
        ia betisoare;
        mananca;
        pune betisoare;
        gandeste;
```

\Rightarrow DEAD LOCK

Soluție I:

sem &[1:5] = ({<} 1);

process filozof [1...4]

```
while (true)
    P(&Ei3);
    P(&Ei+1);
    mânâncă;
    V(&Ei3);
    V(&Ei+1);
    săndesc
```

{ a filozofi merg
și iau întai bețigorul
din dreapta apoi pe cel
din stânga

process filozof [5]

while (true)

```
P(&E1);
P(&E5);
mânâncă;
V(&E1);
V(&E5);
```

{ al c-lea semafor
ia întai bețigorul din
stânga apoi din
dreapta

Ineficient: are comportament sequential.

Soluție II:

- filozofii cu id par încep cu stânga apoi cu dreapta.

- filozofii cu id impar încep cu dreapta apoi cu stânga.

↓
balansăm încărcatura

Problema cititorilor și scriitorilor

Cum ajungi să tratezi accesul la o resursă critică ???

- | | | |
|---|--|---|
| N cititori
↓
<ul style="list-style-type: none"> • n cititori pot citi pe un mom. dat. • citește doar dacă nu scrie nimenei. | | N scriitori
↓
<ul style="list-style-type: none"> • 1 singur scriitor să scrie la un mom. dat. • scrie doar dacă nu citește nimenei. |
|---|--|---|

I.: Excludere mutuală strictă: not good for readers.

II:

↙ number of readers

- {
- doar primul reader face P
 - doar ultimul reader face V

int nr=0; sem mutex R = 1; rw = 1;

process Cititor [1... M]

while (true)
 P (mutex R);

nr = nr + 1;
 if (nr == 1) P(rw); //primul proces
 V(mutex R);
 citește_din_memorie;

P (mutex R);
 nr = nr - 1;
 if (nr == 0) V(rw); //ultimul proces
 V(mutex R);

process Scriitor [1... n]

while (true)
 P (rw)

scrie_in_resursa;
 V (rw)

• execuție conditonală

Problema cu II : starvation

cititorii pot citiesc unul după altul iar scriitorii nu au timp să mai scrie.

Fairness ??

Sincronizare condiționată!

Cum??

invariant global $\frac{nr}{nw} \leq 1$

RW: $(nr == 0 \text{ || } nw == 0) \&\& nw \leq 1$

trebuie nimeni nu citește sau nimeni nu scrie și poate scrie maxim o persoană.

Reader: incrementez nr dacă nu scrie nimeni

Writer: incrementez nw dacă nu scrie nimeni și nu citește nimeni.

III:

int nv=0; nw=0;

process Cititor [1...m]

 while (true)

 l \langle await (nw == 0) nr++ \rangle

 citește

 l \langle nr-- \rangle

rămăși locat până când
se înălță deplinăste
condiția

operatie
atomică

process Scriitor [1...n]

 while (true)

 l \langle await (nw == 0 $\&\&$ nv == 0) nw++ \rangle

 scrie

 l \langle nw-- \rangle

IV Split binary Semaphore.

//

semafor format din mai
multe semafoare binare
combinante, tratate unitar

- { 1. semaforul e : excludere mutuală
2. semaforul r : semnalizare condiționată reader
3. semaforul w : semnalizare condiționată writer

→ tratate ca unul singur.

- cel mult un semafor este 1 la un moment dat.

(\Rightarrow)

suma semafoarelor componente este maxim 1

- instrucțiunile P și V se execută în excludere mutuală.

Tehnica : Pasarea ștafelei.

- ștafeta este preluată printr-o operație P asupra procesului care detine ștafeta.
- un proces poate preda ștafeta altui proces ulterior printr-o operație V pe semaforul procesului respectiv, în funcție de politica de fairness.

Politica de fairness :

- noi cereri de la cititori sunt întârziate dacă un cititor este așteptat.
- un cititor întârziat este trezit doar dacă nu există un scriitor în așteptare.

nr = 0 // nr cititori care folosesc resursa
nw = 0 // nr scriitori care folosesc resursa
sem e = 1 // întoarcere secțiune atomică
sem r = 0 // întârzire cititori dacă nw > 0 sau dr > 0
sem w = 0 // întârzire scriitori dacă nw > 0 sau nr > 0
dr = 0 // cititori înfărecăți
dw = 0 // scriitori înfărecăți

Cititor :

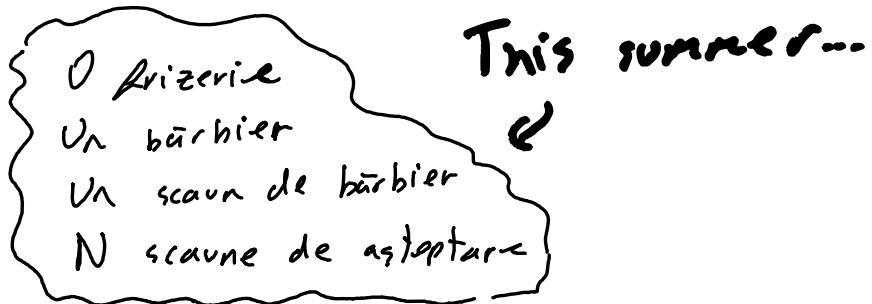
oprește alte procese să continue.
verifică dacă un scriitor vrea să scrie
da: se pună în așteptare,
lucă alte procese să continue,
predă stațeta scriitorului
se pună să citească
verifică dacă mai sunt alii cititori care așteaptă
să citească
da: le predă stațeta
citește
verifică dacă acum nu citește nimenei și e cineva
care vrea să scrie
da: scade nr scriitori care așteaptă
și predă stațeta
verifică dacă nici nu scrie nimani nici nu citește
da: lucează alte procese să treacă.

Scritor

împreună alte procese să treacă
verifică pe dacă acum citește cineva
fie dacă cineva vrea să scrie
dacă se pună în așteptare
lastă alte procese să treacă
predă ștafeta scriitorului

scrie } nu lasă alte procese să treacă
mai 0 } scrie
persoana } lasă alte procese să treacă
data verifică dacă sunt cifri care așteaptă
și nimic nu vrea să scrie
da: le predă și făcheta cifratorilor
verifică dacă așteaptă cineva să scrie
da: îi predă lui și făcheta
verifică dacă nici nu așteaptă cineva
să scrie nici să citească.
da: lasă alte procese să treacă.

Problema Bărbierului



- O clienti \Rightarrow bărbierul doarme
- șosește client \Rightarrow trezeste bărbierul / așteaptă
- nu are loc să aștepte \Rightarrow pleacă.

Bărbier doarme == semafor bărbier Gata
client == semafor ocupă scaun
semnal pe bărbier gata

scavne_liber=0, clienti=0, b_gata=0
proces bărbier

```

    | white (true)
    | P (clienti)
    | P (scavne)
    | scavne_liber++
    | V(bărbier_gata)
    | V scavne
  
```

```

proces clienti{1 .. N}
  | while (true)
  |   P (scavne)
  |   if (sc_liber > 0)
  |     sc_liber --
  |     V (clienti)
  |     V (scavne)
  |     P (bărbier_gata)
  |   else
  |     V (scavne)
  
```

Problema Secțiunilor Critice

Până acum:

verifici dacă ai voie să intri }
aștepti } semafoare.
dai voie să intre

Proprietăți:

- mutual exclusion ✓
- decizia de cine intră nu trebuie amânată la infinit X
- timeout pt regimne X
- + eficiență + fairness

I: Dekker : see back

↓ BETTER : kiss

II: Peterson's → limită la 2 threads.
Y works??

turn poate avea doar o singură valoare.

- nu merge pt mai multe threaduri ptc dacă așteptă N threads nu se ștă în threaduri.



IV: Ba Kery.

principiu : se clau bilete de ordine
se servesc în funcție de bilete.

- nu merge pt că atribuirea
tichetelor nu este atomică \Rightarrow
2 procese pot avea un ticket.

Problema : acces la ticket



V: face max pe tichete.

principiu : strigă cine e ultimul; $i = \text{max} + 1$
astfel pt că cind e un
ticket mai mic decât al meu
întru doar când sunt max.

obs: update în fundal de maxim
problema: 2 tichete pot avea același max.
pb: max + 1.



VI tie break: dacă au același ticket, se alege în funcție de poziție.

problema: pot să compar cu j altunci
când j nu define ticketul.



VII: Mai folosesc un vector care simulează când un proces a terminat de ales.

Problema: over load datorită max +1 pt id.



VIII: Când un ticket ajunge la max rămân blocat până când ajunge la ceva mai mic decât maxim.



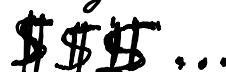
• CSE enter (i)

```
do
|   ticket[i] = 0
|   choosing[i] = true
|   ticket[i] = max(ticket[1...N-1]) + 1
|   choosing[i] = false
|   while ticket[i] ≥ MAXIMUM
|   for j = 0...N
|       while choosing[j]
|           continue
|           while (ticket[j] && (ticket[j], j) < (ticket[i], i))
|               continue
```

Inca folosim semafoare
ptc Bakery este algo
~ busy waiting \Rightarrow costisitor

• CSE exit (i)

```
ticket[i] = 0;
```



Cum implementăm o Barieră ???

CV Pasarea Stafetei

```
sem b=0; c=1;  
int nb=0;  
process proc [1...n]
```

```
# enter barrier  
P(c)  
nb = nb + 1  
if (nb = n)  
    V(c)  
    V(b)  
else if (nb > 1)  
    V(c)  
    V(b); nb--  
    if (nb > 0)  
        V(b)  
  
# exit barrier.  
P(c)  
nb  
V(c)
```

problema: procesele se pot bloca între ele.

Problema fumătorilor

Fumătorii

- asteaptă \rightarrow ingrediente
 - rulăză
 - fumează
- } au cel puțin un ingredient

Agentul are ingredientele

Soluție:

Mai trei niște cozi auxiliare care se ocupă de distribuție în funcție de ultimul ingrediant primit.

East - West Bridge

- pod prea ingust pt masini in anelele directii.
- podul face max 3 masini.



Acumulator cu reader - writer? Da.

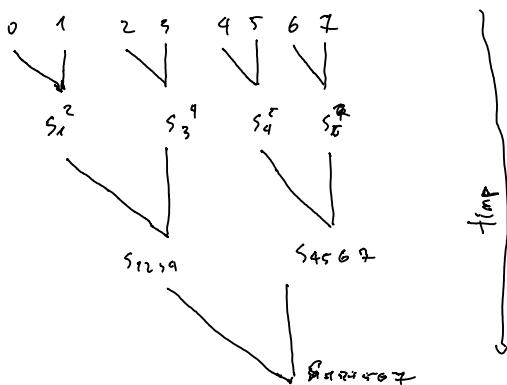
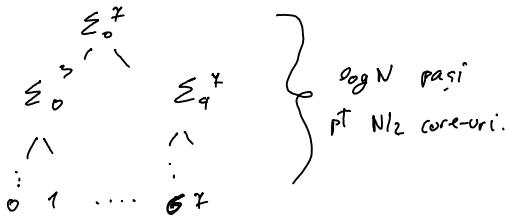
Ordinca semafoarelor
FIFO

trebuie folosit notify all
while (ticket[m] >>) } ca in
wait } Bakery.

Există deja în Java.

Paralelismul de date

Suma d. vectorului



int a [1:n]

process suma [1..n]

for $j = 1 \dots \sup(\log_2 n)$ //nr iteratii

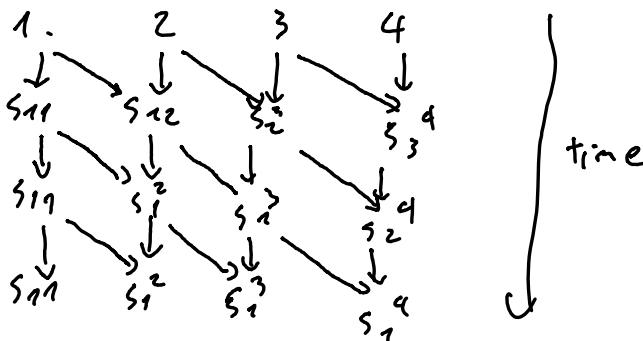
if $k \% 2^j == 0$ //micsorez intervalul
 $a[k] = a[k - 2^{j-1}] + a[k]$

bariera

Cafcu pul cumular profit

La fel doar că nu mai avem și căte 2. Le pun și pe cele impare să lucreze.

$$S_1^5 = S_2^5 + S_1^1$$



```

int a[1:n], temp [1:n]
process suma [k:1 to n]
    int d=1;
    while (d < n)
        temp [k] = a [k];
        barrier
        if (k-d >= 1)
            a [k] = temp [k-d] + a [k];
            barrier
        d += 2
    
```

Parallel Scan

scan $\circ \langle x_0 \dots x_n \rangle$

$=$

$\langle x_1, x_2 \circ x_2, \dots, x_1 \circ x_2 \circ \dots \circ x_n \rangle$

Ex: filtru.

e usor să găsești el.

egru să le pun pe poz. în vectorul de ieșire.

Soluție:

- 1 Folosim rupere paralelă pt a calcula un bit-vector pentru id. care se potrivesc.
- 2 Aplicăm sume prefix pe elementele bit-vectorului
- 3 Aplicăm paralel map pt a produce ieșirea.

Aplicarea logaritmică a oricărui filtru.

Scan = se aplică același operatie pt toate elem. vectorului.

Difuzarea unei Valori

Broadcast

$\text{int } t$
 $t = P$
 $A[i] = t;$
 }

pas 1
 procc. și punе
 val în memorie

for $i = 0 \dots (\log N - 1)$

{

 process P_j [$j = z^i + \dots + z^{i+1}$]

 $\text{int } t;$

 $t = A[j - z^i]$

 $A[j] = t;$
 }

pas 2
 la fiecare
 iteratie
 se deschide
 procesoarele

Partajare din aproape în aproape.

Operatii cu liste

1. Afișare știr cine e vecinul direct corespunzător.
2. Afișarea procesor în paralel se va întâia cine
3. este vecinul său și se va întâia cine este
4. vecinul vecinului.
5. întreb direct vecinul vecinului.

↓

doblez distanța la care
fac interogările

Algoritmi Distribuiți

Comuni rafii:

sincronă → așteptare în ordine

asincronă. → send nu e blocant
recv e blocant



nu așteptare în ordine

Canalul : chan nume-canal (tip1 id1 ... tipn idn)

grup indexat : chan rezultat [1:n] (int)
canale

operatii : send nume-canal (expr1, ..., exprn)
 receive nume-canal (var1, ..., varn)
 empty nume-canal

Ex : filtre.

Replicated Workers

Integrarea matematică

terma cand diferența ariei calculată la pasul i și pasul i+1 < eroare.

chan sac (real a, b, fa, fb, aria);
chan rez (real a, b, aria);

administrator :

{ calculează aria într-o lățime
pună aria în sac
cât timp nu s-a terminat
însumează bucatările primite.

workeri:

{ ia task din sac
spurge intervalul
calculează aria din stânga + dreapta
însumez. și fac diff dintre aria mare
și sumă.
diff < E
 └ pun la rez
else
 └ creez 2 taskuri pe fiecare jum.
 └ le pun în sac

Complexitatea algoritmilor distribuiti

Foster

Un thread poate face:

- calcul
- comunicare
- inactivitate

Complexitatea = timpul curs de la executia ultimului proces pana la terminarea primului proces.

$$T = T_{comm}^j + T_{comp}^j + T_{idle}^j$$

transmis / primire
date.
(comunicare inter/intra
procesor)

depende de
implementare
din problemei
num threads
caching

- nu are ce face \Rightarrow
load balancing needed
- nu are date \Rightarrow
(comm + comp) overlap.

$$\begin{aligned} T_{msg} &= \text{timpul} \\ &\text{transmiterii unui mesaj} \\ &= t_{\text{startup}} + t_{\text{covant}} * \text{nr_covinte} \\ &\quad !! \\ T_{msg} &= t_{\text{startup}} + t_{\text{covant}} * \text{nr_proc_comm_concurrent} * \text{nr_covinte} \end{aligned}$$

Parallel Roy-Floyd

Se descompune matricea pe rânduri.
Fiecare task are 1 sau mai multe rânduri.

for $k = 0 \dots N-1$

```

    } for i = local_start ... local_end
        }   } for j = 0 .. N-1
            }   | l[i,j]_{k+1} = min ( {l[i,j]}_{k+1}, {l[i,k]}_k + l[k,j]_k
    }

```

Problema în sistem distribuit:

un singur thread are sten matricei \Rightarrow
la fiecare pas K trebuie să facă broadcast.

$$T_{broadcast} = \log P \underbrace{(t_s + t_v \cdot N)}_{\text{time line}}$$

$$T_{Cloud} = \frac{t_c N^3}{P} + \underbrace{N \otimes_P (t_s + t_w \cdot N)}_{\text{time communicate}}$$

time
 computation

Modelul LogP

L - latency - cat capatareste mesajul

o - overhead - durata executiei send

g - gap - timp intre senduri

P - nr module Procesor / Memorie

ex.: Difuzarea valorii

$$P_0 \rightarrow P_1$$

$$P_0, P_1 \rightarrow P_2, P_3$$

$$P_0, P_1, P_2, P_3 \rightarrow P_4, P_5, P_6, P_7$$

} 3 send-uri

$$P=8; o=2; g=2; L=6 \Rightarrow 30 \text{ unitati}$$

obs.: daca fiecare proces trimite val imediat ce primește \Rightarrow mai eficient. 24 unitati

Ceasuri fizice

Ordonarea evenimentelor

Sisteme nedistribuite \leftarrow timpul este neambiguu
(dat de clock-ul sistemului)

Sisteme distribuite: \leftarrow fiecare are ceasul lui.

Diferența de timp dintre 2 calculatoare.

Sincronizarea ceasurilor fizice.

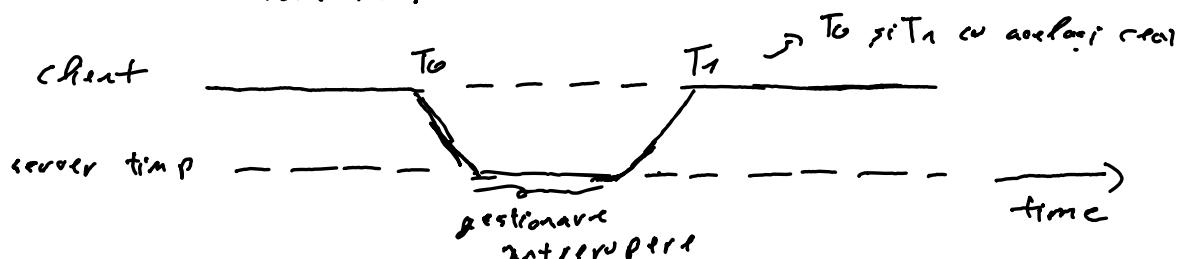
Trebue să avem garanția că atunci când citesc o dată să fie ultima actualizată.

Cum?

- algoritmul Flaviu Cristian - adaptive internal clock synchronization.

ceasurile comunică între ele periodic,
spunându-se și timpul curent.

incorrect ptc pot apărea variații în
consecuție \Rightarrow noua val. setată va fi
incorrectă.



Algoritmul Berkeley

- serverul de timp trimite periodic mesajele clientilor pt a afla timpul lor
- face o medie
- le spune cum să își actualizeze ceasurile

Le trimite ora lui

clientii te spun diff dintre ora lui și a lor.

problema : timpul \neq UTC

Network Time Protocol NTP

organizează rețea hic servere de timp :

Layer 1 {
- US Naval Observatory
- GPS
- Atomic clocks

Layer 2 { - servere importante

E complex :(

Sincronizază ceasurile până la 1-50 ms

imperfect pt \leftarrow
systeme distribuite.

Soluția (lui Lamport)

Ceașuri Logice

Un sistem distribut = multime de stări și acțiuni care schimbă starea.

Evenimentele sunt ordonate după dimenziune fizică de producere. (\Leftarrow Dificil).

Better: Ordinea relativă a evenimentelor
 (\rightarrow)

$a \rightarrow b \Leftrightarrow a$ precede în timp pe b

$a \rightarrow b \Leftrightarrow$ transmiterea + receptia unui mesaj.

$a \rightarrow b$
 $b \rightarrow c \quad / \Rightarrow a \rightarrow c$

cel logic = întreg incrementat la producerea
unui eveniment

fiecare proces are la inceput reacul lui,
initializat cu 0.

fiecare mesaj are un camp tf

lum part colosește relația \rightarrow
cum?

- ev. intern se produce:

cl ++

cl asociat evenimentului

- trimite mesaj

cl ++

tf = cl

- primește mesaj

cl = max (cl, tf) + 1

a \rightarrow b \Rightarrow cl(a) < cl(b)

Unde se folosesc ??

Semaphore distribuite

când un proces execută P sau V \Rightarrow
 \Rightarrow trebuie să facă broadcast pt
celelalte procese.

Trebuie să aștepte răspuns că se poate.
DAR: se pierde ordinea mesajelor și nu
știu cine a trimis request primul. \Rightarrow
 \Rightarrow trebuie să asociem un timestamp \Rightarrow
 \Rightarrow folosesc ceasuri logice.

Poate apărea aceiasi problemă \Rightarrow trebuie
incă o sincronizare.

Algo:

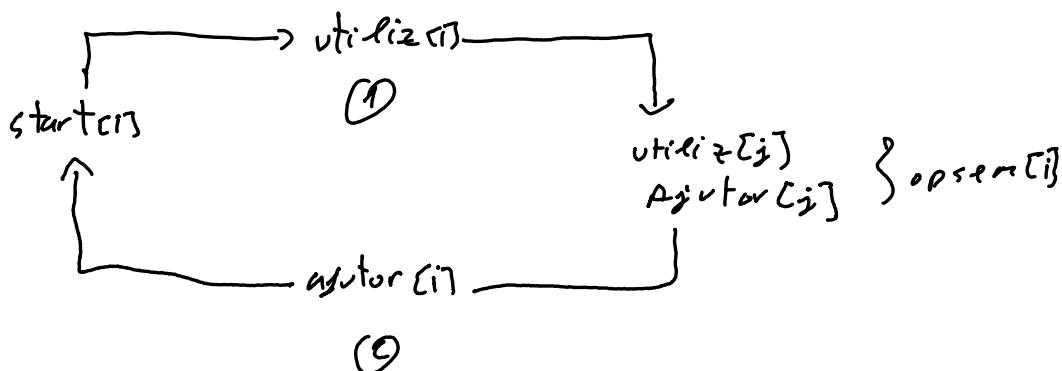
primește mesaj P, V

stochează requestul după ts

face broadcast să anunțe că a primit
stochează o variabilă s

$\left\{ \begin{array}{l} V - s++ ; \text{sterge mesaj} \\ P - iL(s > 0) s-- ; \text{sterge mesaj} \end{array} \right.$

Decuplare:



(1) aplicația în sine

(2) procese care ajută aplicația.

opsem = canal folosit de user pt că
vrea să facă o op.

start = stau blocat pînă când mă
de blocarea agutorul.

Ceașuri logice vectoriale

lămpoart asigură

$$e \rightarrow f \Rightarrow +e \subset +f$$

dar

$$e \rightarrow f \Leftarrow +e \subset +f \quad ???$$

cum funcționează?

!

{ fiecare proces are asociat un tabelu întreg.
 Local în minte căte evenimente minim
 am văzut produce de acel proces.

P_i are asociat $V_i [1 \dots n]$

$V_i[i] =$ nr de evenimente produse de P_i

$V_i[j] =$ nr de evenimente (minim / până la acel moment) executate P_j

trimite mesaj m din P_i

$P_i[i]++$

P_i adaugă V_i la mesaj cu timerstamp. vtm
primit mesaj m în P_j

$V_j[k] = \max [V_j[k], v_{tm}[k]]$; v_{tm} ;

$V_j[k]++$

Reguli

$$\begin{aligned} VT_1 = VT_2 \Rightarrow VT_2[i] &= VT_1[i], \forall i \\ VT_1 \leq VT_2 \Rightarrow VT_1[i] &\leq VT_2[i], \forall i \\ VT_1 < VT_2 \Rightarrow VT_1[i] &\leq VT_2[i] \text{ și} \\ VT_1[i] &< VT_2[i] \end{aligned}$$

vectori de
unprezentă
de timp
asociați
recursiv

$\left. \begin{array}{l} VT(a) < VT(b) \Rightarrow evenimentul a \\ \text{precede cauzal } b \\ VT(a) \neq VT(b) \Rightarrow \text{concurrente} \end{array} \right\}$

Ordonare Cauză Multicast

Procesele comunică doar cu multicast.

$$m \rightarrow m' \Rightarrow \text{livrare } p(m) \rightarrow \text{livrare } p(m')$$

Elle cărui proces are un ceas vectorial cu toate el 0
numărăm doar transmiterea mesajelor.

$V_i \in \mathbb{N}$ - evenimente de transmitere în P_i

$V_j \in \mathbb{N}$ - evenimente de transmitere în P_j
de care P_i știe

P_s transmite m

- $V_s \in \mathbb{N} \quad ++;$
- P_s adaugă V_s la m ;

P_d primește m cu v_{tm}

- P_d are m în coadă
- dacă $V_d \in \mathbb{N} = V_d \in \mathbb{N} + 1$ // dacă e următorul timestamp
 - e livrat mai departe
- dacă $v_{tm}[k] \leq v_d[k]$, pt $k \neq s$ // $k \neq s$ și
 - e livrat mai departe
 - nu avăzut
mai multe rețele
decofări,

Algoritmi Undă



Nodurile trebuie să respecte o topologie de comunicări (un nod trebuie să vorbească doar cu anumite noduri).

program distribuit \rightarrow sistem de transacții

$$S = (C, \rightarrow, I)$$

multimedie \uparrow \nwarrow setul configurații lor
configurații relație de inițiale C, C'
 transiție binară
 pe C

O execuție:

$$E = (\gamma_0, \gamma_1, \gamma_2 \dots) \quad \gamma_i \in I \text{ și } \gamma_i \rightarrow \gamma_{i+1}$$

configurație terminală \Rightarrow nu are succesor.

E maximal $\begin{cases} \nearrow \text{infinit} \\ \searrow \text{ajunge în stare terminală} \end{cases}$

\exists fungibilitate din γ dacă $\gamma \rightarrow \gamma_1 \rightarrow \dots \rightarrow \gamma_k$

Convenții :

pt sistem = tranziții + configurații

pt proces = evenimente + stări

Unei execuții E îi corespunde o secvență
+ stări.

Relația de ordine causală <

a, b două evenimente diferite

$a \prec b \Rightarrow a$ s-a petrecut înaintea lui b

Execuția E ~ F dacă

au același coloanță de evenimente

respectiv același relație causală

au același stare terminală

Here it comes...

Mesajele se transmit dependente de topologie.

Toate procesele participă.

calcul \Leftrightarrow undă

Procesele:

initiatori - primul \Rightarrow proces / stres.

neinitiatori - primul element = recv.

Cunoștințe inițiale.

- cine sunt eu

- vecinii mei

- ordinea vecinilor (setul direcției)

Numar decizii

\rightarrow decizie / proces

Complexitatea 
cunoașterea topologiei de rețea nod
setul direcției

- muchii incidente = direcția spre care conduce în rețea
- etichetele sunt la fel pt fiecare nod.

direcția $i \rightarrow j$ are sensul $(j - i) \% N$

Algoritmul inel

Răb.

Algoritmul Arbore

Aplicat pe topologii arborescentă

frunze = inițiatori.

Fiecare proces trimite un mesaj.

Când un proces primește mesaj de pe toate canalele în afara de unul decide.

trimite mesaj.

Potem scrie același cod pt initiator și follower.

E unde??

1. e calcul finit?

folosește N mesaje \Rightarrow atinge o configurație finală după N pasi

2. cel puțin un proces face decide
redus la absurd

3. decide e precedat de un eveniment -
inducție

Algoritmul Ecoo

Un singur initiator.

Se inunda cu mesaje tok care dau reply la rădăcina și stabilesc un arbore de acoperire. Pe haza arborelui, se transmit în apoi mesajele.

$\epsilon \neq$ Nr muchii mesajie

σ_{opt} timp propagare

Algoritmul Fuzelor

(Hearth Beat)

ab.

Algoritmul lui Finn

Nu are diagramă

Păstrează identificatori:

I_{NC} - mulțimea proceselor q pt care un eveniment $i_{q,p}$ precede cel mai recent eveniment în P.

N_{NC} - mulțimea proceselor q pt care fiecare vecin r are un eveniment care precede cel mai recent eveniment P.

How it works??!

Initial $I_{NC} = \{P\}$; $N_{NC} = \emptyset$

P trimită mesaj cu I_{NC}/N_{NC} când ele cresc când primește mesaj cu I_{NC}/N_{NC} actualizat (v) când P primește mesaj de la toți învecinii, este inserat în N_{NC}

când $I_{NC} == N_{NC} \Rightarrow P$ decide

Algoritmă Unui Lider

- prin algoritmul arbore
 - mai adăugăm o fază de work-up:
 - trezestă frunze
 - trătesc procesul de vot.
 - prin algoritmul inel
 - inițiatorul se anunță pe el și trimite mesaj
 - când alte noduri primesc mesaj decid și dăv mai departe.
- * În mesaj trebuie să se rețină toate identitățile găsite pt cazul ce mai nu e fi inițiatori.

Algoritmul LeLana

- fiecare proces difuzează id-ul său
- fiecare proces colectează nr. celorlalte procese
- fiecare proces afluă maximul
- procesul al cărui $nr == max$ este liderul.

FOARTE MULTE MESAJE...}

Algoritmul Le Lam - Chang - Roberts

Mesaje transmise în sensul acelor de ceas.

Fiecare proces transmite celui din dreapta un mesaj cu id-ul său.

Procesul compară mesajul cu id-ul propriu:

- 
- $m > id \rightarrow$ transmite m în dreapta
 - $m < id \rightarrow$ elimină m
 - $m = id \rightarrow$ procesul devine lider

Algoritmul Hirschberg - Sinclair

$O(N \log N)$

Lucrarea pe inel bidirectional.

Candidatura e anunțată de inițiator la o distanță de 1 față de el.

Dacă nodurile sunt de acord respond OK.

Dacă primește OK trimite la distanță dublă.

Nodul care nu dă OK devine nouă candidat.

Stabilirea Topologiei

Nodul - - -

Terminarea programelor
distribuite