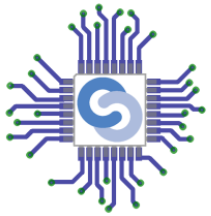
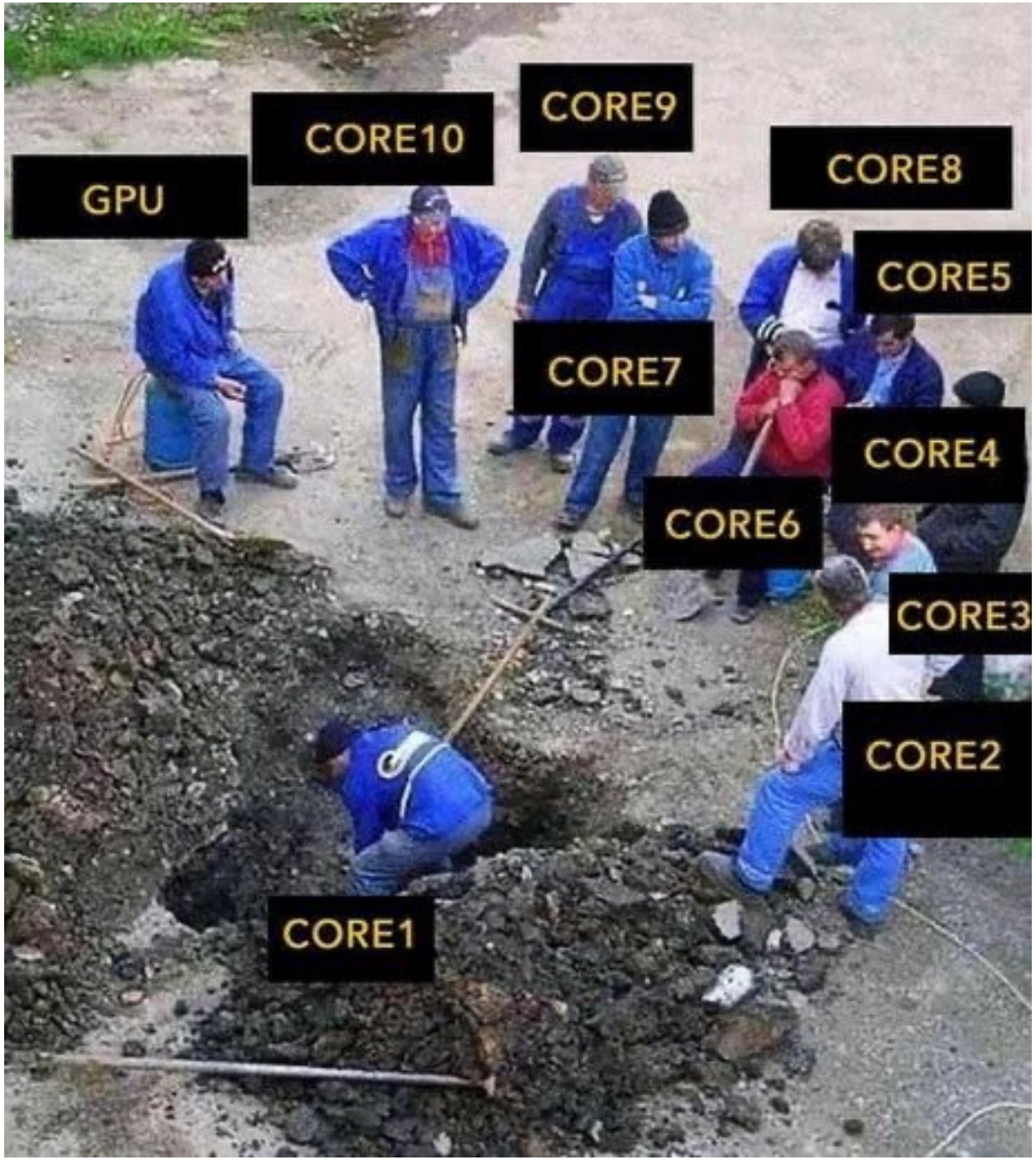


Calculatoare Numerice (2)

- Cursul 10 –

Multiprocesoare

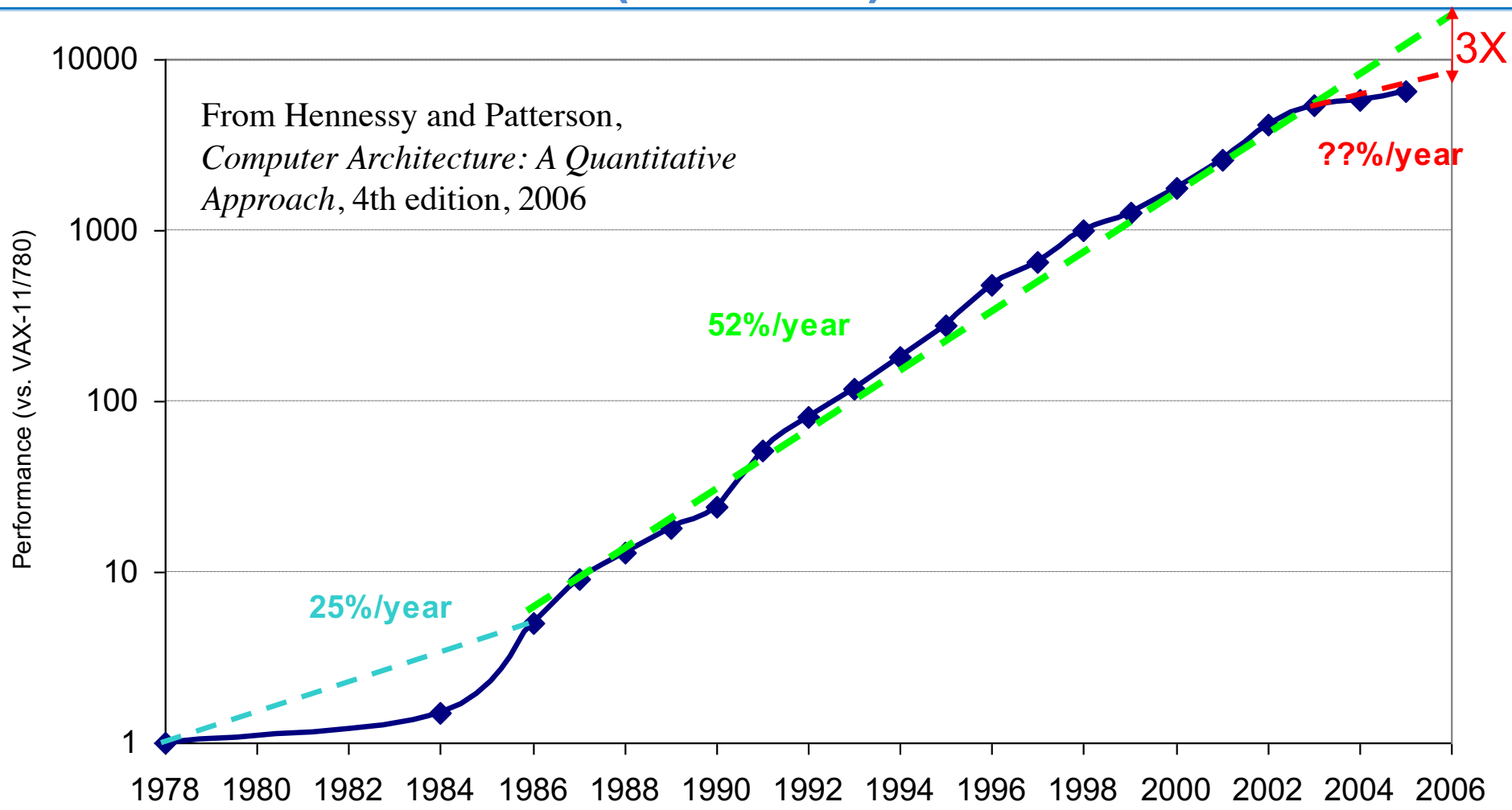
Facultatea de Automatică și Calculatoare
Universitatea Politehnica București


GPU
CORE10
CORE9
CORE8
CORE5
CORE7
CORE4
CORE6
CORE3
CORE2
CORE1

Recapitulare: Memoria Virtuală

- Suport pentru memorie virtuală implementat standard în toate procesoarele moderne
  - Creează iluzia unui spațiu amplu și partajat de memorie protejată
  - Programele pot fi scrise independent de configurația de memorie a mașinii pe care acestea rulează
- Tabelele de pagini ierarhice exploatează faptul că spațiul de adrese virtual nu este compact pentru a reduce dimensiunea informațiilor legate de mapare
- TLB cache translation/protection – informații care fac VM practică
  - Nu ar fi acceptabil să avem referințe la memorie multiple pentru o instrucțiune
- Interacțiunea dintre TLB lookup și cache tag lookup
  - Vrem să evităm inconsistențele la adresarea virtuală

Uniprocessor Performance (SPECint)



- **VAX : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**
- **RISC + x86: ??%/year 2002 to present**

Déjà vu?

Embedded Systems Laboratory

"… today's processors … are nearing an impasse as technologies approach the speed of light.."

David Mitchell, *The Transputer: The Time Is Now* (1989)

- Transputer-ul nu a apărut pe piață la momentul potrivit (Performanțele uniprocesor↑)
  ⇒ Procrastinarea recompensată: 2X seq. perf. / 1.5 ani
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

Paul Otellini, President, Intel (2005)

- Toate companiile de microprocesoare trec la MP (2X CPUs / 2 yrs)
  ⇒ Procrastinarea penalizată: 2X sequential perf. / 5 ani

Manufacturer/Year	AMD/'07	Intel/'07	IBM/'07	Sun/'07
Processors/chip	4	2	2	8
Threads/Processor	1	1	2	8
Threads/chip	4	2	4	64

Alți Factori ⇒ Multiprocesoare

- Creșterea aplicațiilor data-intensive
  - Baze de date, servere fișiere, …
- Creșterea interesului în servere, performanța serverelor
- Creșterea performanțelor desktop-urilor nu mai e așa de importantă
  - Mai puțin partea de grafică
- Înțelegere mai bună a cum pot fi folosite eficient multiprocesoarele
- Avantajul reducerii costurilor de design prin replicare
  - Comparativ cu reproiectarea de la (aproape) zero

Taxonomia Flynn

M.J. Flynn, "Very High-Speed Computers",
Proc. of the IEEE, V 54, 1900-1909, Dec. 1966.

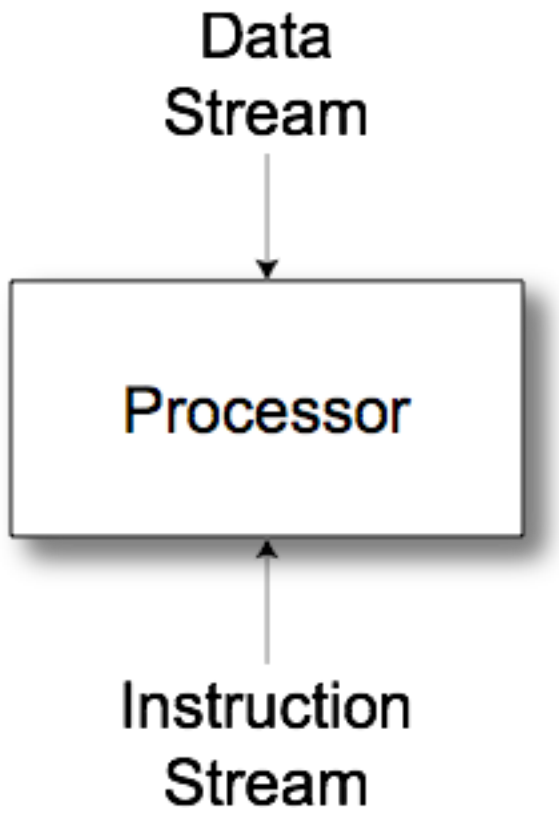
- Flynn - clasificare după flux de date și de control (1966)

Single Instruction, Single Data (SISD) (Uniprocessor)	Single Instruction, Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction, Single Data (MISD) (????)	Multiple Instruction, Multiple Data MIMD (Clusters, SMP servers)

- SIMD ⇒ Data-Level Parallelism
- MIMD ⇒ Thread-Level Parallelism
- MIMD populare datorită
  - Flexibilității: N programe sau 1 program multithreaded
  - Cost-effective: același MPU într-un desktop și în mașina MIMD

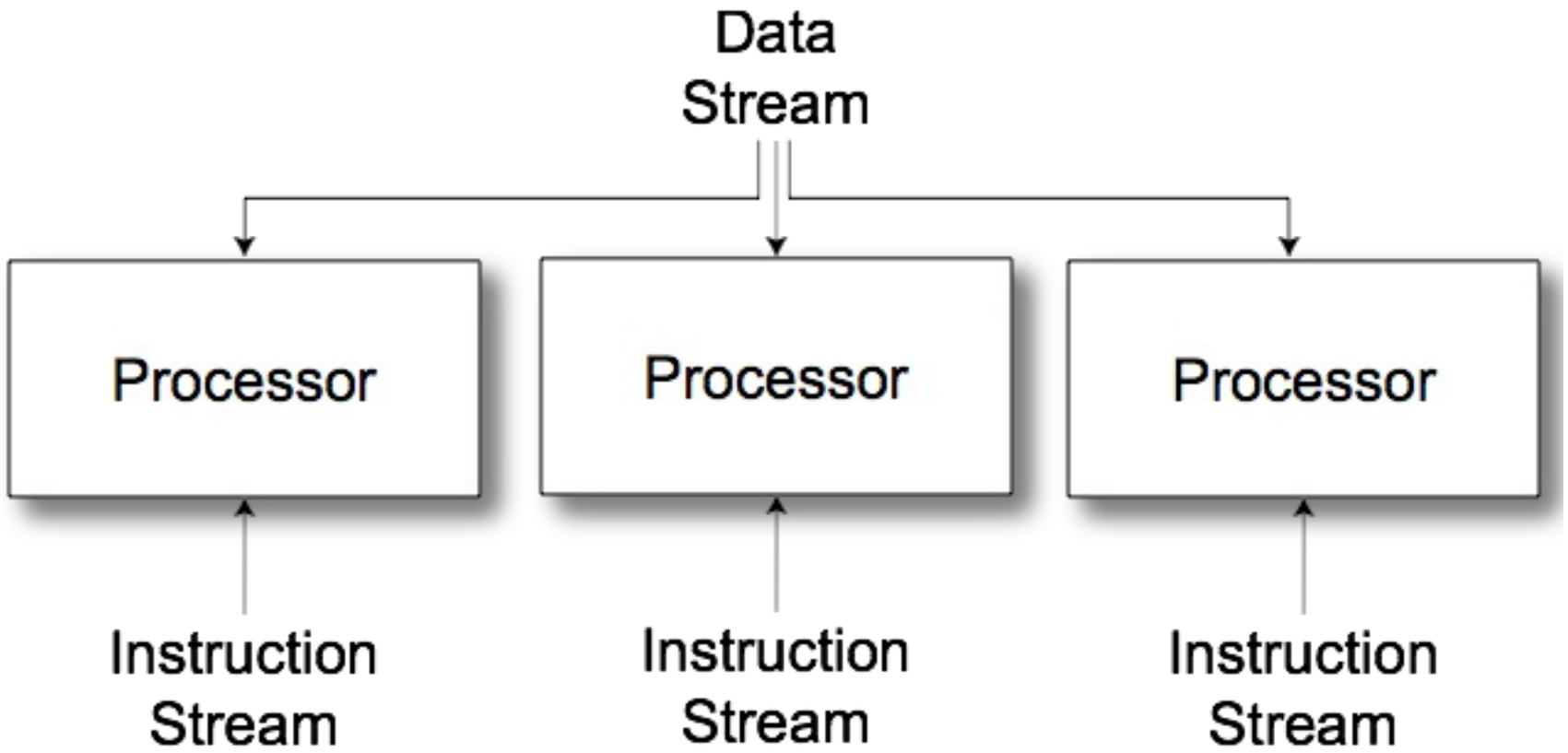
SISD

- Sisteme Uniprocesor



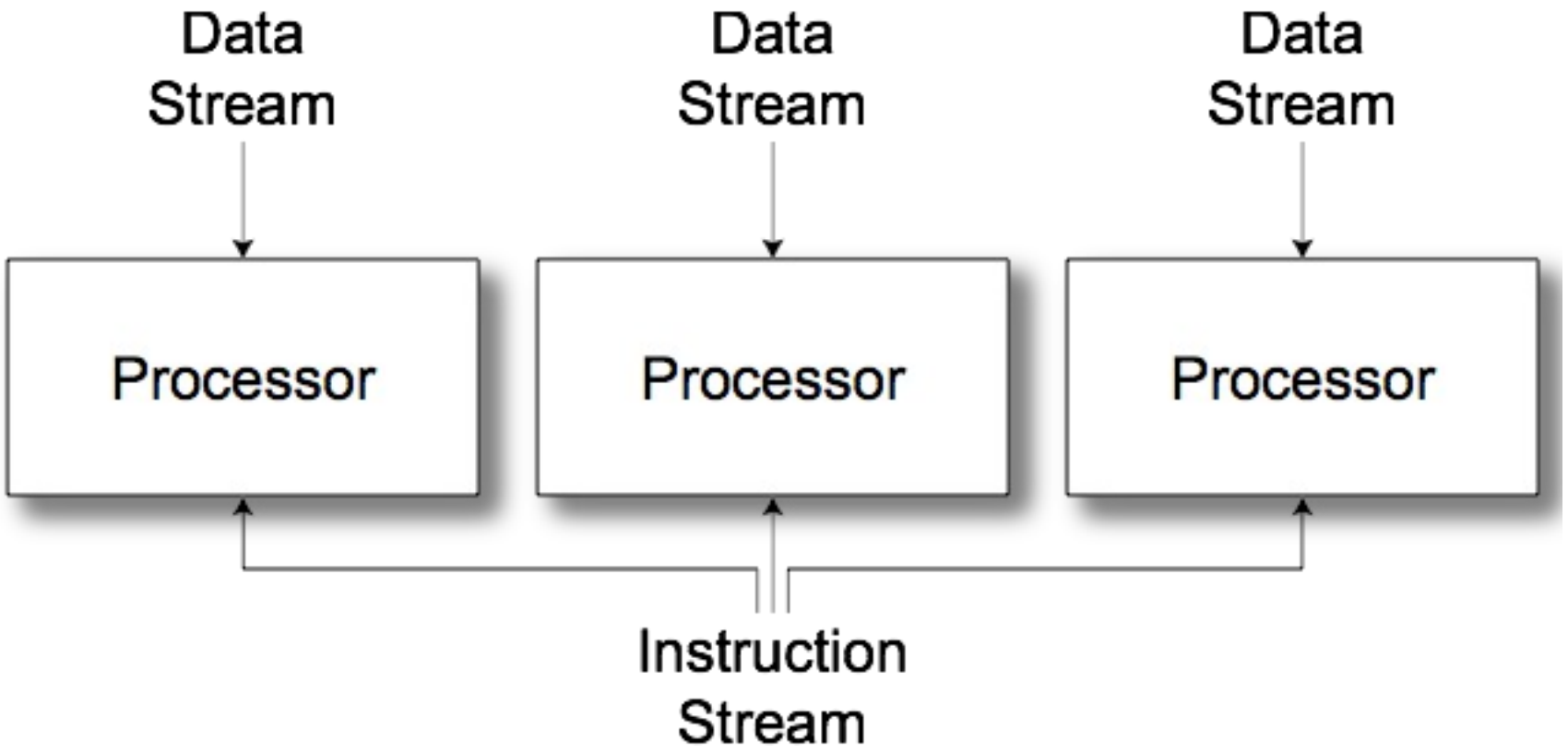
MISD

- Nu există exemple comerciale
  - De ex. aplică aceleași operații pe un set de date și găsește numerele prime



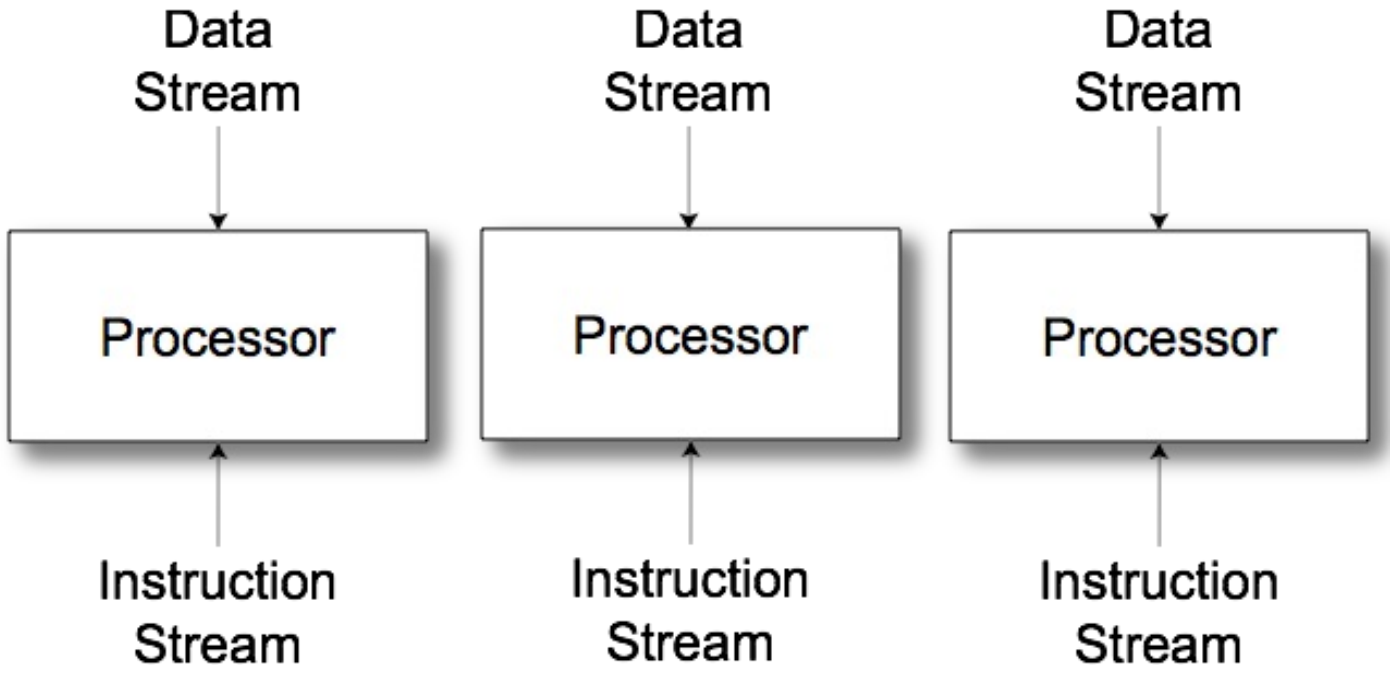
SIMD

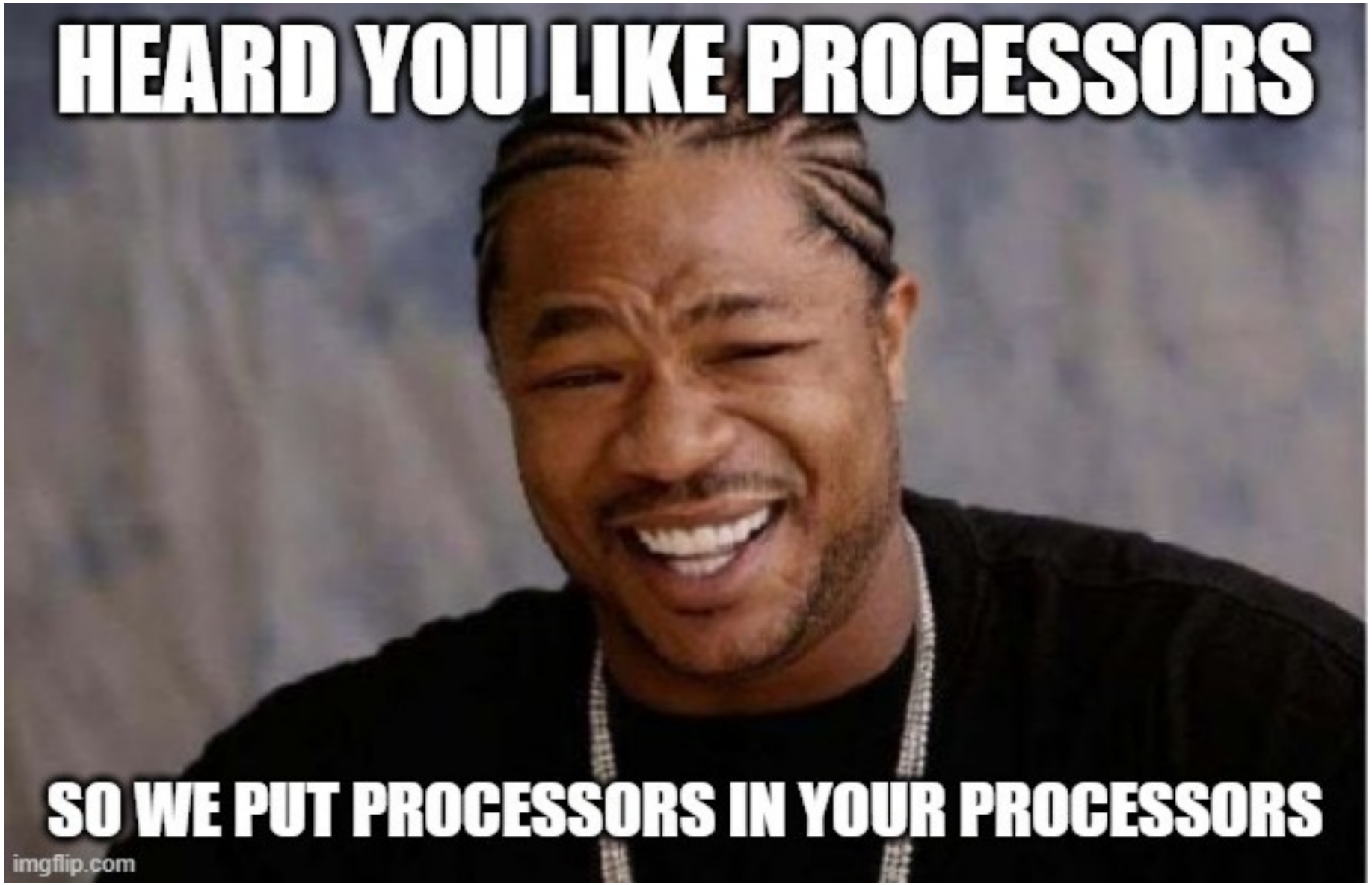
- Vector/Array Computers, procesoare grafice (GPU)



MIMD

- Message Passing
- Shared memory/distributed memory
    - Uniform Memory Access (UMA)
    - Non-Uniform Memory Access (NUMA)




HEARD YOU LIKE PROCESSORS
SO WE PUT PROCESSORS IN YOUR PROCESSORS
imgflip.com

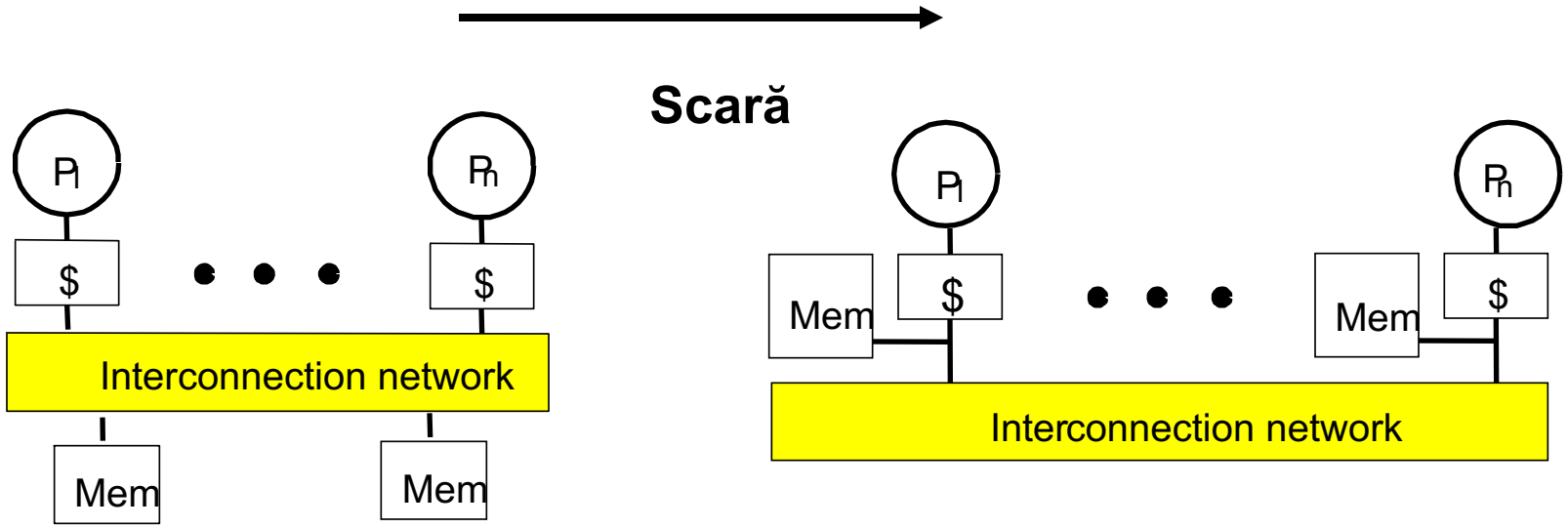
Back to Basics

- “Un calculator paralel este o colecție de elemente de procesare care cooperează și comunică pentru a rezolva probleme mari de calcul într-un timp scurt.”
- Arhitectura Paralelă = Arhitectura Calculatoarelor + Arhitectura de Comunicație

Două modele pentru Arhitectura de memorie și comunicații

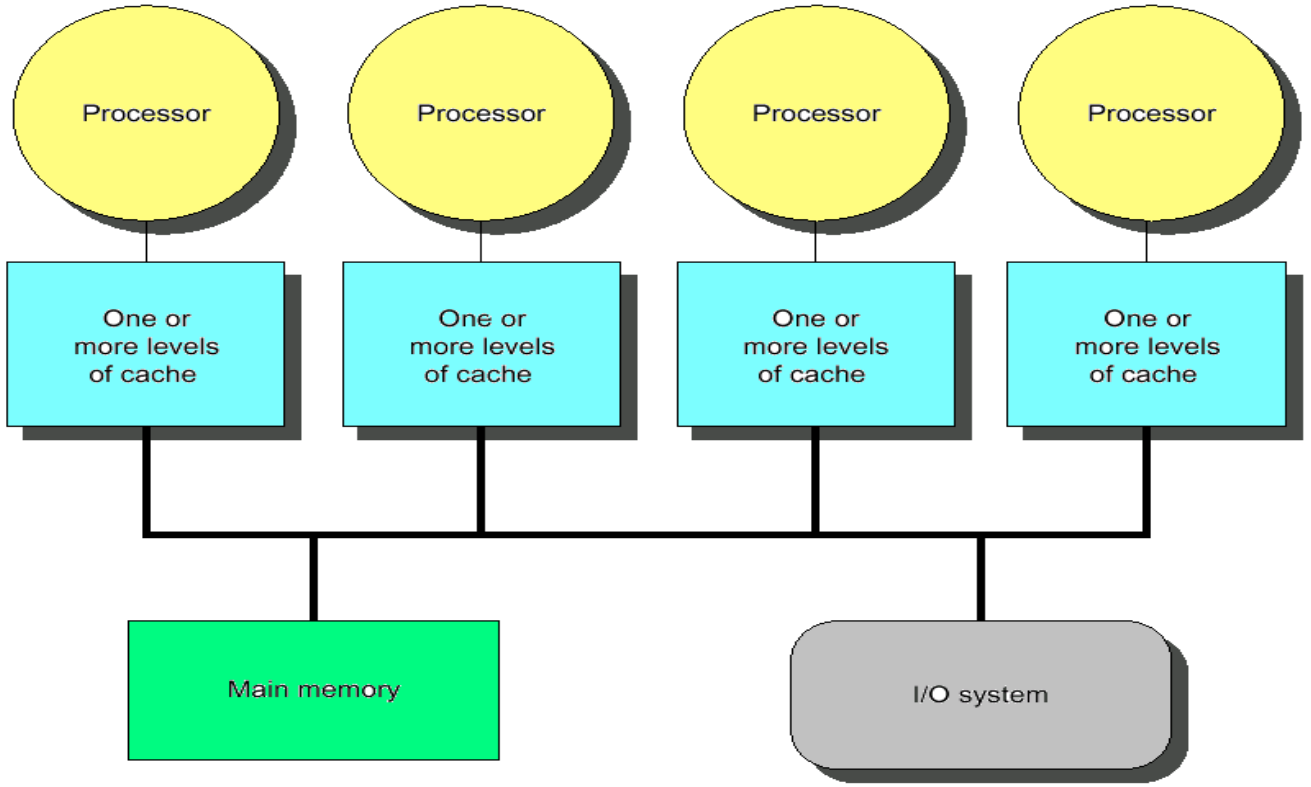
1. Comunicațiile apar prin pasarea explicită de mesaje între procesoare: message-passing multiprocessors (aka multicomputers)
   - Sistemele moderne tip *cluster* conțin unități de calcul independente ce comunică prin mesaje
2. Comunicațiile se petrec prin intermediul unui spațiu de adresă partajat (via operații load și store): shared-memory multiprocessors care pot fi:
   - UMA (Uniform Memory Access time) cu adrese partajate și memorie centralizată
   - NUMA (Non-Uniform Memory Access time multiprocessor) cu adrese partajate și memorie distribuită

Memorie centralizată vs. distribuită



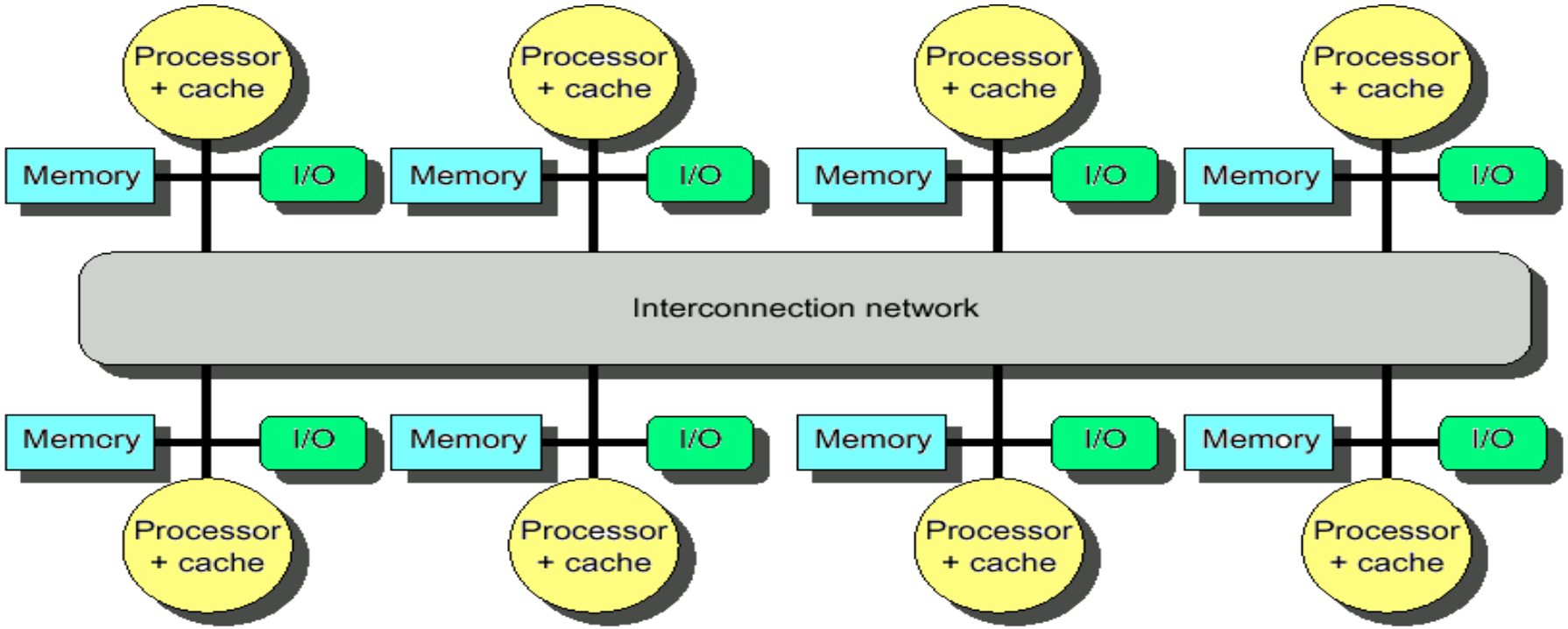
Memorie centralizată **Memorie distribuită**

Memorie Centralizată Partajată



- Procesoarele partajează o singură memorie centralizată (UMA) printr-o magistrală de interconexiuni
- Fezabil pentru un număr mic de procesoare
- Arhitecturile cu memorie centralizată sunt cea mai comună formă de design MIMD

Memorie Distribuită



Folosește memorie distribuită fizic (NUMA) ce permite un număr mare de procesoare Avantaje:

- Permite scalarea lățimii de bandă a memoriei
- Reduce latența memoriei

Dezavantaj:

- Complexitate mărită în comunicația de date

Procesoare cu memorie centralizată

- Numite și symmetric multiprocessors (SMPs) deoarece memoria partajată unică are o relație simetrică cu toate procesoarele
- Cache mare $\Rightarrow$ o singură memorie poate să satisfacă cererile unui număr mic de procesoare
- Poate să scaleze la câteva zeci de procesoare prin folosirea unui switch și a mai multor bancuri de memorie
- Deși, d.p.d.v. tehnic se poate scala și mai mult, devine mai puțin atrăgător pe măsură ce crește numărul de procesoare ce partajează aceeași memorie centralizată

Multiprocesoare cu memorie distribuită

- **Pro:** Metodă Cost-effective de a scala lățimea de bandă
  - Doar dacă majoritatea acceselor sunt la memoria locală
- **Pro:** Reduce latența acceselor locale la memorie

- **Contra:** Comunicația de date dintre procesoare e mai complexă
- **Contra:** Software-ul trebuie să fie conștient de localitatea datelor pentru a profita de lățimea de bandă mărită

Problemele procesării paralele

- O mare problemă e că % dintr-un program este inerent secvențial
  - Ce înseamnă inerent secvențial?
- Presupunem 80X speedup de la 100 procesoare. Care este procentul original de program care este paralelizabil?

  a. 10%

  b. 5%

  c. 1%

  d. <1%

Răspunsul, conform legii lui Amdahl

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times ((1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}) = 1$$

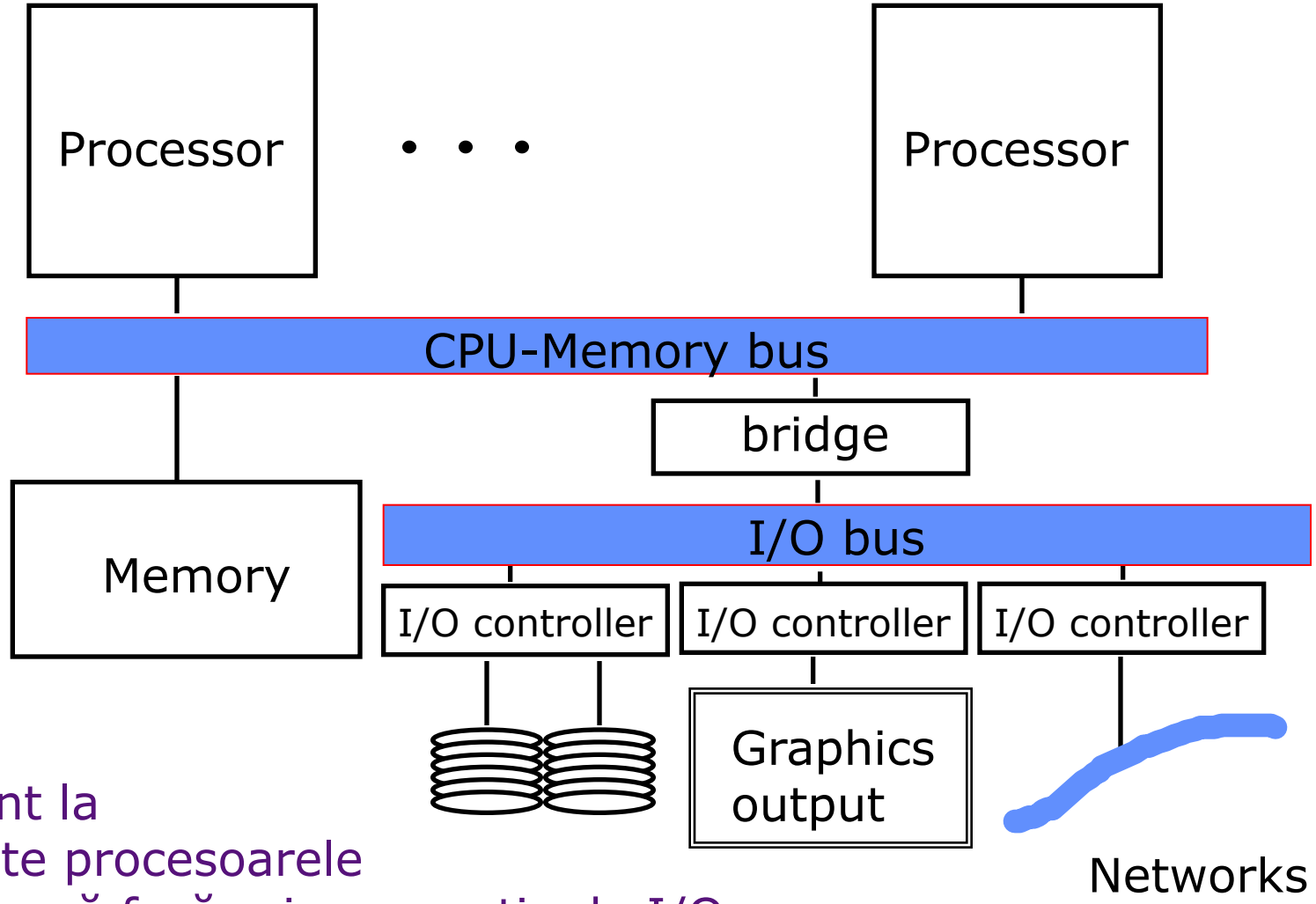
$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Comunicare și sincronizare

- Procesele paralele trebuie să coopereze pentru a termina mai repede un singur task
- Necesită comunicație distribută și sincronizare
  - Comunicație pentru valorile datelor, sau ”ce”
  - Sincronizare pentru control, sau ”când”
  - Comunicația și sincronizarea sunt deobicei inter-dependente
    - Ex. “ce” depinde de “când”
- Message-passing agregă datele și semnalele de control
  - Sosirea mesajelor codifică “ce” și “când”
- În mașinile cu memorie partajată, comunicația este menținută prin cache-uri coerente și sincronizarea prin operații atomice cu memoria
  - Datorită apariției multiprocesoarelor single-chip, este foarte posibil ca sistemele cache-coherent cu memorie partajată să fie forma dominantă de multiprocesoare
  - Cursul de astăzi se axează pe problema sincronizării

Multiprocesoare simetrice



simetrie

- Toate memoriile sunt la distanțe egale de toate procesoarele
- Orice procesor poate să facă orice operație de I/O (setează un transfer DMA)

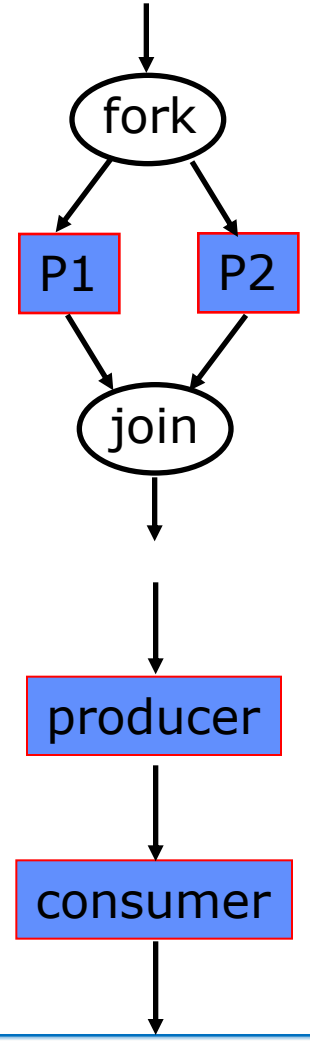
Sincronizare

Nevoia de sincronizare apare de fiecare dată când avem procese concurente într-un sistem
(chiar și într-unul uniprocesor)

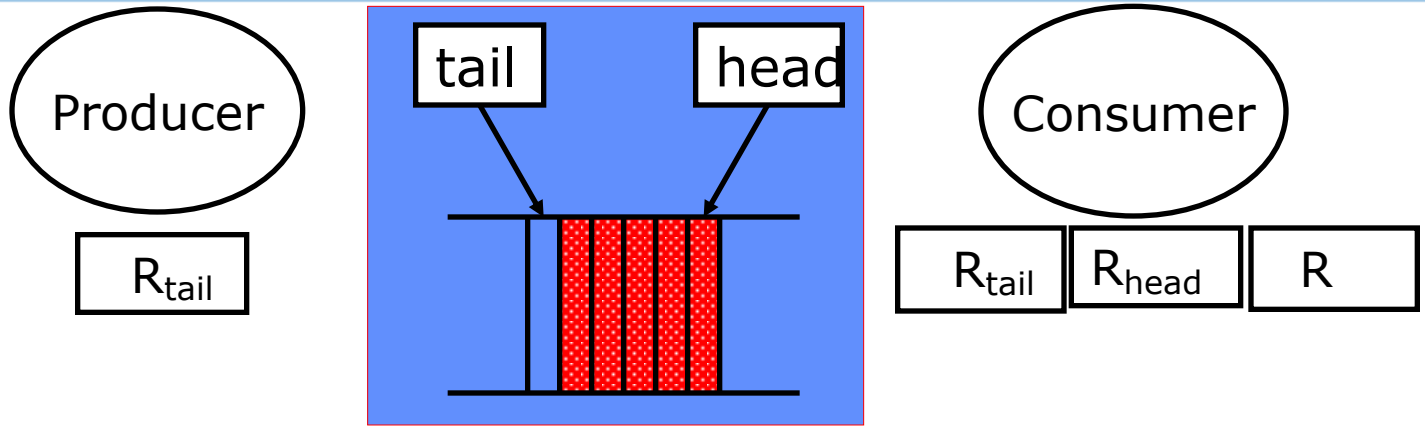
Forks & Joins: În programarea paralelă, un proces poate să aștepte până când s-au produs mai multe evenimente

Producer-Consumer: Un proces consumator trebuie să aștepte până ce un proces producător a generat datele

Folosirea exclusivă a unei resurse: Sistemul de operare trebuie să asigure că doar un singur proces utilizează o resursă la un moment dat



Exemplu producător-consumator



Producer posting Item x:

Load R_{tail}, (tail)

Store (R_{tail}), x

R_{tail}=R_{tail}+1

Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail)

if R_{head}==R_{tail} goto spin

Load R, (R_{head})

R_{head}=R_{head}+1

Store (head), R_{head}

process(R)

Programul este scris presupunând că instrucțiunile sunt executate în ordine.

Probleme?

Exemplu producător-consumator

Embedded Systems Laboratory

Producer posting Item x:

Load R_{tail}, (tail)

1 Store (R_{tail}), x

$R_{tail}=R_{tail}+1$

2 Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail) 3

if $R_{head}==R_{tail}$ goto spin

Load R, (R_{head}) 4

$R_{head}=R_{head}+1$

Store (head), R_{head}

process(R)

Poate fi actualizat pointer-ul tail înainte ca elementul x să fie stocat?

Programatorul presupune că dacă 3 se execută după 2, atunci 4 se execută după 1.

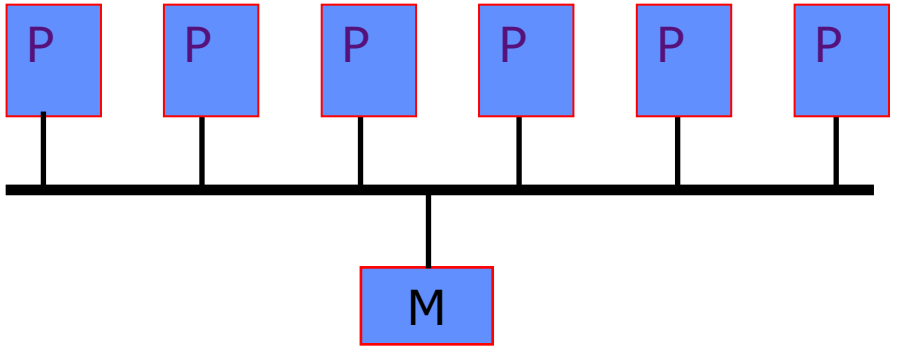
Secvențele cu probleme sunt:

2, 3, 4, 1

4, 1, 2, 3

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' ?

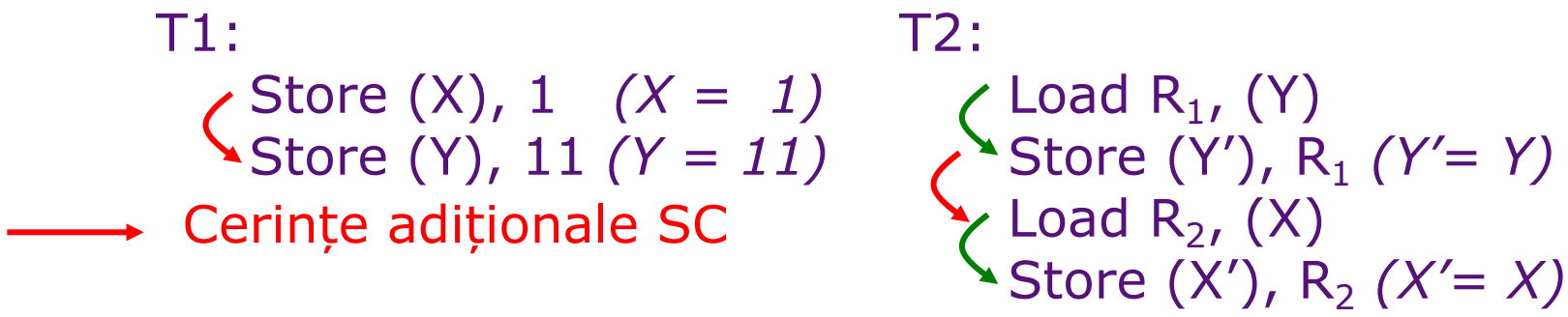
$(X',Y') \varepsilon \{(1,11), (0,10), (1,10), (0,11)\}$?

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 (⟶)

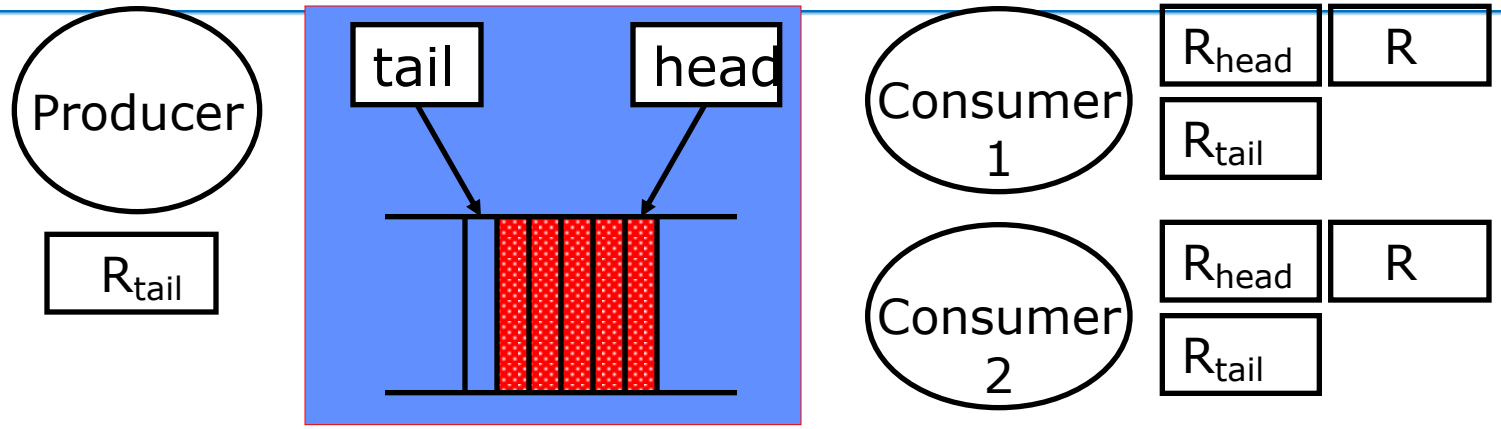
Care sunt cele din exemplele noastre ?



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

Mai multe despre asta mai târziu

Exemplu consumatori multipli



Producer posting Item x:

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

Consumer:

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

Secțiune critică:
Trebuie executată atomic de un singur consumator ⇒ locks

Ce nu e în regulă cu acest cod?

Locks or Semaphores

E. W. Dijkstra, 1965

Embedded Systems Laboratory

Un semafor (mutex) este un întreg ne-negativ ce implementează următoarele operații:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P() și V() trebuie executate atomic, adică fără

- *intreruperi* sau
- *accese intercalate la s* de către alte procesoare

```
Process i
    P(s)
        <critical section>
    V(s)
```

Valoarea inițială a lui s determină numărul maxim de procese din regiunea critică

Implemetarea semafoarelor

Semafoarele (mutual exclusion) pot fi implementate folosind instrucțiuni obișnuite Load and Store în modelul unei memorii cu Consistență Secvențială. Cu toate acestea, protocoalele de excluziune mutuală sunt greu de proiectat...

Soluție mai simplă:

instrucțiuni atomice read-modify-write

Exemple: *m este o locație de memorie, R un registru*

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

```
Fetch&Add (m), RV, R:
    R ← M[m];
    M[m] ← R + RV;
```

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

Exemplu consumatori multipli *Test&Set Instruction*

Embedded Systems Laboratory

```
P:      Test&Set (mutex),Rtemp
        if (Rtemp!=0) goto P
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead=Rhead+1
        Store (head), Rhead
V:      Store (mutex),0
        process(R)
```

Critical Section

Alte instrucțiuni atomice read-modify-write (Swap, Fetch&Add, etc.) pot de asemenea să implementeze P și V

Ce se întâmplă dacă procesul se oprește în regiunea critică?

Sincronizare nonblocantă


```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
      then  M[m]=Rs;
            Rs=Rt ;
            status ← success;
      else  status ← fail;
```

status *este un argument implicit*

```
try:   Load Rhead, (head)
spin:  Load Rtail, (tail)
       if Rhead==Rtail goto spin
       Load R, (Rhead)
       Rnewhead = Rhead+1
       Compare&Swap(head), Rhead, Rnewhead
       if (status==fail) goto try
       process(R)
```

Load-reserve & Store-conditional

Embedded Systems Laboratory

Registre speciale pentru a ține flag-ul și adresa și rezultatul lui store-conditional

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

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

```
try:
spin:   Load-reserve Rhead, (head)
        Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead = Rhead + 1
        Store-conditional (head), Rhead
        if (status==fail) goto try
        process(R)
```

Performanța mutex-urilor

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:

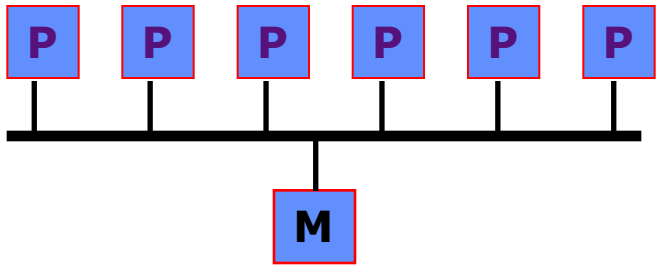
degree of contention,

cache-uri,

out-of-order execution și Loads & Stores

mai târziu in curs ...

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

Embedded Systems Laboratory

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):

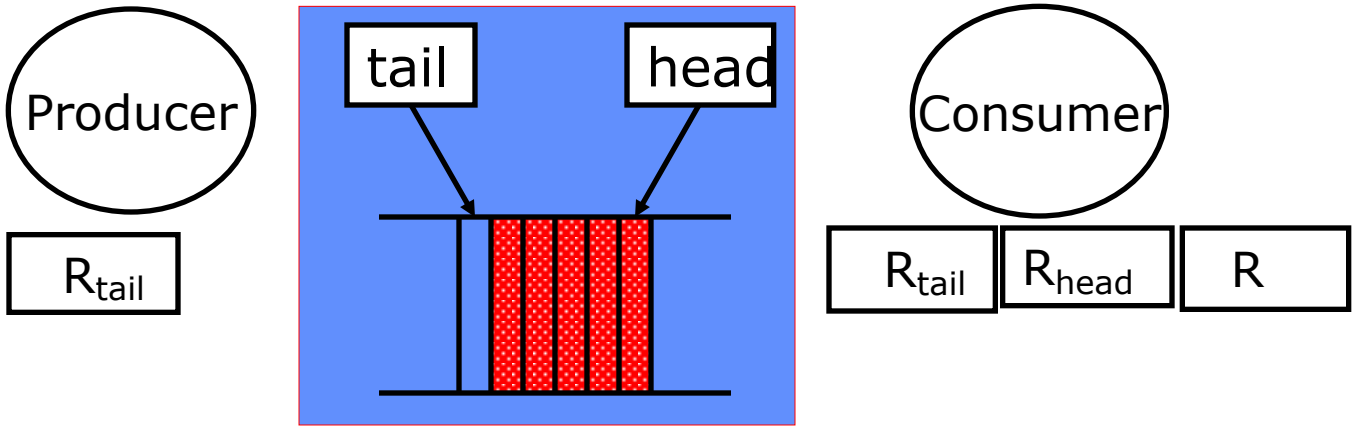
Membar #LoadLoad, Membar #LoadStore

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 Rtail, (tail)
        Store (Rtail), x
        Membar_SS
        Rtail=Rtail+1
        Store (tail), Rtail
```

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

Consumer:

```
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Membar_LL
        Load R, (Rhead)
        Rhead=Rhead+1
        Store (head), Rhead
        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

Embedded Systems Laboratory

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

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*

Embedded Systems Laboratory

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

Embedded Systems Laboratory

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 mutual*
- *Soluția pentru n procese a fost dată de Dijkstra și este puțin mai complicată!*

Analiza algoritmului lui Dekker

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;
```


Multithreaded programming
Theory
Actual

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