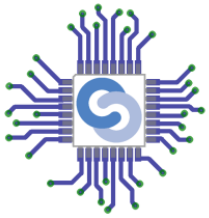


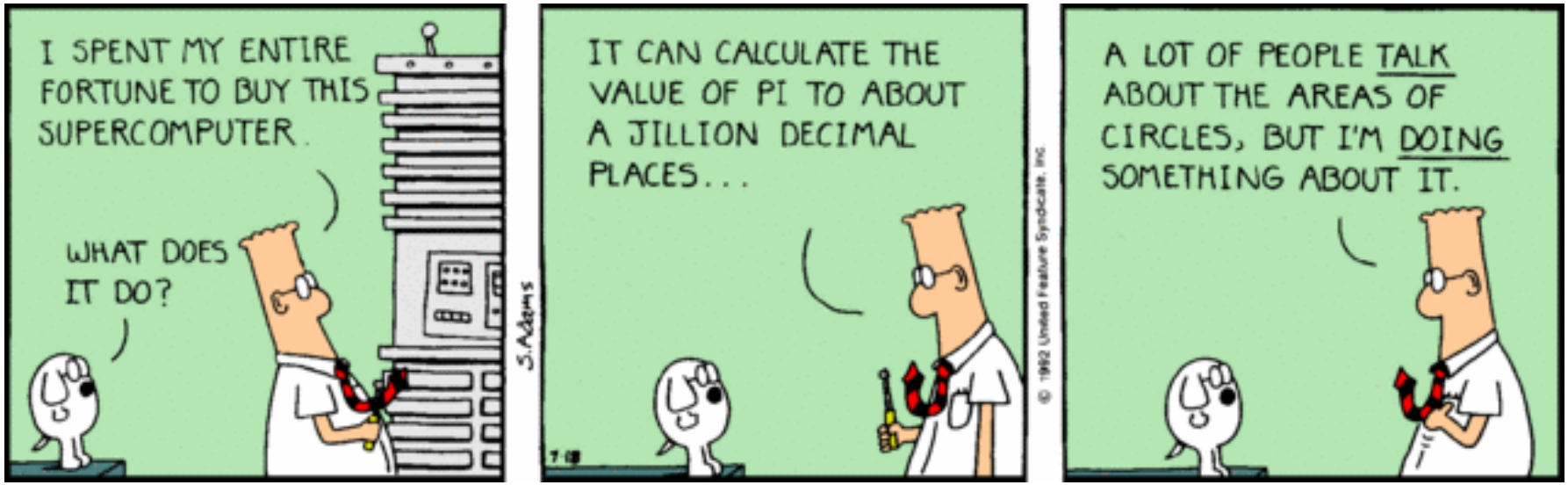
Calculatoare Numerice (2)

- **Cursul 11 –**

Multiprocesoare 2

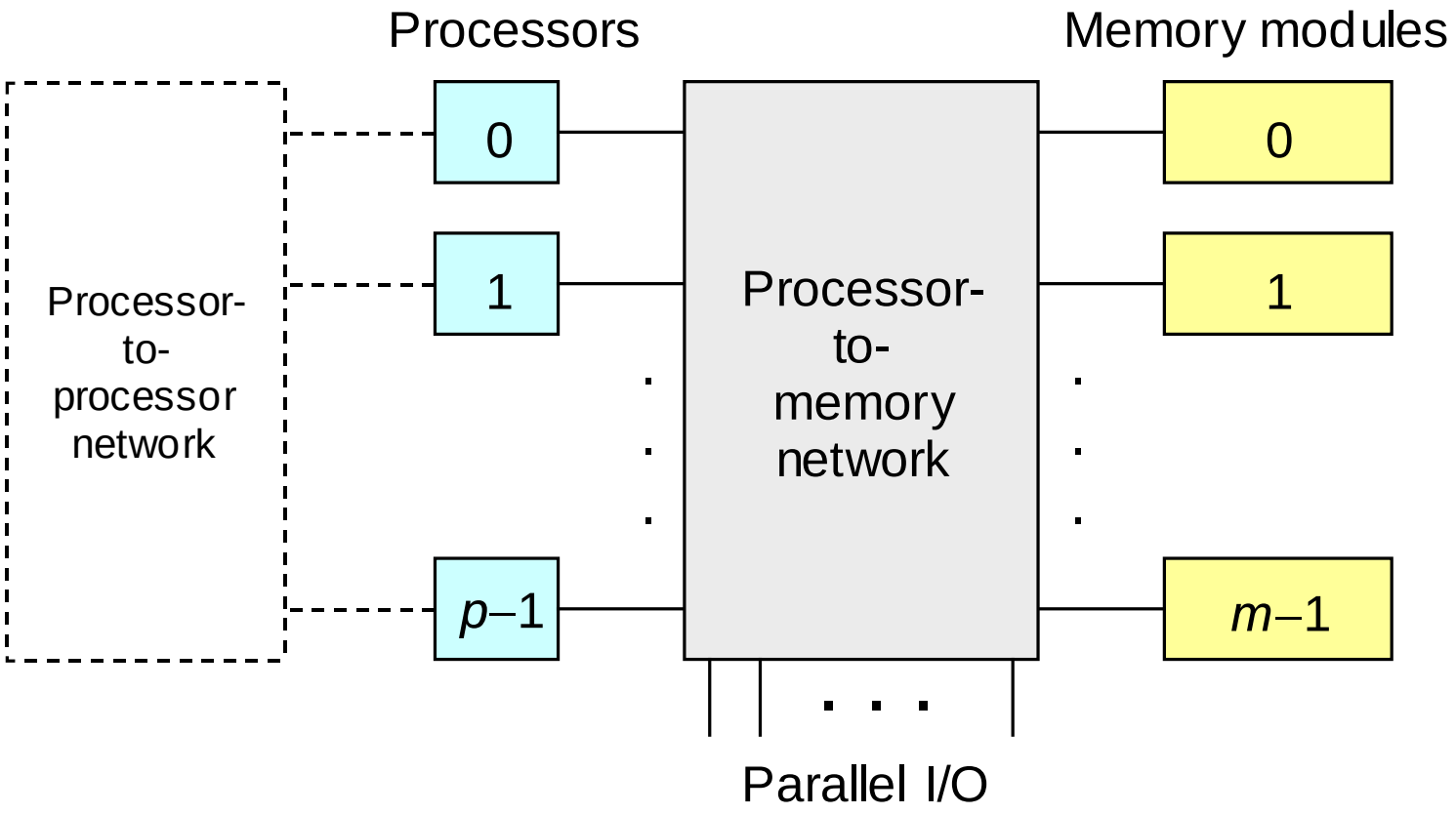
Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day



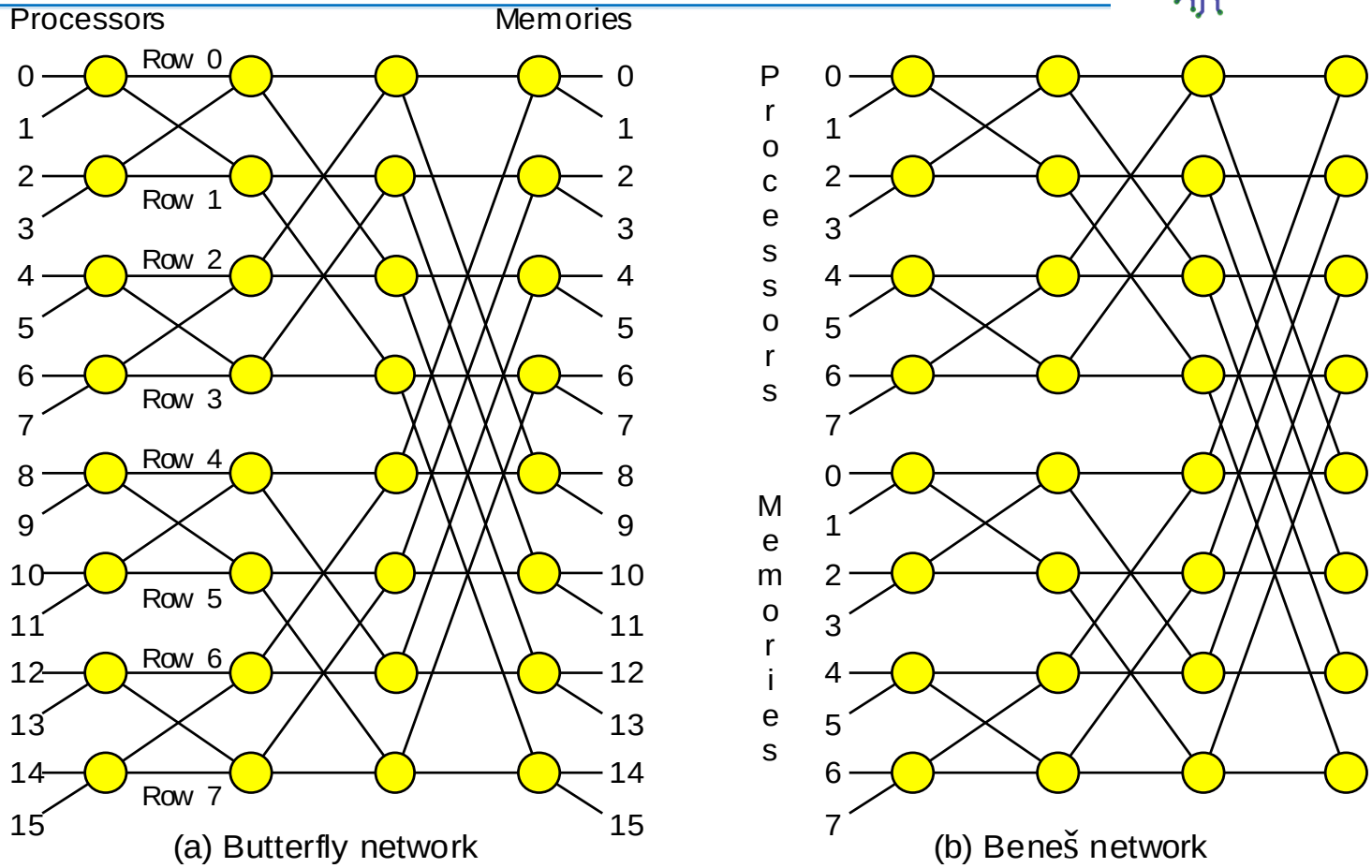
http://dilbert.com/strips/comic/1992-07-18/

Memoria partajată centralizată



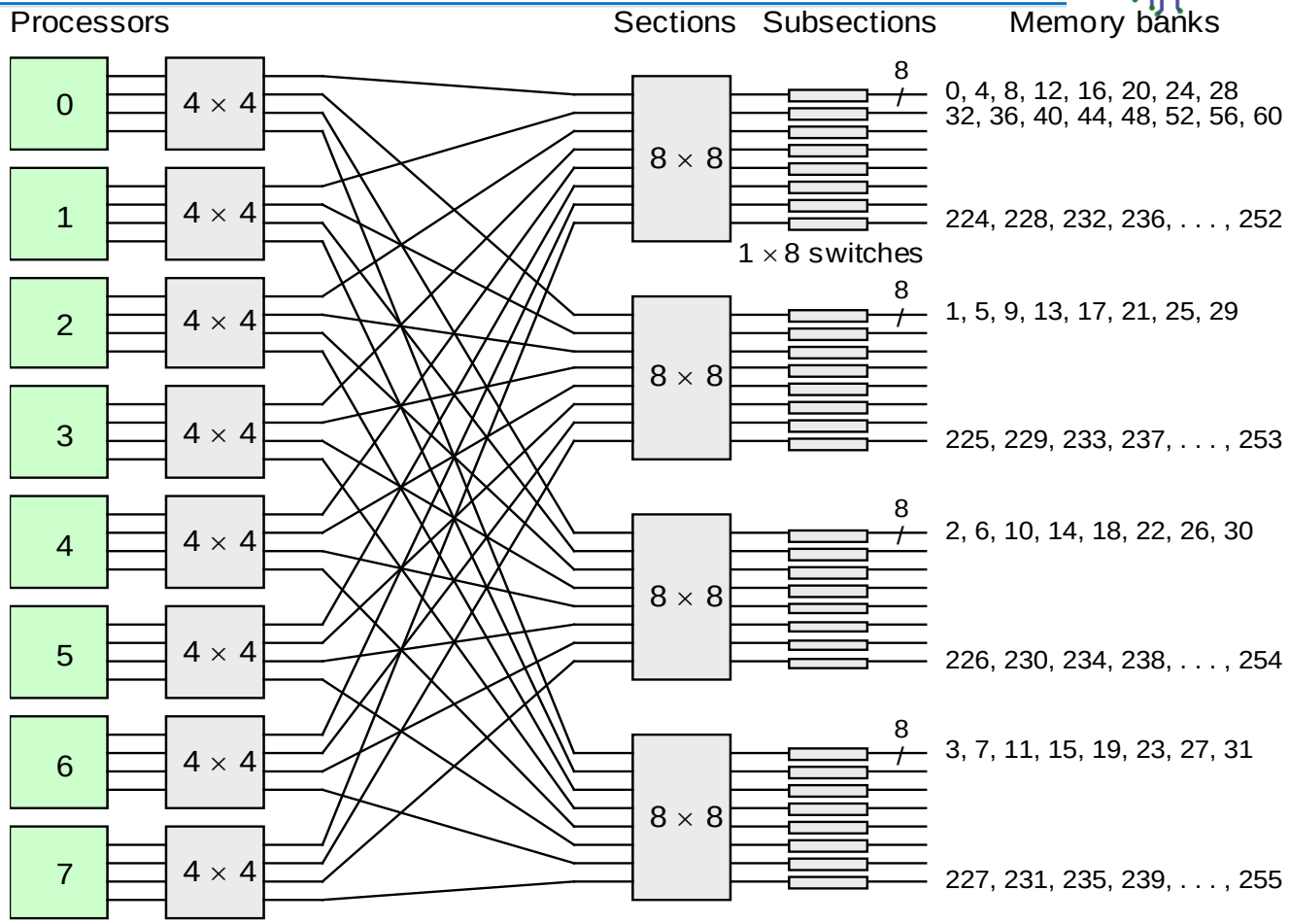
Structură multiprocesor cu memorie partajată

Rețele de interconectare Procesor-Memorie



Rețelele fluture și Beneš: exemple de rețele de interconectare procesor-memorie

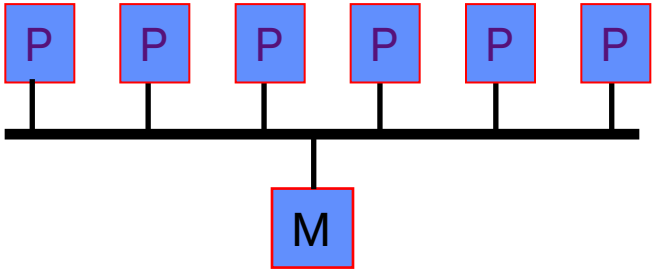
Rețele de interconectare Procesor-Memorie



Interconectarea a opt procesoare la 256 bancuri de memorie la Cray Y-MP (1988), un supercomputer cu procesoare vectoriale multiple

Consistența secvențială

Model de memorie



" A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =

întrețesere arbitrară cu păstrarea ordinei referințelor la memorie pentru programele secvențiale

Consistența secvențială

Task-uri secvențiale concurente: T1, T2
Variabile partajate: X, Y (inițial X = 0, Y = 10)

T1:
Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:
Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

Care sunt răspunsurile corecte pentru X' și Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)~~} ?

Dacă y este 11 atunci x nu poate fi 0

Consistența secvențială

Consistența secvențială impune mai multe contrângeri de ordonare de memorie ca și cele impuse de dependențele de memorie ale programelor uni-procesor (⟶)

Care sunt cele din exemplele noastre ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

Store (X'), R_2 *(X'= X)*

⟶ Cerințe adiționale SC

Poate un sistem cu cache și out-of-order execution să pună la dispoziție o imagine consistentă secvențial a memoriei?

Excluziunea mutuala si instructiuni blocante

Instrucțiuni blocante atomice read-modify-write

e.g., Test&Set, Fetch&Add, Swap

vs

Instrucțiuni atomice non-blocante read-modify-write

e.g., Compare&Swap,

Load-reserve/Store-conditional

vs

Protocoale bazate pe operații Load Store obișnuite

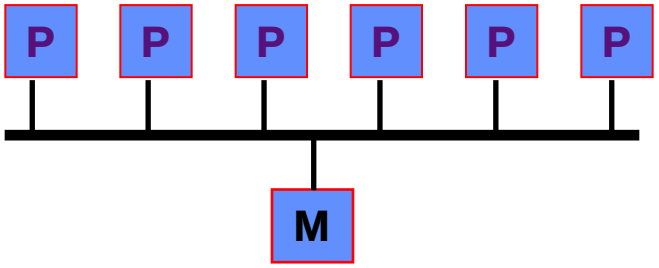
Performanța depinde de mai mulți factori:

degree of contention,

cache-uri,

out-of-order execution și Loads & Stores

Probleme în implementarea Consistenței Secvențiale



Implentarea CS este complicată de două probleme

- Capabilități de execuție *Out-of-order*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Cache-uri*

  Cache-urile pot preveni ca efectul unui store să fie văzut de alte procesoare

Bariere la memorie

Instrucțiuni care secvențializează accesul la memorie

Procesoarele cu arhitecturi de memorie slab-cuplate (ex. Permit reordonarea op. Loads & Stores la adrese diferite) trebuie să implementeze bariere la memorie (instrucțiuni) pentru a forța serializarea acceselor la memorie

Exemple de procesoare cu memorii slab-cuplate:

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

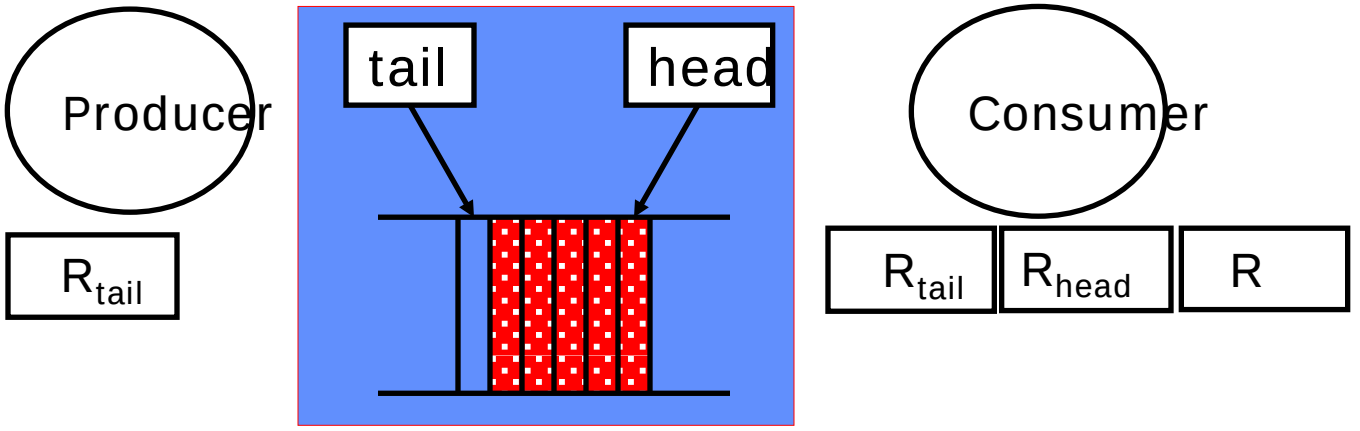
Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

Barierele la memorie (Memory fences) sunt operații costisitoare dar pot fi folosite numai atunci când sunt necesare

Folosirea barierelor la memorie



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
$Membar_{SS}$
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
$Membar_{LL}$
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

ensures that R is not loaded before x has been stored

Data-Race Free Programs

a.k.a. Properly Synchronized Programs

Embedded Systems Laboratory

Process 1

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

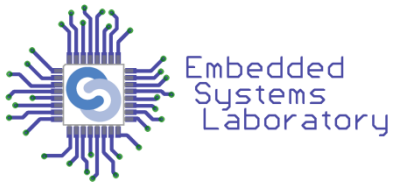
Variabilele de sincronizare (e.g. mutex) sunt separate de variabilele de date

Accesele la variabilele partajate de date sunt protejate în regiunile critice

$\Rightarrow$ *nu avem conflicte de date în afară de lock-uri*

În general, nu poate fi dovedit că un program este lipsit de conflicte de date.

Bariere pentru programele concurente



Process 1

```
...
Acquire(mutex);
membar;
    < critical section>
membar;
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
membar;
    < critical section>
membar;
Release(mutex);
```

- Modelul de memorie permite reordonarea instrucțiunilor de către compilator, cât timp această reordonare nu este făcută peste o barieră
- Procesorul nu trebuie să facă prefetch sau să speculeze peste o barieră

Excluziune mutuală folosind Load/Store

Embedded Systems Laboratory

Un protocol bazat pe două variabile partajate c1 și c2.
Inițial c1 și c2 sunt 0 *(not busy)*

Process 1

```
    ...
    c1=1;
L:  if c2=1 then go to L
      < critical section>
    c1=0;
```

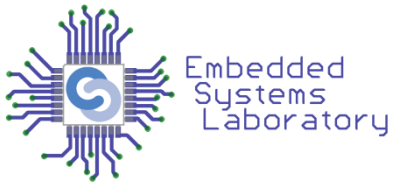
Process 2

```
    ...
    c2=1;
L:  if c1=1 then go to L
      < critical section>
    c2=0;
```

Care este problema?

Deadlock!

Excludere mutuală: *a doua încercare*



Pentru a evita *deadlock*, lasă un proces să renunțe la lock (i.e. Process 1 setează c1 la 0) cât timp așteaptă.

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock nu mai este posibil dar este o posibilitate redusă de *livelock*.

- Un proces nenorocos poate să nu intre niciodată în secțiunea critică $\Rightarrow$ *starvation*

Un protocol pentru excludere mutuală

T. Dekker, 1966

Protocol bazat pe trei variabile partajate c1, c2 și turn.
Inițial, c1 și c2 sunt 0 *(not busy)*

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* asigură că doar procesul i poate să aștepte
- variabilele c1 și c2 asigură *excluderea mutuală*

Soluția pentr n procese a fost dată de Dijkstra și este puțin mai complicată!

Analiza algoritmului lui Dekker

Embedded Systems Laboratory

Scenario 1

```
...                    Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
                    then go to L
   < critical section>
c1=0;
```

```
...                    Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
                    then go to L
   < critical section>
c2=0;
```

Scenario 2

```
...                    Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
                    then go to L
   < critical section>
c1=0;
```

```
...                    Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
                    then go to L
   < critical section>
c2=0;
```

N-process Mutual Exclusion

Lamport's Bakery Algorithm

Embedded Systems Laboratory

Process i

Initially num[j] = 0, for all j

Entry Code

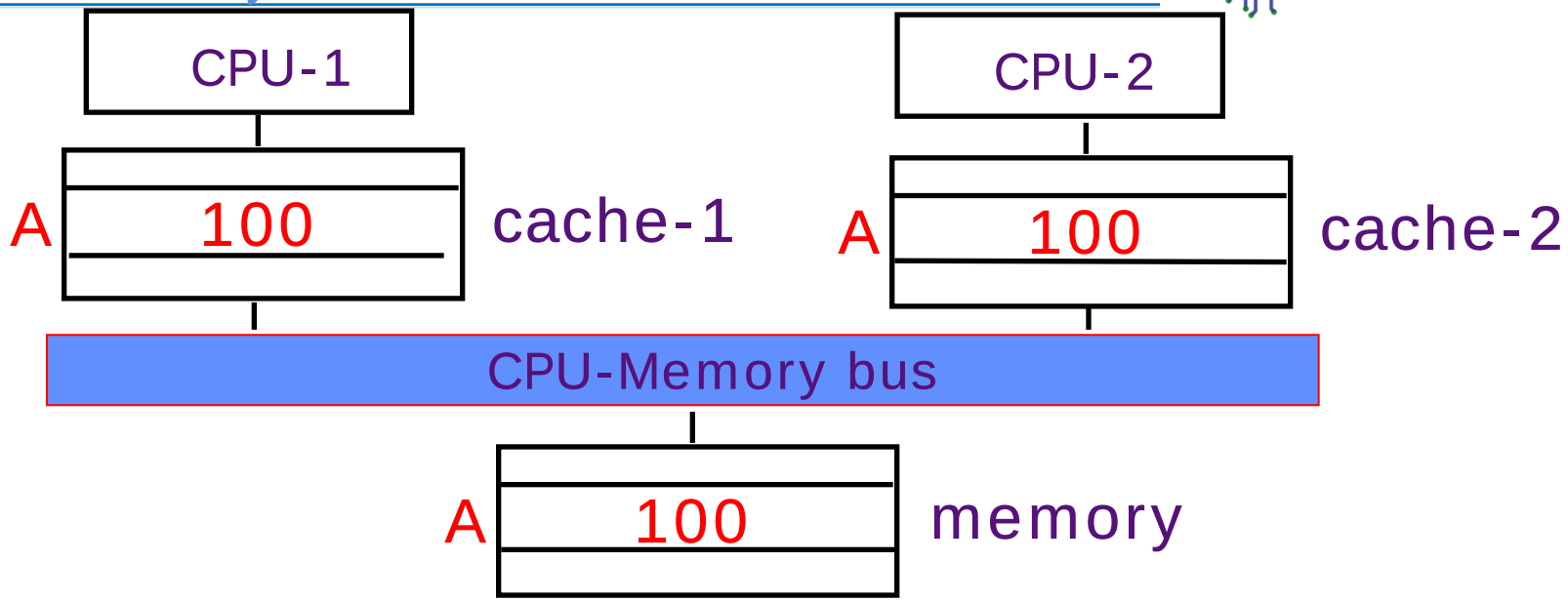
```
choosing[i] = 1;
num[i] = max(num[0], …, num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++)  {
    while( choosing[j] );
    while( num[j] &&
           ( ( num[j] < num[i] ) ||
             ( num[j] == num[i] &&  j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```

Consistența memoriei la SMP-uri



Presupunem că CPU-1 actualizează A la 200.

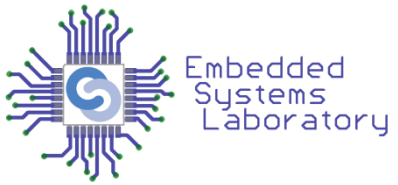
write-back: *memoria si cache*-2 au valori vechi

write-through: cache-2 are valoarea veche

Contează aceste valori neactualizate?

Cum este văzută memoria partajată de software?

Write-back Caches & SC



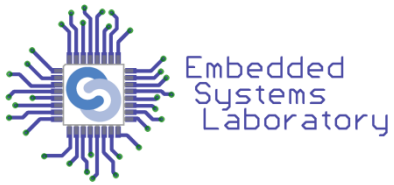
	prog T1	cache-1	memory	cache-2	prog T2

• T1 is executed	ST X, 1
	ST Y,11
		X= 1
		Y=11
			X = 0
			Y =10
			X'=
			Y'=
				Y =
				Y'=
				X =
				X'=
					LD Y, R1
					ST Y', R1
					LD X, R2
					ST X',R2

• cache-1 writes back Y		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = Y'= X = X'=	
• T2 executed		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-2 writes back X' & Y'		X= 1 Y=11	X = 1 Y =11 X'= 0 Y'=11	Y =11 Y'=11 X = 0 X'= 0	

inconsistent

Write-through Caches & SC



prog T1	cache-1	memory	cache-2	prog T2

ST X, 1
ST Y,11
	X= 0
	Y=10
		X = 0
		Y =10
		X'=
		Y'=
			Y =
			Y'=
			X = 0
			X'=
				LD Y, R1
				ST Y', R1
				LD X, R2
				ST X',R2

- T1 executed

cache-1	memory	cache-2
X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = Y'= X = 0 X'=

- T2 executed

cache-1	memory	cache-2
X= 1 Y=11	X = 1 Y =11 X'= 0 Y'=11	Y = 11 Y'= 11 X = 0 X'= 0

Cache-urile write-through nu mențin nici consistența secvențială

Menținerea consistenței secvențiale (CS)

CS este suficientă pentru programe tip producer-consumer și cu excluziune mutuală (e.g., Dekker)

Copiile multiple ale unei locații în diferite Cache-uri pot cauza degradarea CS.

Este nevoie de suport hardware pentru

- doar un singur procesor la un moment dat are permisiune de write la o locație de memorie
- nici un procesor nu poate să încarce o copie a vechii locații după o scriere

$\Rightarrow$ Protocoale de coerență a cache-ului

Protocoale de menținere a coerenței cache pentru CS

write request:

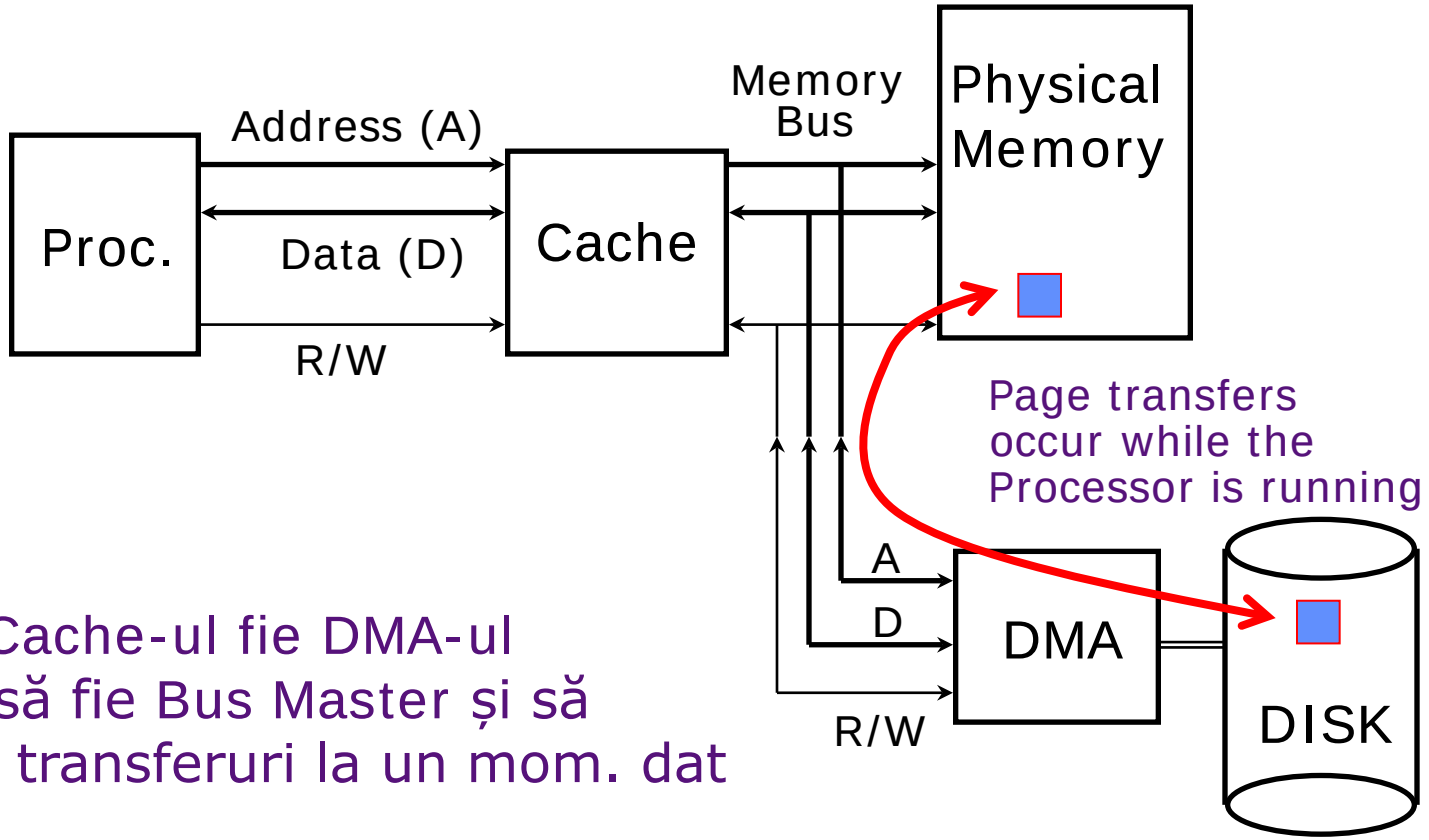
adresa este *invalidată* (*actualizată)* în toate cache-urile înainte (după) o operație de write

read request:

dacă o este gasită o copie ”murdară” într-un cache, se efectuează un write-back înainte de citirea memoriei

Ne vom concentra pe protocoale de Invalidare și nu pe protocoale Update

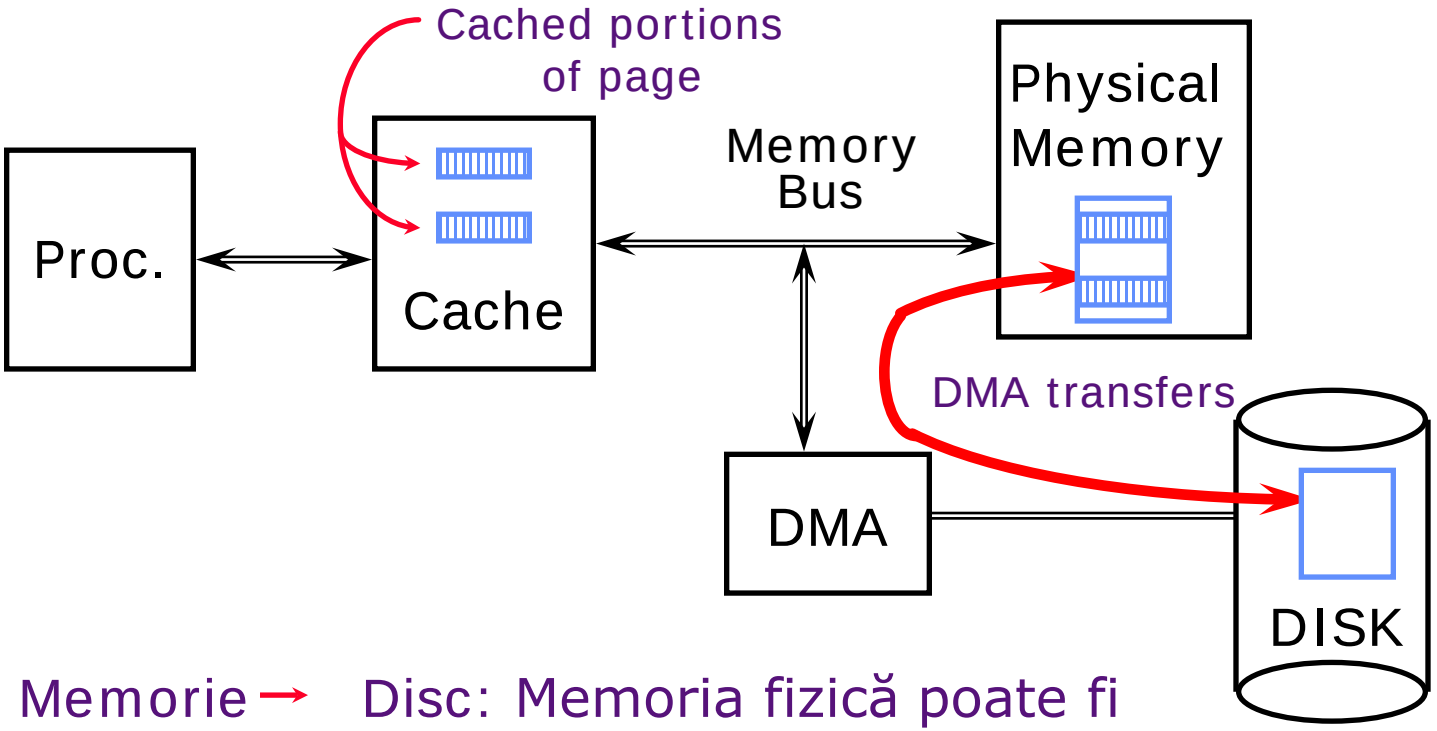
Warmup: Parallel I/O



Fie Cache-ul fie DMA-ul pot să fie Bus Master și să facă transferuri la un mom. dat

(DMA vine de la Direct Memory Access)

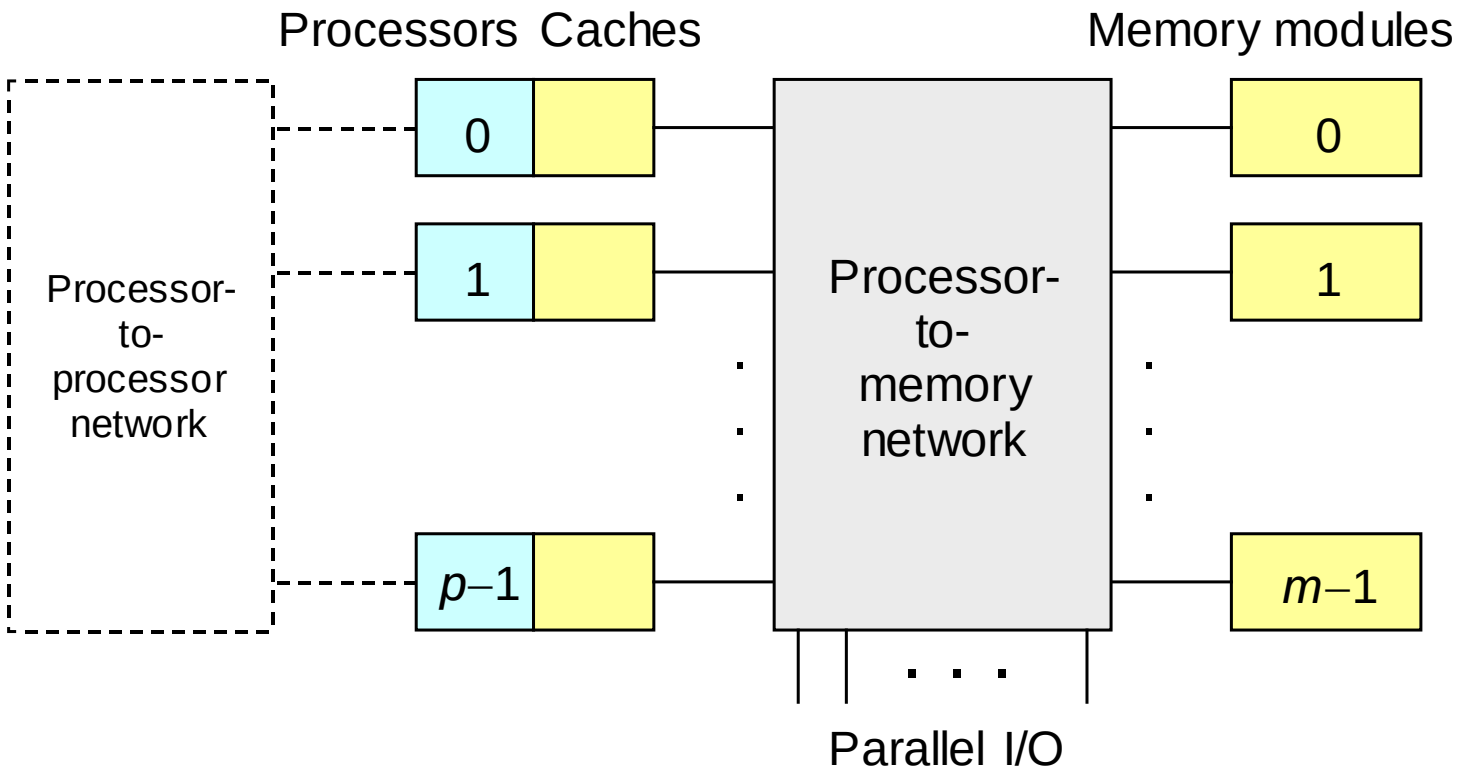
Probleme cu Parallel I/O



Memorie → Disc: Memoria fizică poate fi neactualizată dacă copia din Cache este ”murdară”

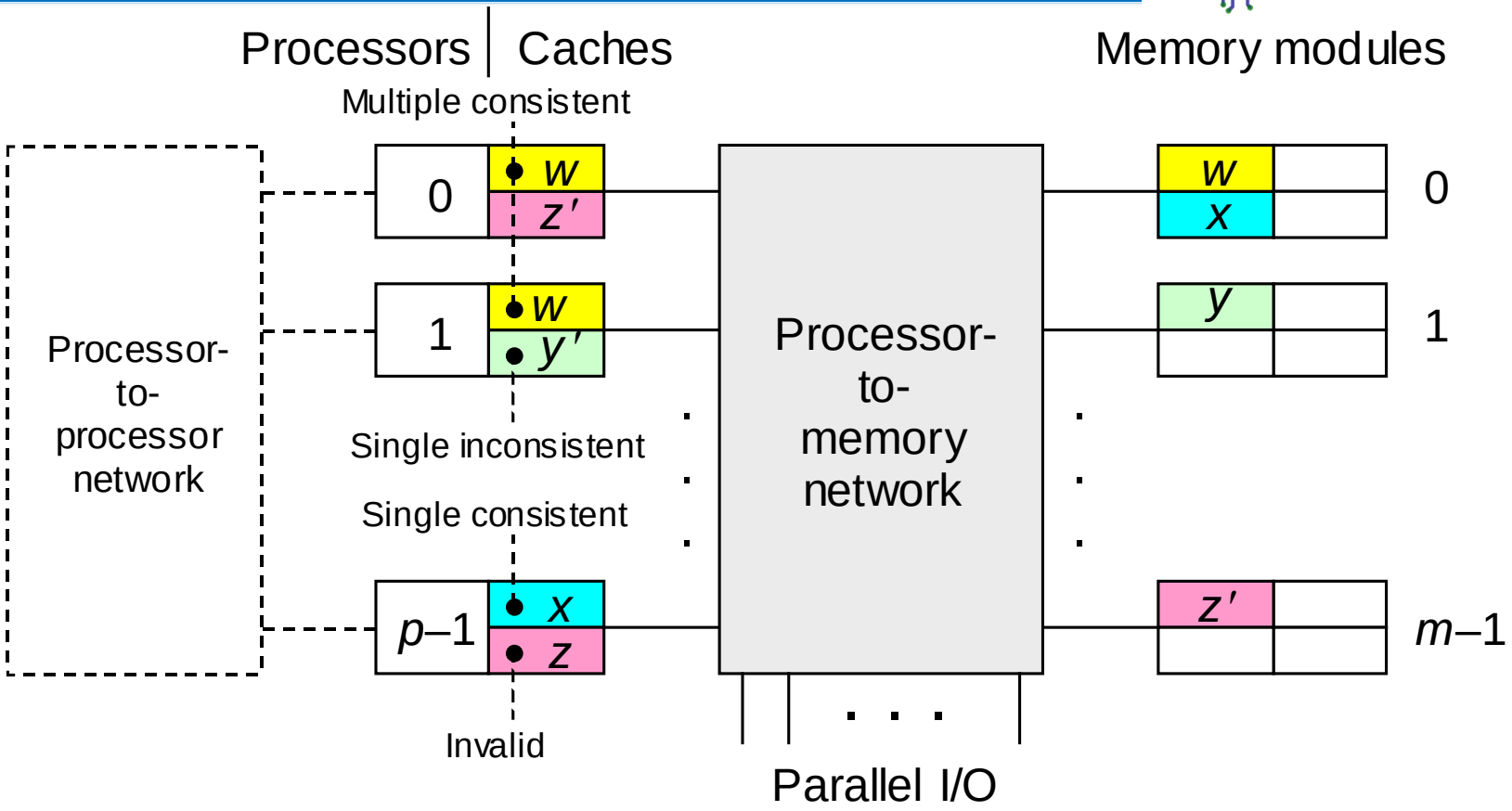
Disc → Memorie: Cache-ul poate să conțină date vechi și să nu vadă write-urile la memorie

Cache-uri multiple și coerența cache-urilor



Memoriile cache dedicate fiecărui procesor reduc traficul la memoria procipală (prin rețeaua de interconectare) dar introduc o serie de probleme de consistență.

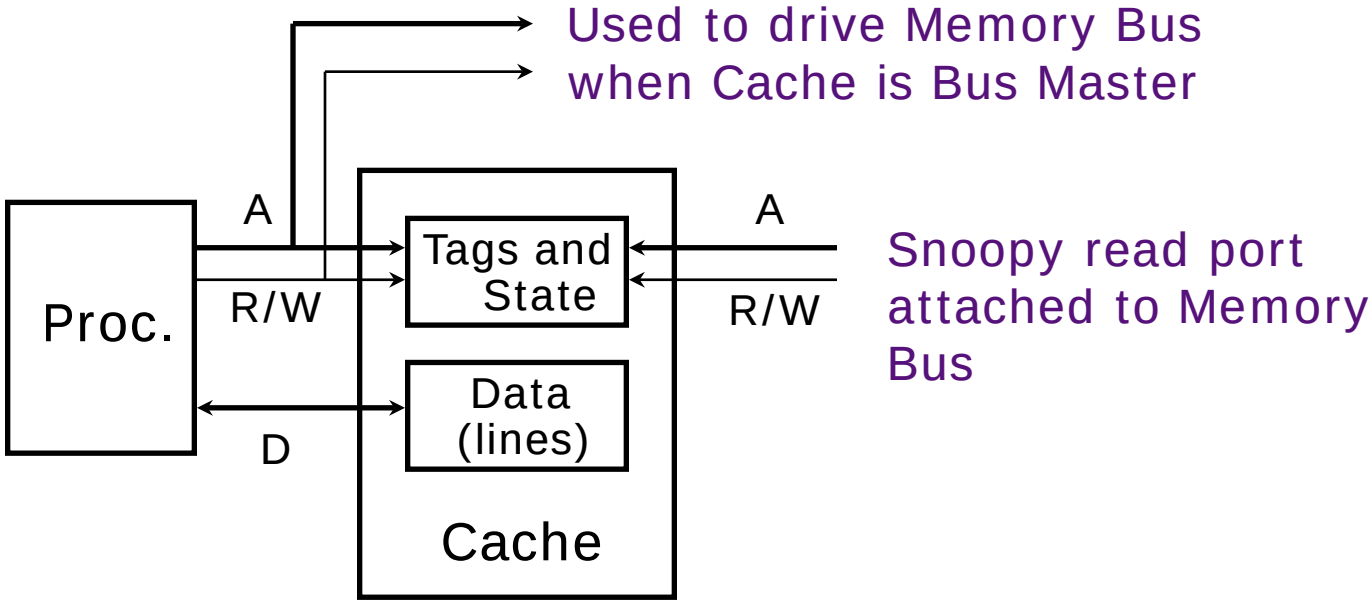
Starea copiilor de date



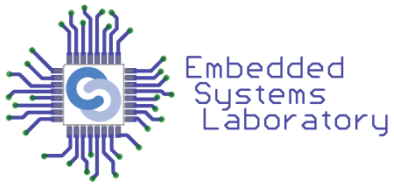
Diferite tipuri de blocuri de date din cache pentru un procesor paralel cu memorie principală centralizată și cache-uri locale pentru fiecare procesor

Snoopy Cache *Goodman 1983*

- Idee: Să avem un cache care spionează (snoop upon) transferurile DMA, și atunci “do the right thing”
- Etichetele snoopy cache sunt dual-port

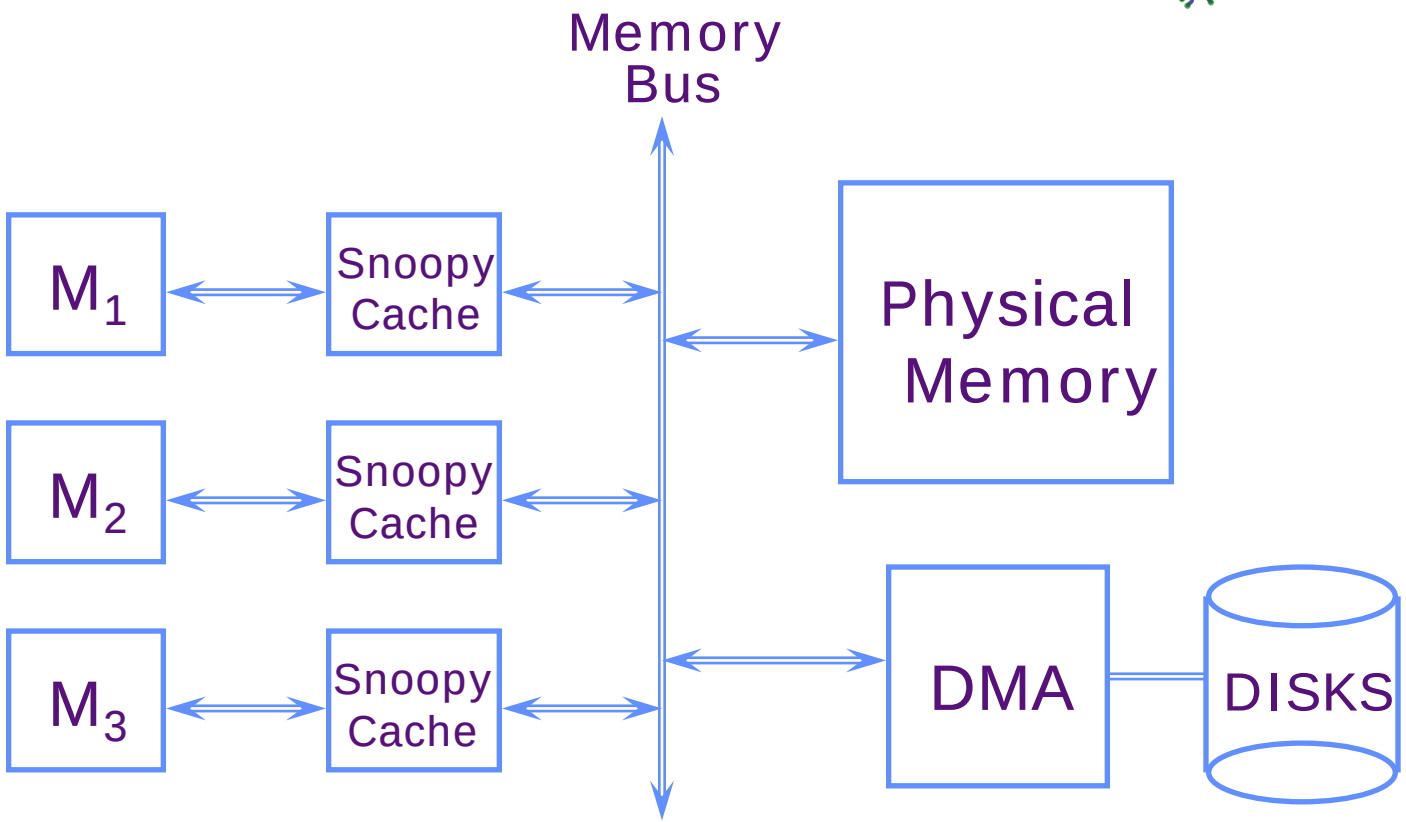


Acțiunile Snoopy Cache pentru DMA



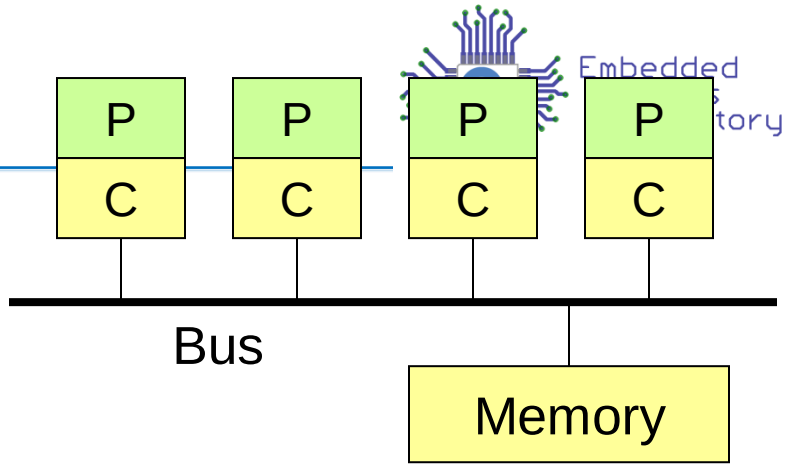
Observed Bus Cycle	Cache State	Cache Action
DMA Read Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
DMA Write Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

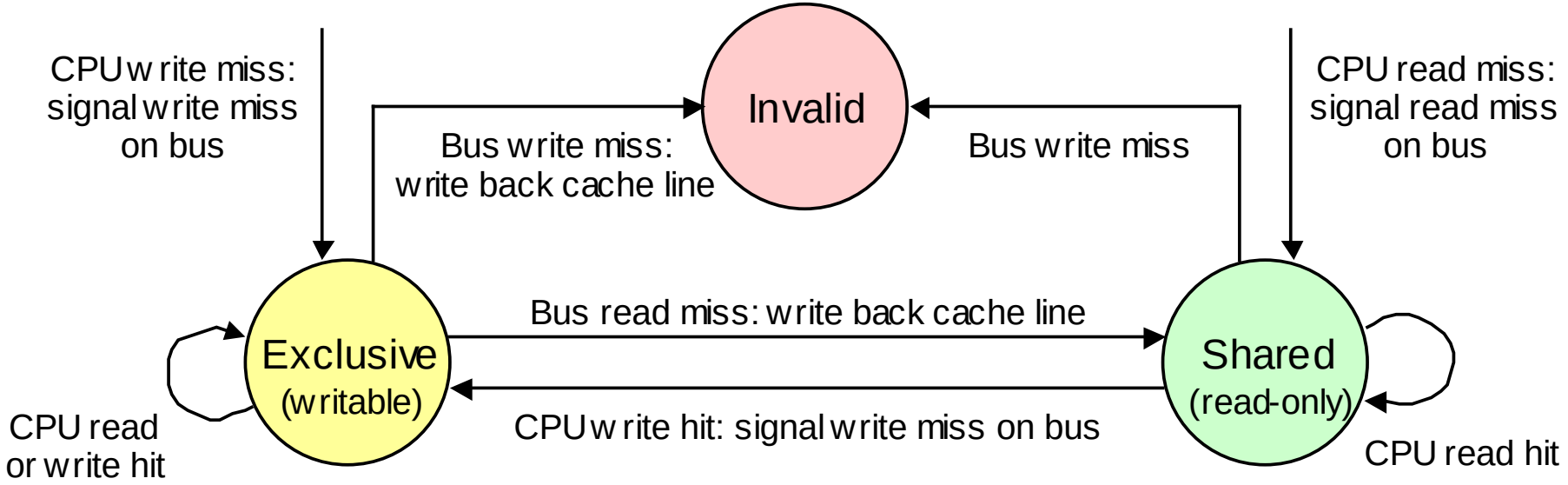
Shared Memory Multiprocessor



Folosește mecanismul snoopy pentru a păstra consistența memoriei pentru toate procesoarele

Protocol snoopy de coerență cache





Automat FSM pentru un protocol de coerență cache ce folosește cache write-back

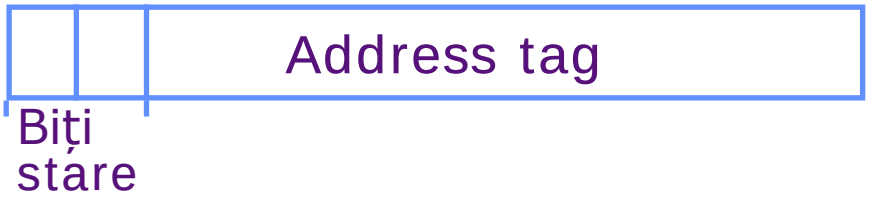
Diagrama de tranziții pentru Cache

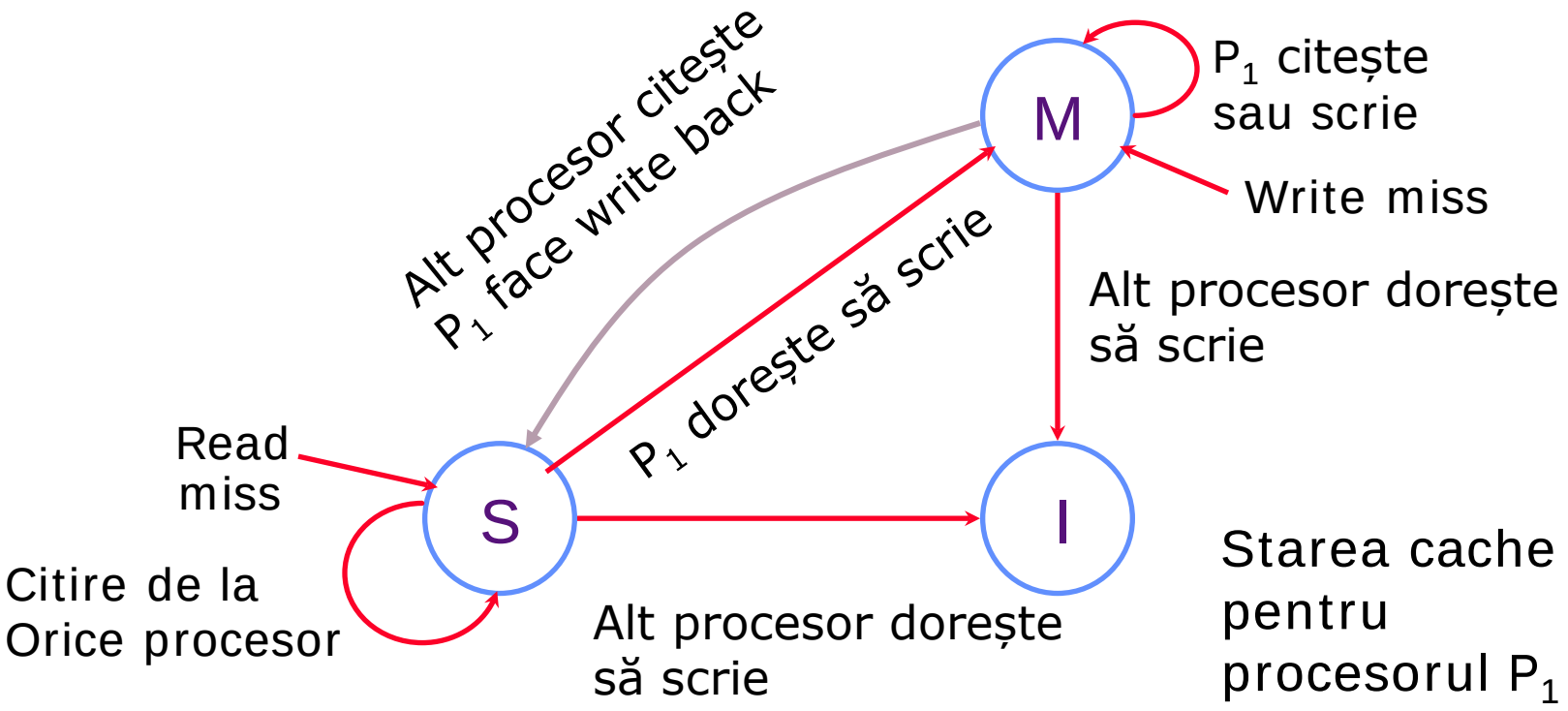
Protocolul MSI

Embedded Systems Laboratory

Fiecare linie din cache are tag

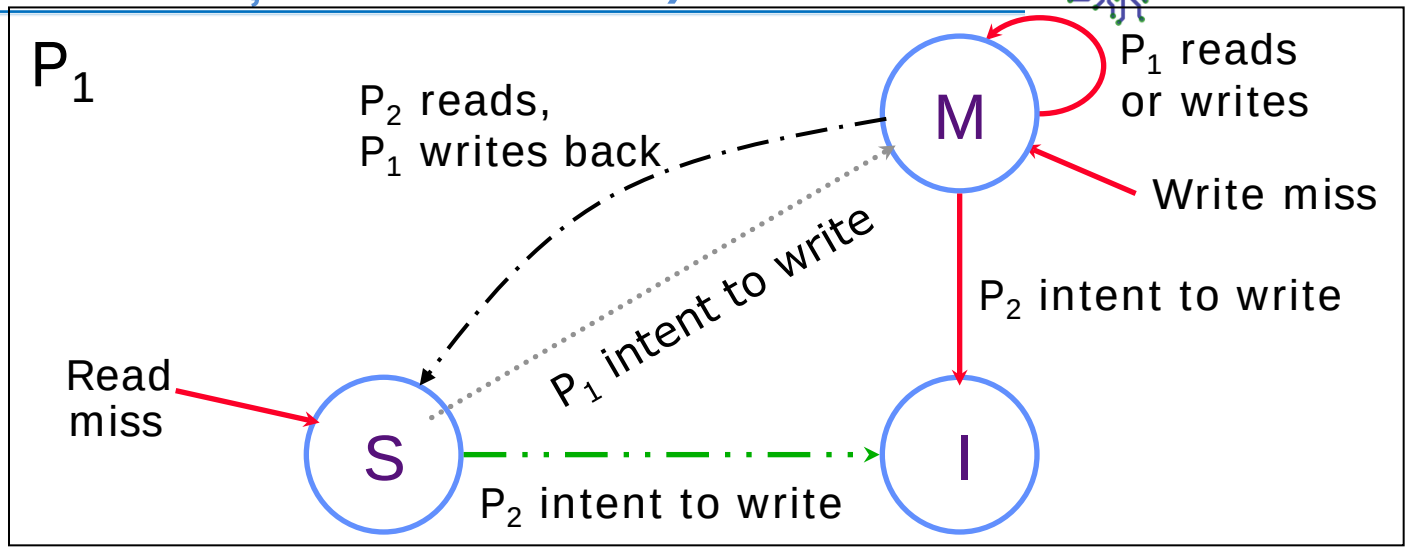
M: Modified
S: Shared
I: Invalid

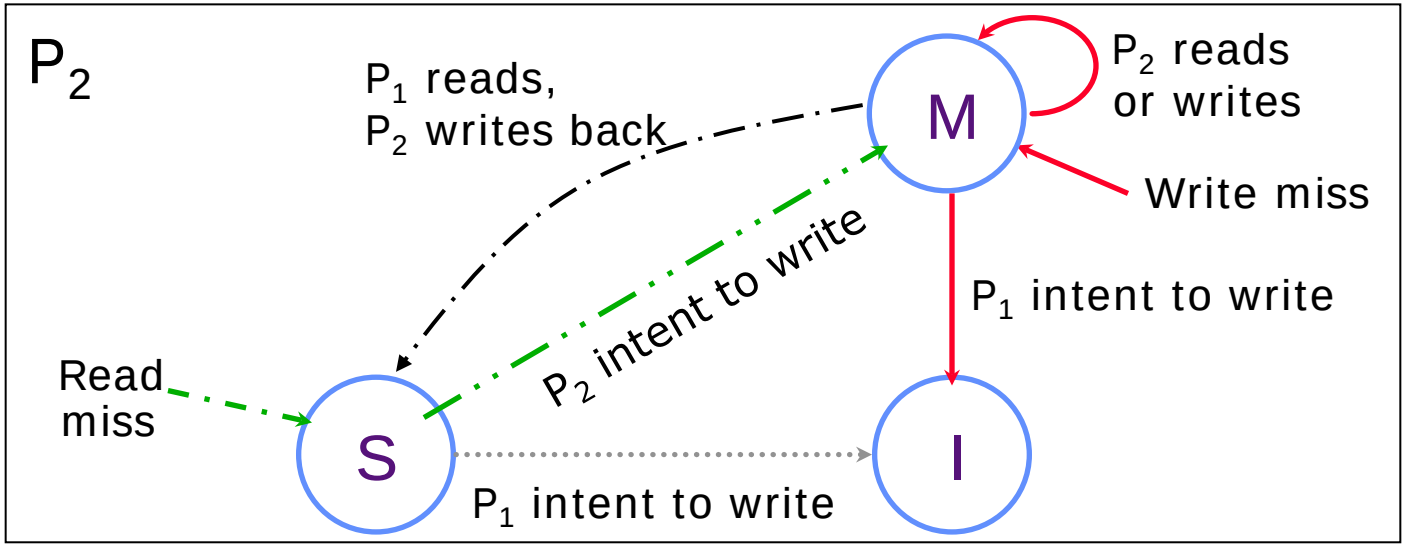




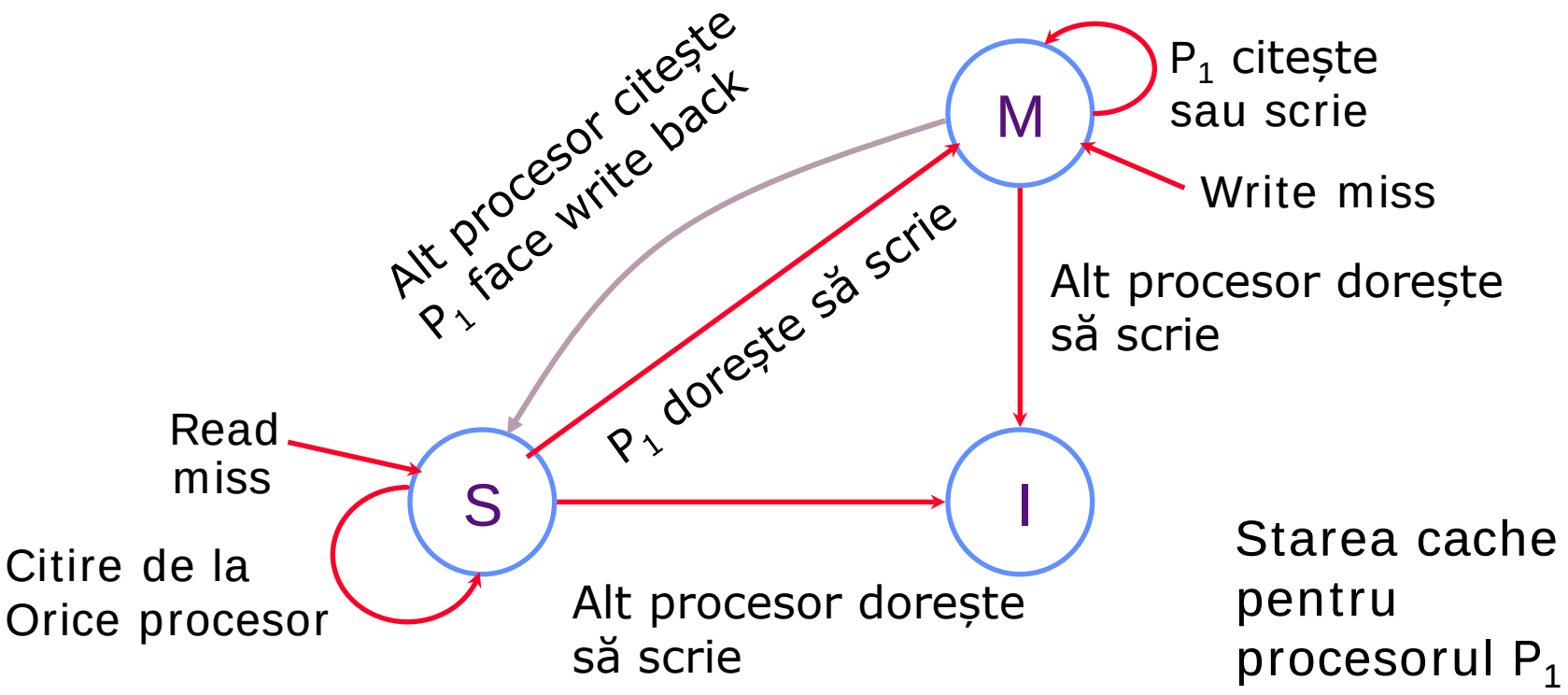
Exemplu cu două procesoare
(Citesc și scriu aceeași linie din cache)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





Observație

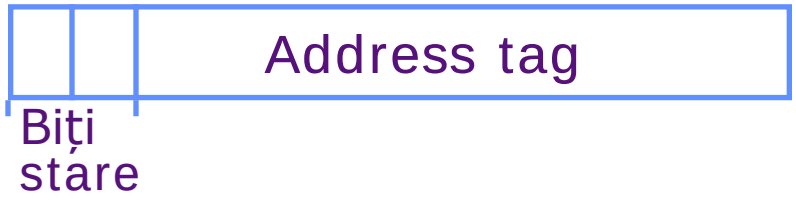


- Dacă o linie este în starea M atunci nici un alt cache nu poate să aibă o copie a linei!
  - Memoria rămâne coerentă, nu pot exista copii diferite

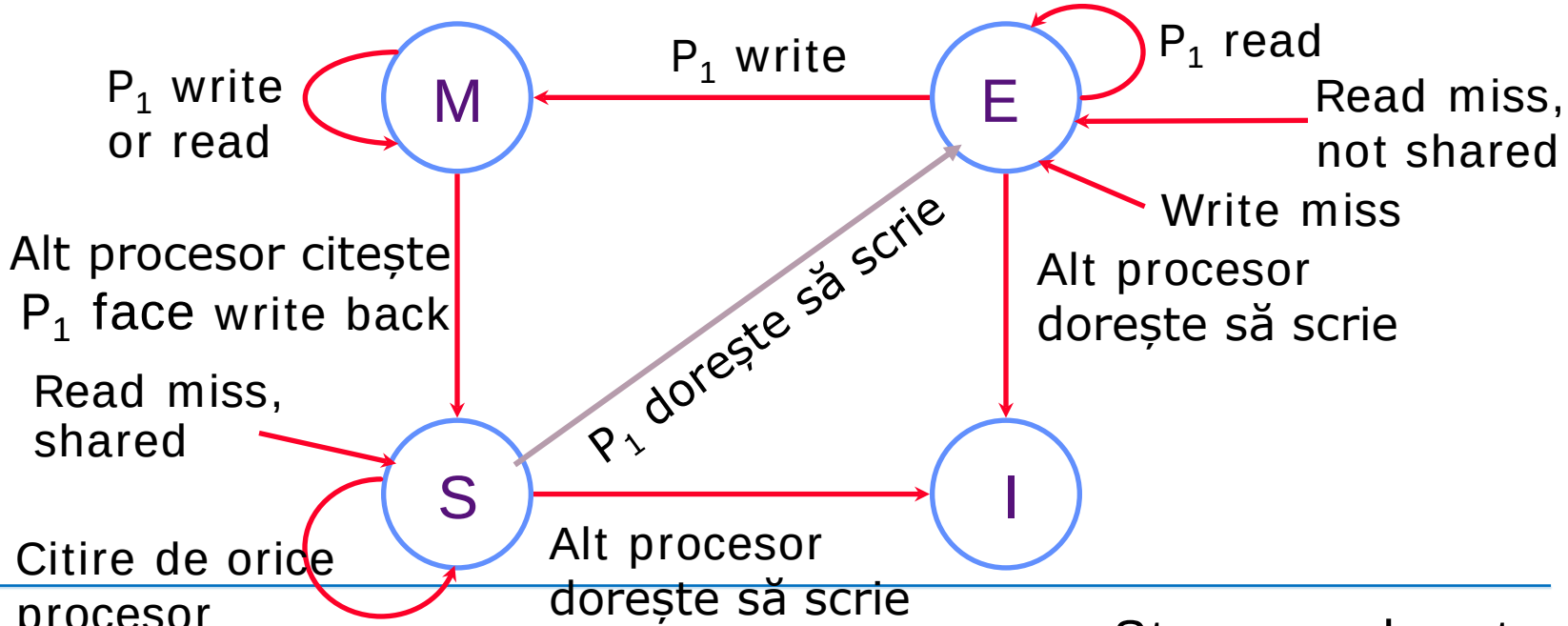
MESI: Protocol MSI îmbunătățit
performanțe mărite pentru date locale

Embedded Systems Laboratory

Fiecare linie are un tag

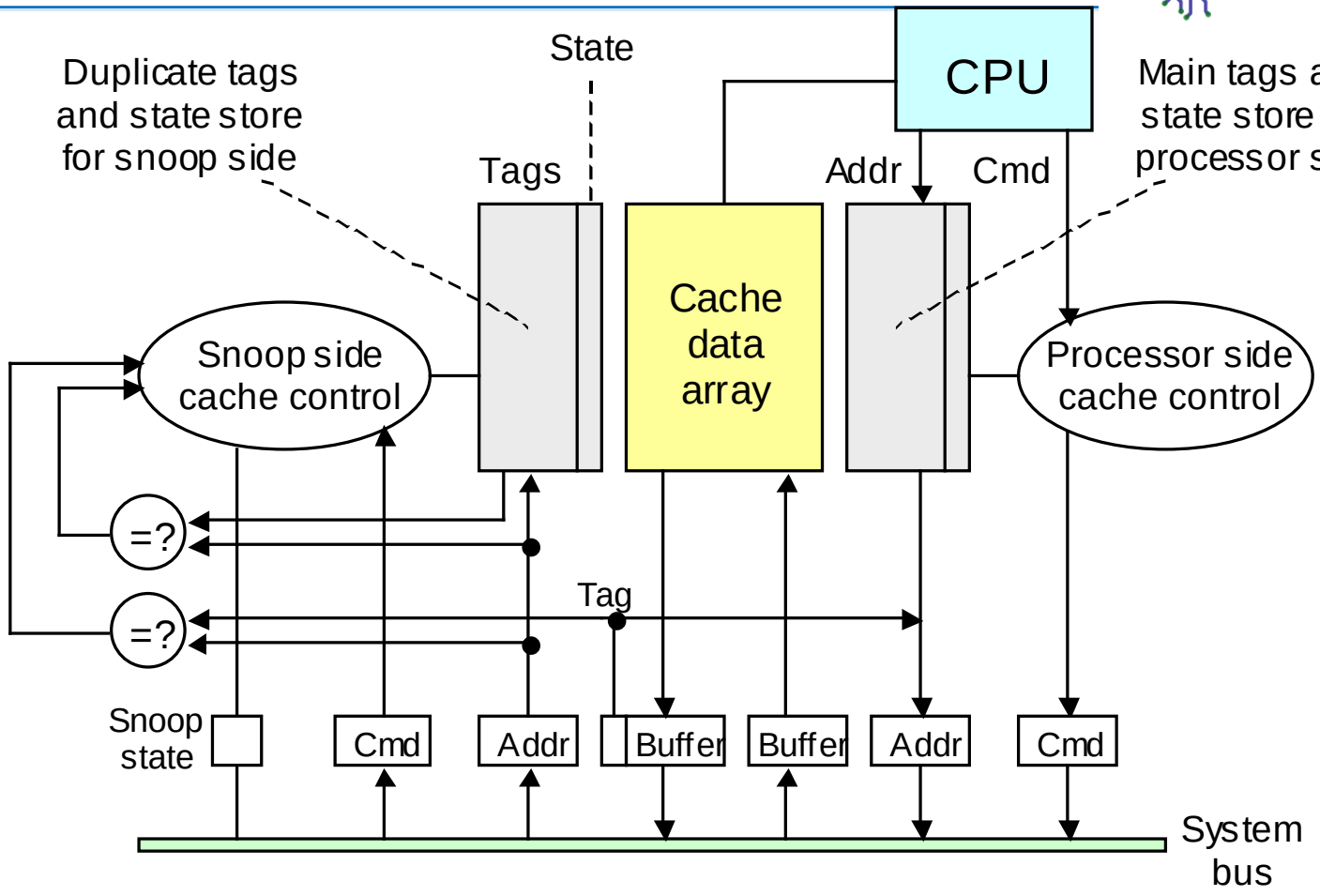


M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



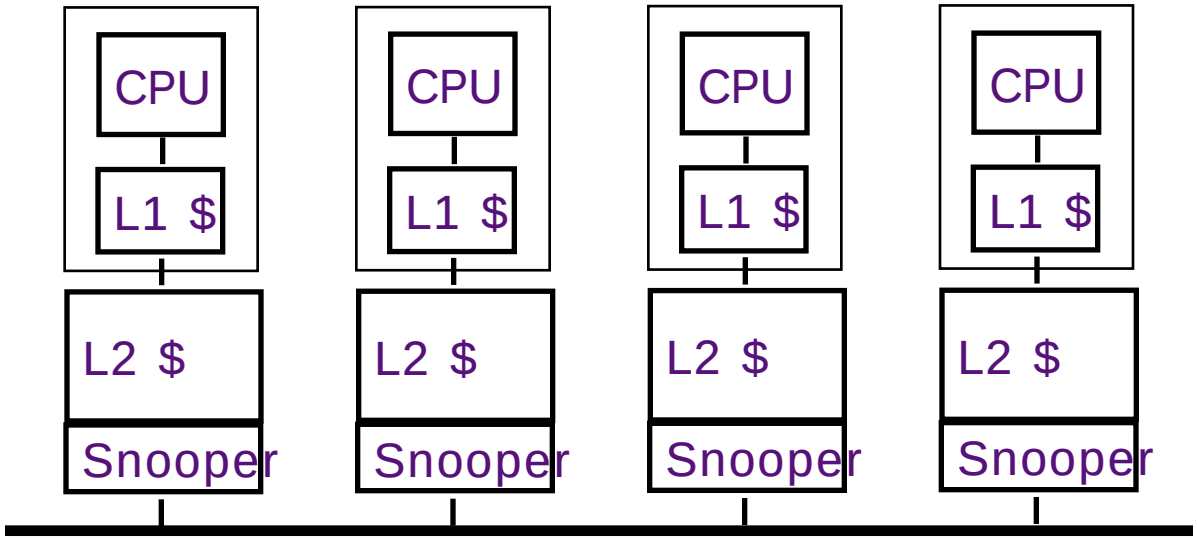
Starea cache pt. procesorul P1

Implementarea Algoritmului Snoopy Cache



Structura principală a unui algoritm snoopy de coerență cache

Snoop optimizat cu cache-uri Level-2

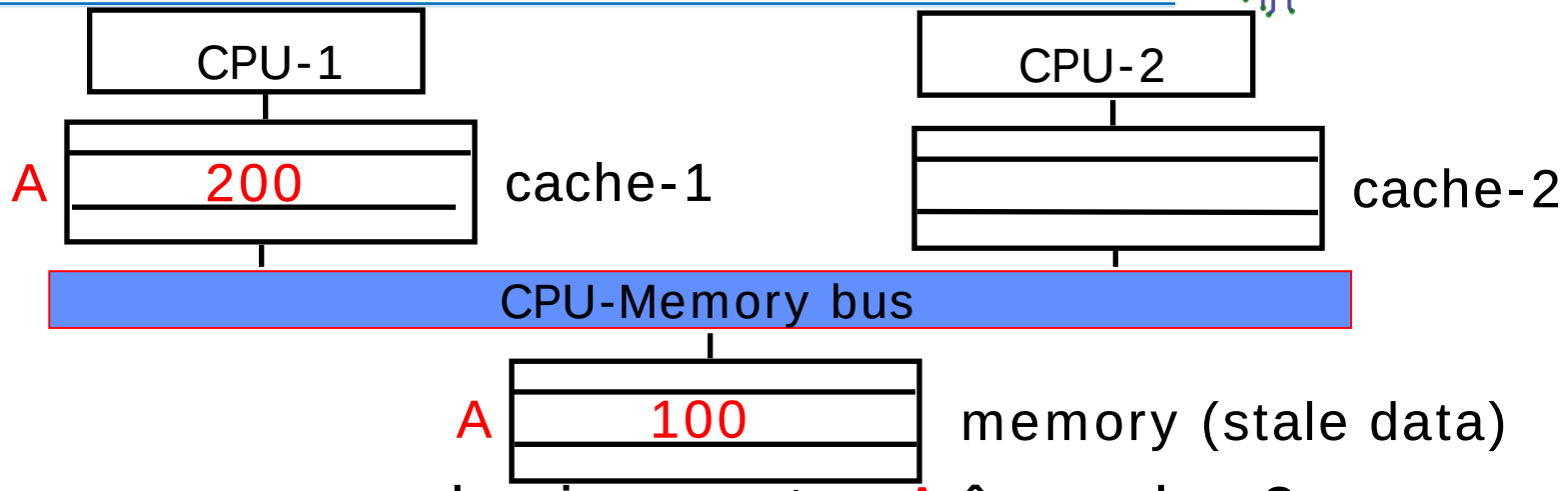


- Procesoarele au de obice cache pe două niveluri
  - mic L1, mare L2 (de obicei ambele on chip acum)
- *Proprietatea de incluziune:* intrările din L1 trebuie să fie în L2

  invalidare în L2 ⇒ invalidare în L1
- Snooping în L2 nu afectează lățimea de bandă CPU-L1

Ce probleme pot să apară?

Intervenție



Când avem un read-miss pentru A în cache-2, se emite pe bus un read request pentru A

- Cache-1 trebuie să facă vizibilă și să își schimbe starea în ”shared”
- Memoria poate să răspundă și ea la cerere!

Știe memoria că are date vechi?

Cache-1 trebuie să intervină prin controllerul de memorie pentru a da datele corecte pentru cache-2

False Sharing

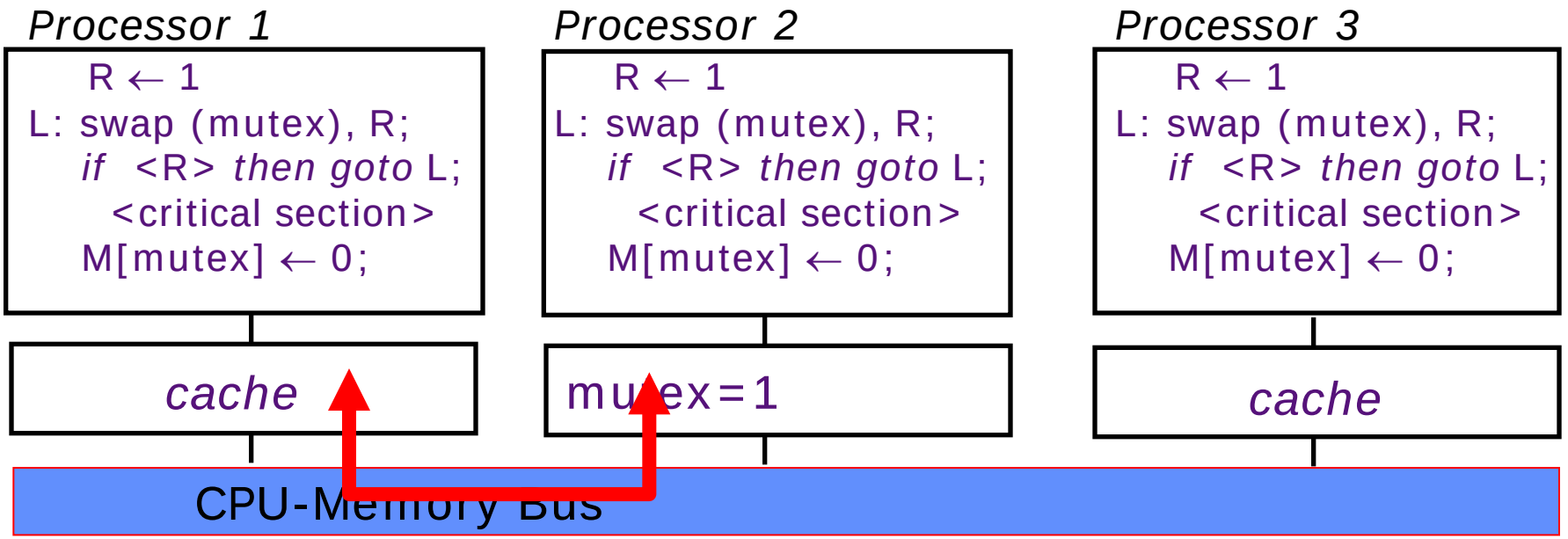
Un bloc cache conține mai mult de un cuvânt de date

Coerența cache-ului este făcută la nivel e bloc și nu la nivel de cuvânt

Presupunem că M_1 scrie $word_i$ și M_2 scrie $word_k$ și ambele cuvinte au aceeași adresă de bloc.

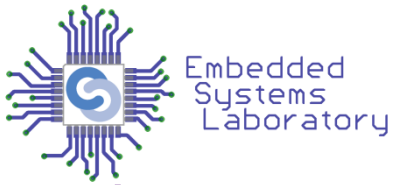
Ce se poate întâmpla?

Sincronizarea și cache-urile:
Probleme de performanță



Protocoalele de coerență a cache-ului vor face mutex-ul să facă *ping-pong* între cache-urile lui P1 și P2.
Acest fenomen poate fi redus prin citirea locației inițiale a mutex-ului (non-atomic) și execuția unui schimb doar dacă acesta are valoarea zero.

Performanța și traficul pe magistrale



În general, o instrucțiune *read-modify-write* necesită două operații pe magistrală, dacă nu avem alte operații la memorie de către alte procesoare

Într-un scenariu multiprocesor, accesul tuturor celorlalte procesoare la magistrală trebuie să fie blocat pe durata execuției operației atomice de read-modify-write

$\Rightarrow$ costisitor pentru magistrale simple

ISA-urile moderne folosesc

load-reserve

store-conditional

Load-reserve & Store-conditional

Registre speciale pentru stocarea flag-urilor de reserve, adresa și rezultatul store-conditional

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional (a), R:
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

Dacă un alt procesor vede o tranzacție de store la adresa din registrul de rezervare, bitul de rezervare este setat la 0

- Mai multe procesoare pot rezerva 'a' simultan
- Aceste instrucțiuni sunt similare cu load și store obișnuite, din pct de vedere al traficului pe bus

Performanță:
Load-reserve & Store-conditional

Embedded Systems Laboratory

Numărul total de tranzacții pe bus nu este neapărat redus, dar spargerea unei instrucțiuni atomice în load-reserve & store-conditional:

- crește utilizarea magistralei, mai ales pentru magistralele care efectuează tranzacții în mai multe etape

- *reduce efectul ping-pong pentru cache* deoarece procesoarele care încearcă să obțină un semafor nu trebuie să facă un store de fiecare dată

Lățimea de bandă limitează performanțele

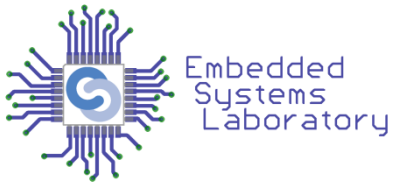
Embedded Systems Laboratory

Avem un sistem multiprocesor cu memorie partajată construit în jurul unui singur bus cu lățimea de bandă de *x* GB/s. Cuvintele de instrucțiuni și date au fiecare 4B lățime, fiecare instrucțiune necesită accesul în medie la 1.4 cuvinte din memorie (inclusiv la instrucțiunea însăși). Rata combinată de hit a cache-urilor este de 98%. Calculați limita sperioară a performanței sistemului multiprocesor în GIPS. Liniile de adresă sunt separate și nu afectează lățimea de bandă de date.

Soluție

Execuția unei instrucțiuni implică un transfer de $1.4 \times 0.02 \times 4 = 0.112$ B. Astfel, limita absolută a perfomanței este de $x/0.112 = 8.93x$ GIPS. Dacă presupunem o lățime a busului de 32 de biți, că nici un ciclu pe bus nu se irosește și o frecvență de ceas pe bus de *y* GHz, limita superioară a perfomanței devine $286y$ GIPS. Magistralele operează la frecvențe de 0.1 - 1 GHz. Prin urmare, o performanță apropiată de 1 TIPS (chiar și ¼ TIPS) este peste capabilitățile acestui tip de arhitectură.

Acknowledgements



- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252