Curs 3

Programare Paralela si Distribuita

Paralelism la nivel de instrucţiune
Paralelism implicit versus paralelism explicit
Modele de calcul versus sisteme
Procese versus Fire de executie
Race-condition - sectiuni critice

PARALLELISM LA NIVEL DE INSTRUCTIUNE

Parallelism in the CPU

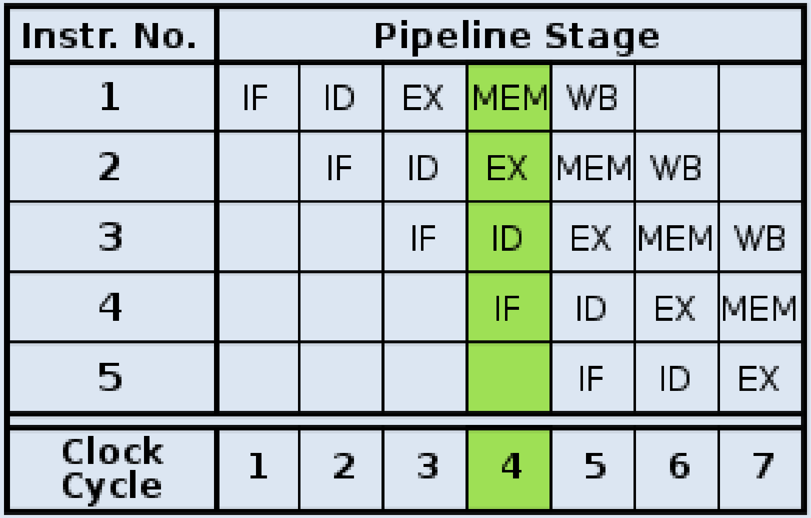
(ref: https://www.tutorialspoint.com/cuda/cuda_tutorial.pdf)

- Following are the five essential steps required for an instruction to finish:
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Instruction execute (Ex)
  - Memory access (Mem)
  - Register write-back (WB)
- This is a basic five-stage RISC architecture.

- There are multiple ways to achieve parallelism in the CPU.
  - one is ILP (Instruction Level Parallelism), also known as pipelining.

Instruction Level Parallelism

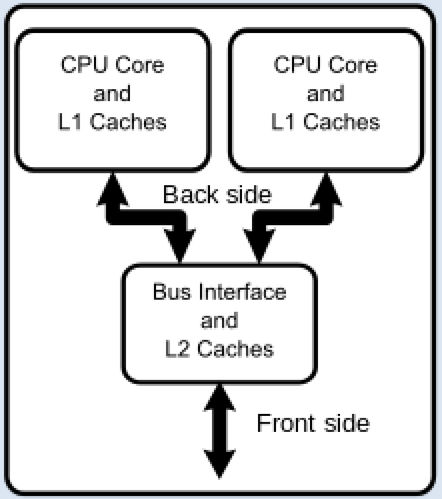
- Using instruction pipelining, the instruction throughput has increased. So, it is possible to process many instructions in one-clock cycle.



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Multi-core processor

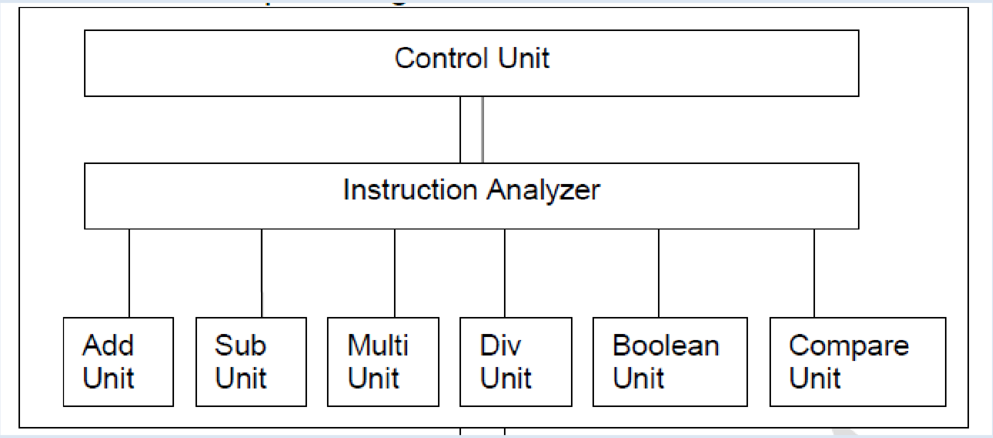
- A **multi-core processor** is a microprocessor on a single integrated circuit (chip) with two or more separate processing units, called cores, each of which reads and executes program instructions.



ILP in a superscalar architecture

- The primary difference between a **superscalar** and a **pipelined** processor is (a superscalar processor is also pipelined) that the former uses multiple execution units (on the same chip) to achieve ILP whereas the latter divides the EU in multiple phases to do that.
  - This means that in superscalar, ***several instructions can simultaneously be in the same stage of the execution cycle***. This is not possible in a simple pipelined chip.
  - Superscalar microprocessors can execute two or more instructions at the same time. They typically have at least 2 ALUs.
- **Superscalar processors can dispatch multiple instructions in the same clock cycle.**
  - This means that multiple instructions can be started in the same clock cycle.
  - In a pipelined architecture at any clock cycle, only one instruction is dispatched.
  - This is not the case with superscalars. But we have only one instruction counter (in-flight, multiple instructions are tracked). This is still just one process !

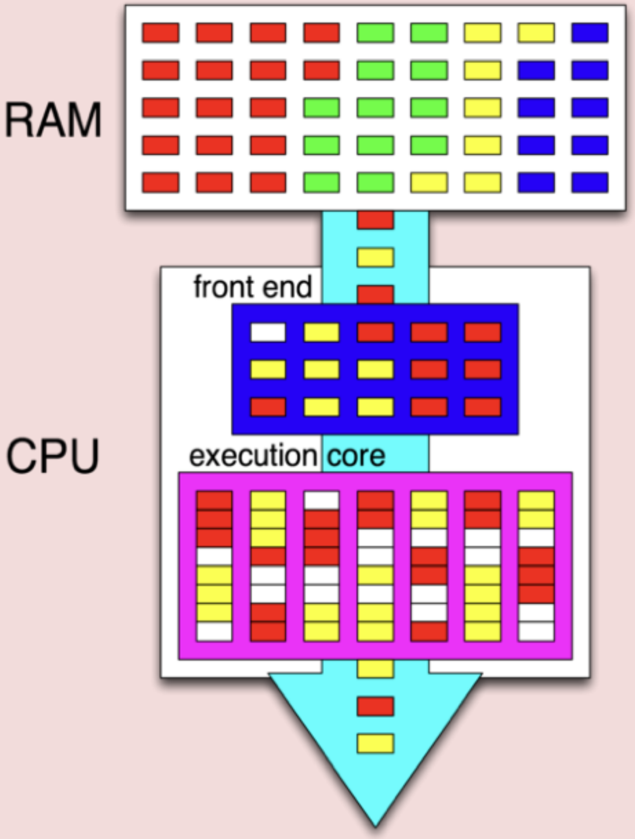
Many functional units



Hyper-threading

- HT is just a technology to utilize a processor core better.
  - Many times, a processor core is utilizing only a fraction of its resources to execute instructions.
- What HT does is that **it takes a few more CPU registers,** and executes more instructions on the part of the core that is sitting idle.
- Thus, one core now appears as two core.
  - It is to be considered that they are not completely independent.
- If both the ‘cores’ need to access the CPU resource, one of them ends up waiting.
  - That is the reason why we cannot replace a dual-core CPU with a hyper-threaded, single core CPU.
  - A dual core CPU will have truly independent, out- of-order cores, each with its own resources.
- HT is Intel’s implementation of SMT (Simultaneous Multithreading).
- SPARC has a different implementation of SMT, with identical goals.

SMT(Simultaneous Multithreading)



- The pink box represents a single CPU core.

- The RAM contains instructions of 4 different programs, indicated by different colors.

- The CPU implements the SMT, using a technology similar to hyper-threading.

=> it is able to run instructions of two different programs (red and yellow) simultaneously.

- White boxes represent pipeline stalls.

Vector processors

- a **vector processor** is a (CPU) that implements an instruction set where its instructions are designed to operate efficiently and effectively on one-dimensional arrays

- **scalar processor** is a CPU whose instructions operate on single data items only,

BUT

- some of these scalar processors have additional
  - single instruction,
  - multiple data (SIMD) or
  - SWAR(*SIMD within a register* **)**

  Arithmetic Units

scalar processors with (SIMD)

- **Pure (fixed) SIMD** - also known as "Packed SIMD“ or SIMD within a Register (SWAR)
  - Examples: Intel x86's MMX, SSE and AVX instructions,
    AMD's 3DNow! extensions, ARM NEON,
    Sparc's VIS extension, PowerPC's AltiVec and MIPS' MSA

- **Predicated SIMD** -associative processing
  - - examples: ARM SVE2 and AVX-512

illustration

- For (…i<n…) C[i] = A[i] + B[i]
  - In every iteration there will be 2 load, 1 store and 1 add instruction(simple view)

- With SIMD instruction set less than the number of CPU instructions - since each instruction will process multiple elements.
  - with SIMD instruction set that has **128 bit register**, processing on 4 integers at a time is possible:

- ○ A single load instruction to get 4 values of A into a 128 bit SIMD register
- ○ A single load instruction to get 4 values of B into a 128 bit SIMD register
- ○ A single add instruction to add corresponding values in both register.

PARALLELISM LA NIVEL DE PROGRAM

Paralelism implicit SAU explicit

- *Implicit Parallelism*

  Programatorul nu specifica explicit paralelismul,
  lasa compilatorul si sistemul de suport al executiei (*run-time support system*)

  sa paralelizeze automat.

- *Explicit Parallelism*

  Programatorul specifica explicit paralelismul in codul sursa prin constructii speciale de limbaj, sau prin directive complexe sau prin apeluri de biblioteci.

Modele de programare paralele Implicite

Implicit Parallelism: Parallelizing Compilers

- Automatic parallelization of sequential programs
  - Dependency Analysis
  - Data dependency
  - Control dependency

Se poate obtine paralelizare dar nu completa si impune analize foarte dificile!

Exemplificare simpla:

JIT(Just In Time) compilation can choose SSE2 vector CPU instructions when it detects that the CPU supports them

Modele de programare paralela Explicita

Cele mai folosite:

- Shared-variable model
- Message-passing model
- Data-parallel model

Legatura intre Modele de Programare si Arhitecturi

Exemplificare pe problema concreta:

Suma de numere

Exemplu de aplicatie paralela: calcularea sumei

$$\sum_{i=0}^{n-1} f(A[i])$$

Solutia generala:

n/p operatii $\longrightarrow$ *p* procesoare (procese).

Se disting doua seturi de date:

-partajate: valorile *A[i]* si suma finala;

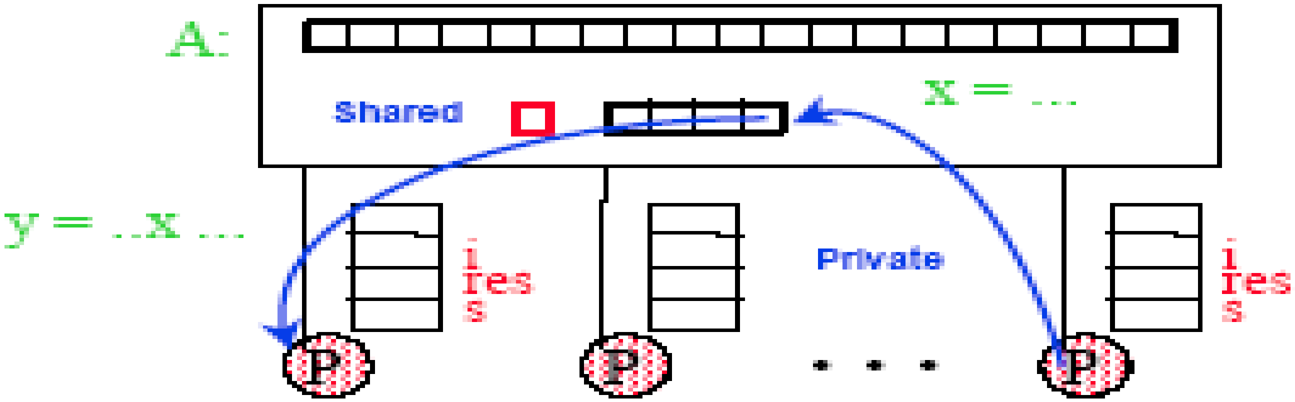
-private: evalurile individuale de functii si sumele partiale

1) ***Model de programare: spatiu partajat de adrese***.

Programul = colectie de fire de executie, fiecare avand un set de variabile private, iar impreuna partajeaza un alt set de variabile.

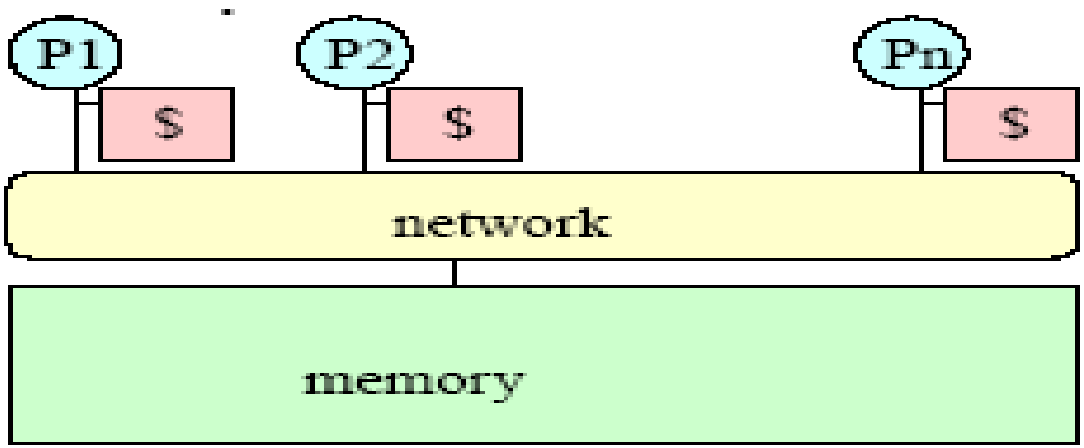
Comunicatia dintre firele de executie: **prin citirea/scrierea variabilelor partajate.**

Coordonarea firelor de executie prin operatii de **sincronizare**: indicatori (flags), lacate (locks), semafoare, monitoare.



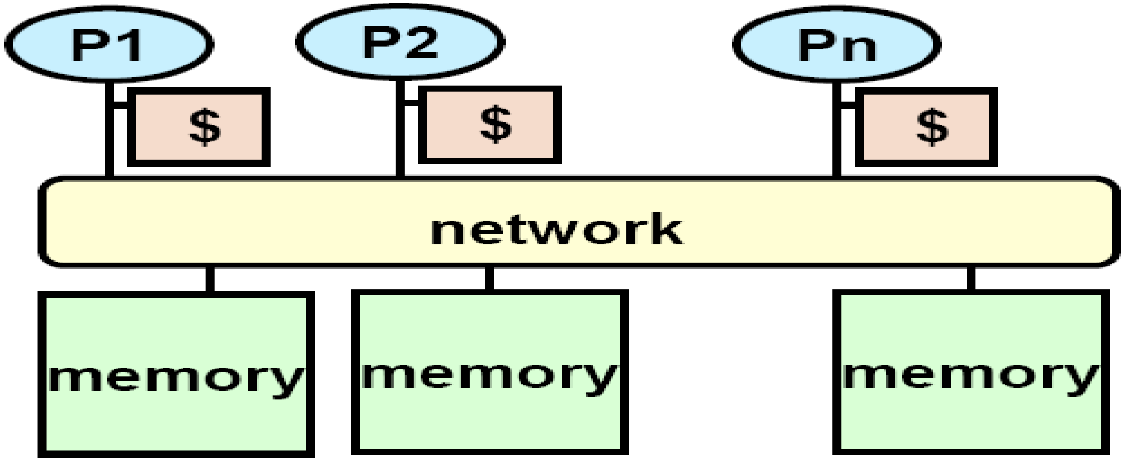
Masina paralela corespunzatoare modelului 1: masina cu memorie partajata (sistemele multiprocesor, sistemele multiprocesor simetrice -- SMP-Symmetric Multiprocessors).

Exemple: sisteme de la Sun, DEC, Intel (Millennium), SGI Origin.



Variante ale acestui model:

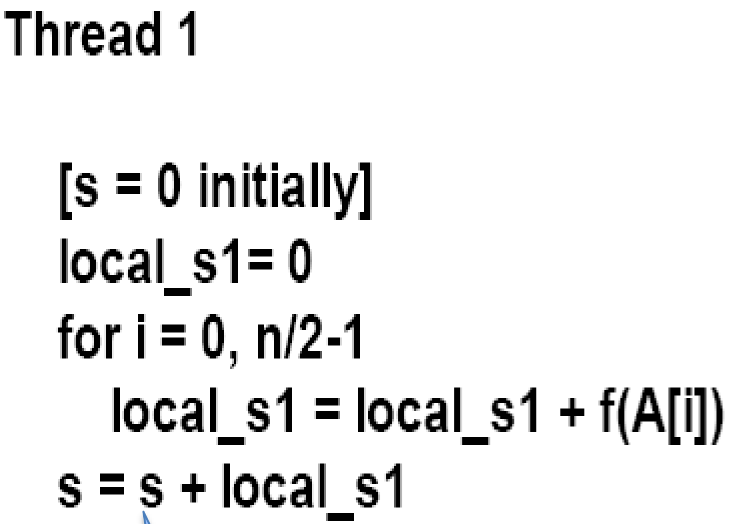
a) masina cu memorie partajata distribuita (logic partajata, dar fizic distribuita. Exemplu: SGI Origin (scalabila la cateva sute de procesoare).



b) masina cu spatiu partajat de adrese (memoriile cache inlocuite cu memorii locale). Exemplu: Cray T3E.

single address space

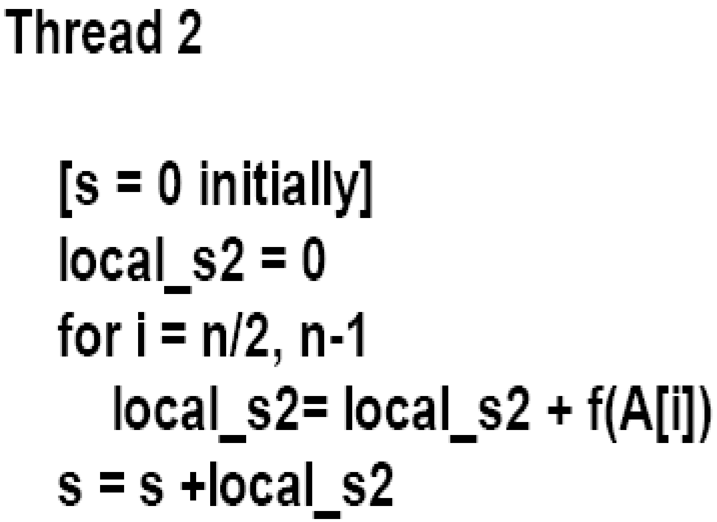
O posibila solutie pentru rezolvarea problemei:



Thread 1

```
[s = 0 initially]
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + f(A[i])
s = s + local_s1
```

?



Thread 2

```
[s = 0 initially]
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + f(A[i])
s = s +local_s2
```

?

Este necesara sincronizarea threadurilor pentru accesul la variabilele partajate !

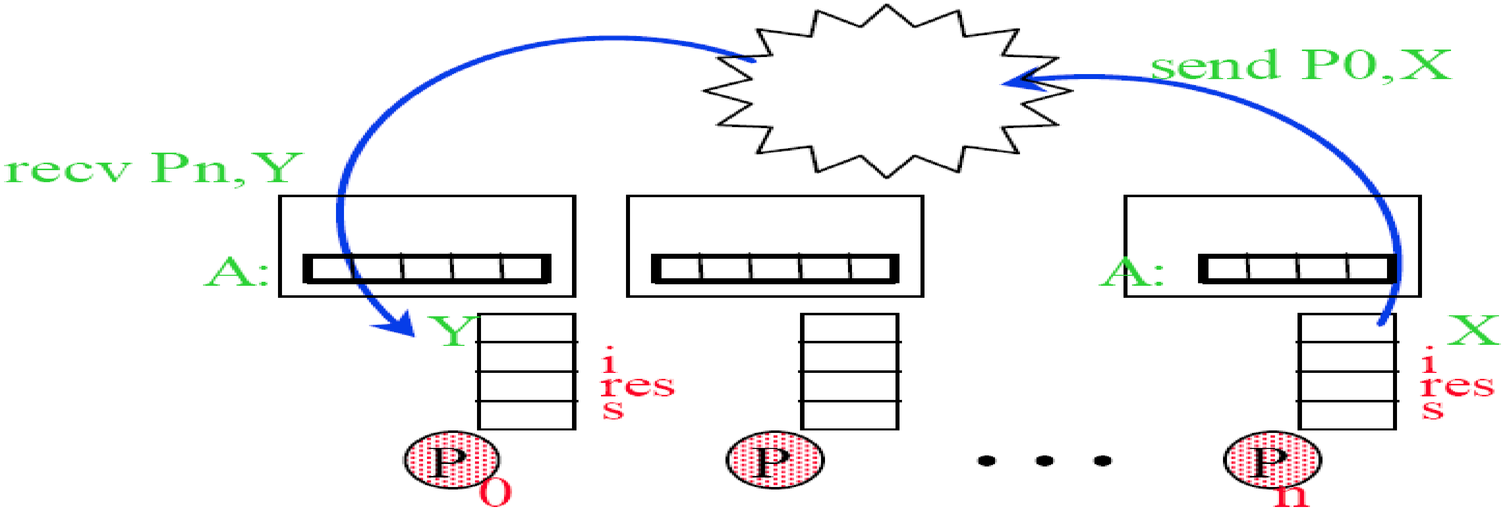
Exemplu: prin excludere mutuala, folosind operatia de blocare(lock):

```
Thread 1                      Thread 2
   lock                          lock
   load s                        load s
   s = s+local_s1                s = s+local_s2
   store s                       store s
   unlock                        unlock
```

2) *Modelul de programare: transfer de mesaje*.

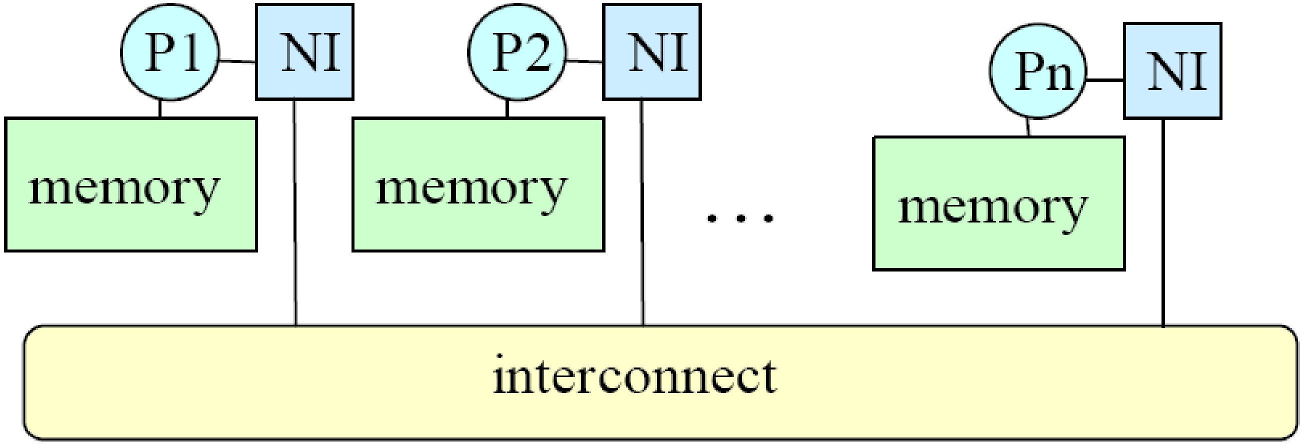
Programul = colectie de procese, fiecare cu thread de control si spatiu local de adrese, variabile locale, variabile statice, blocuri comune.

Comunicatia dintre procese: prin transfer explicit de date (exemplu:perechi de operatii corespunzatoare **send** si **receive** la procesele sursa si respectiv destinatie; alte variante sunt posibile).
Coordonarea: transfer de mesaje
Datele partajate din punct de vedere logic sunt partitionate intre toate procesele.
=> asemanare cu programarea distribuita!



Exista biblioteci standard (exemplu: MPI si PVM).

Masina corespunzatoare modelului 2
sistem cu memorie distribuita (multicalculator):



Exemple: Cray T3E (poate fi incadrat si in aceasta categorie), IBM SP2, NOW, Millenium.

O posibila solutie a problemei in cadrul modelului in transfer de mesaje simplificare => suma se calculeaza : *s = f(A[1]) + f(A[2])* :
(consideram operatii: send si receive blocante !!!)

Procesor 1

```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

Procesor 2

```
xlocal = f(A[2])
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote
```

Deadlock?

sau:

Procesor 1

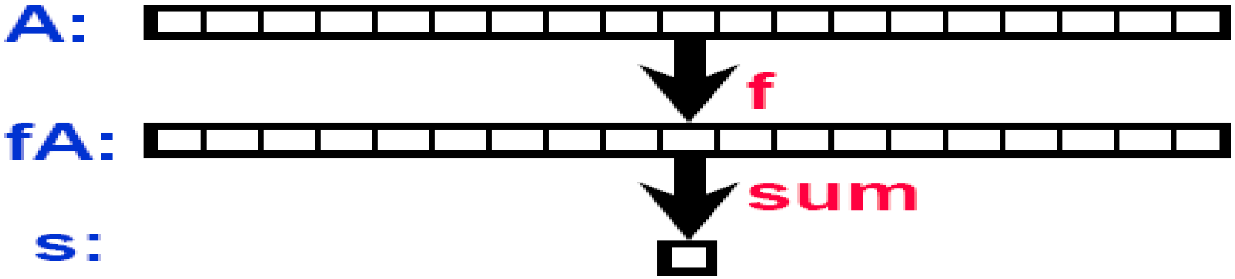
```
    xlocal = f(A[1])
    send xlocal, proc2
    receive xremote, proc2
    s = xlocal + xremote
```

Procesor 2

```
    xlocal = f(A[2])
    receive xremote, proc1
    send xlocal, proc1
    s = xlocal + xremote
```

3) *Modelul de programare: paralelism al datelor*.

Program: Thread singular, secvential de control care controleaza un set de operatii paralele aplicate intregii structuri de date, sau numai unui singur subset.

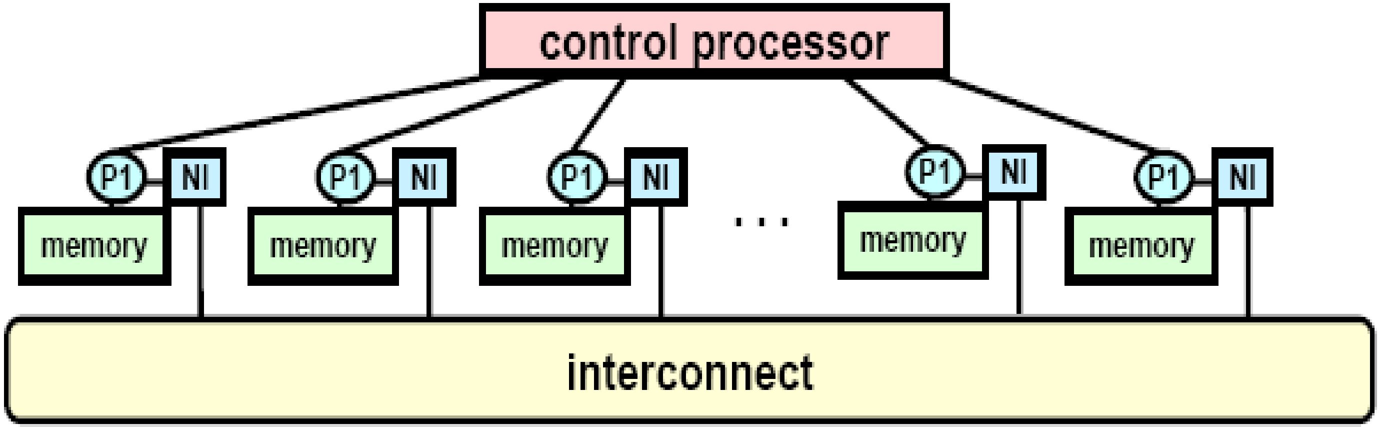


map + reduce

Comunicatia: implicita, in modul de deplasare a datelor.

Eficienta numai pentru anumite probleme (exemplu: prelucrari de tablouri)!

Masina corespunzatoare modelului 3: sistem SIMD - Single Instruction Multiple Data (numar mare de procesoare elementare comandate de un singur procesor de control, executand aceeasi instructiune, posibil anumite procesoare inactive la anumite momente de timp - la executia anumitor instructiuni).



Exemple: CM2, MASPAR, sistemele sistolice VLSI

Varianta: masina vectoriala (un singur procesor cu unitati functionale multiple, toate efectuand aceeasi operatie in acelasi moment de timp).

~ GPU

4) Modelul hibrid

=> cluster de SMP-uri sau CLUMP (mai multe SMP-uri conectate intr-o retea).

Fiecare SMP: sistem cu memorie partajata!

Comunicatia intre SMP-uri: prin transfer de mesaje.

Exemple: Millennium, IBM SPx, ASCI Red (Intel).

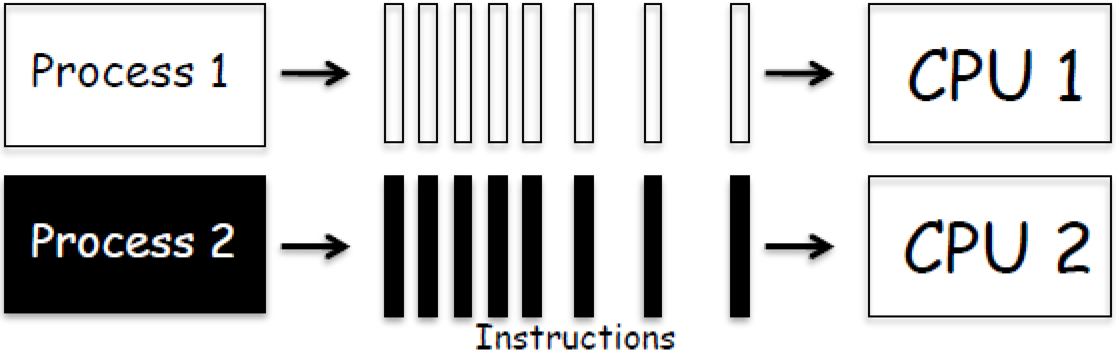
Model de programare:

- se poate utiliza transfer de mesaje, chiar in interiorul SMP-urilor!
- Varianta hibrida!

Procese versus Fire de executie

Multiprocessing -> Paralelism

- *Multicore processors*

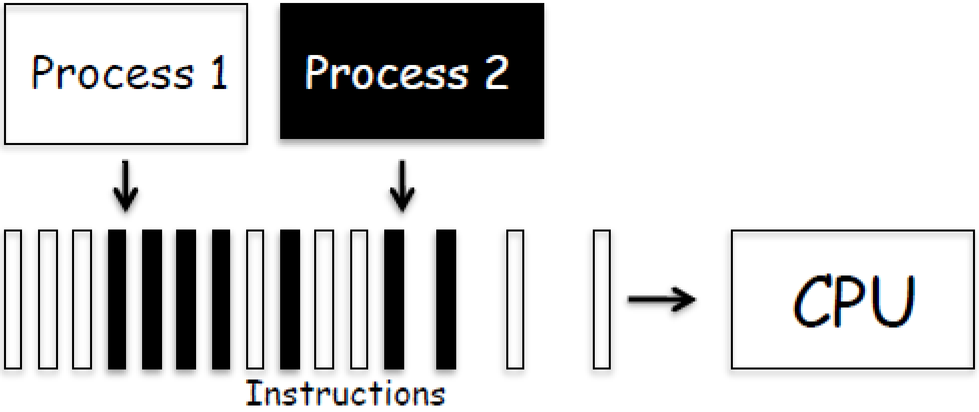


- Daca se folosesc mai multe unitati de procesare

=>Procesele se pot executa in acelasi timp

Multitasking-> Concurenta

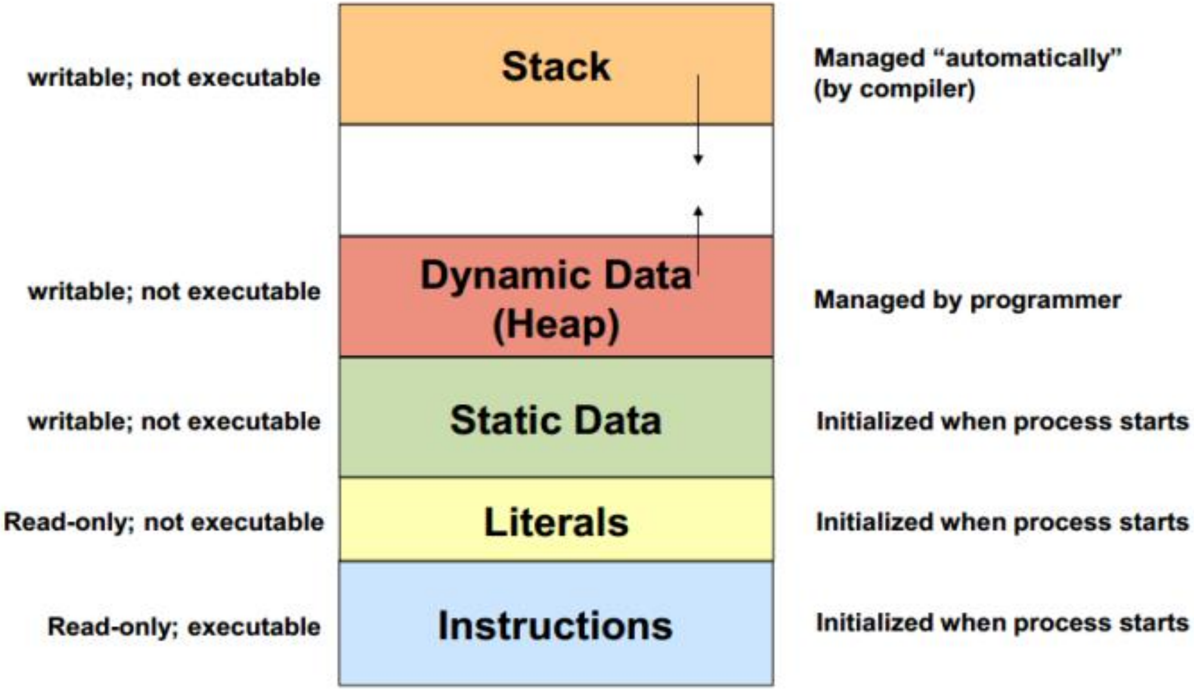
- Sistemul de operare schimba executia intre diferite taskuri
- Resursa comuna = CPU



- *Interleaving*
  - sunt mai multe taskuri active, dar doar unul se executa la un moment dat
- *Multitasking*:
  - SO ruleaza executii intretesute

Procese

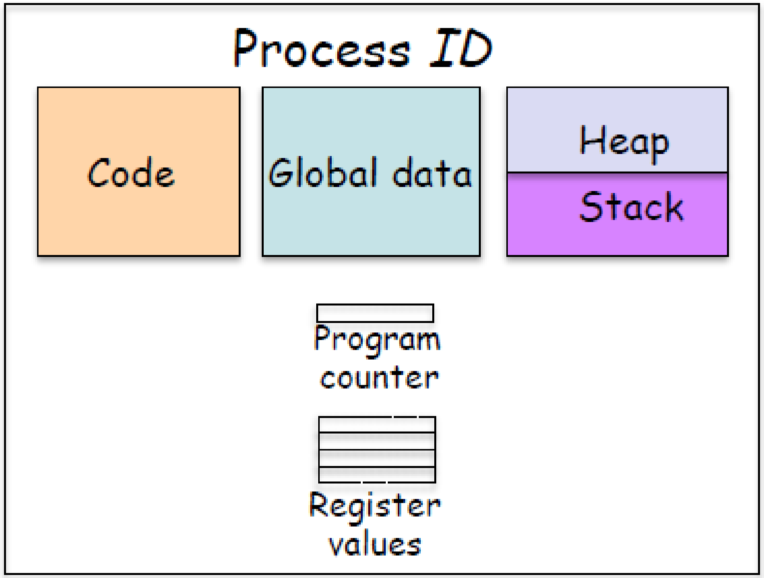
- Un program (secvential) = un set de instructiuni
  (in paradigma programarii imperative)
- Un proces = o instanta a unui program care se executa



Stack based allocation is **decided** at compile time and is **executed** at run time.

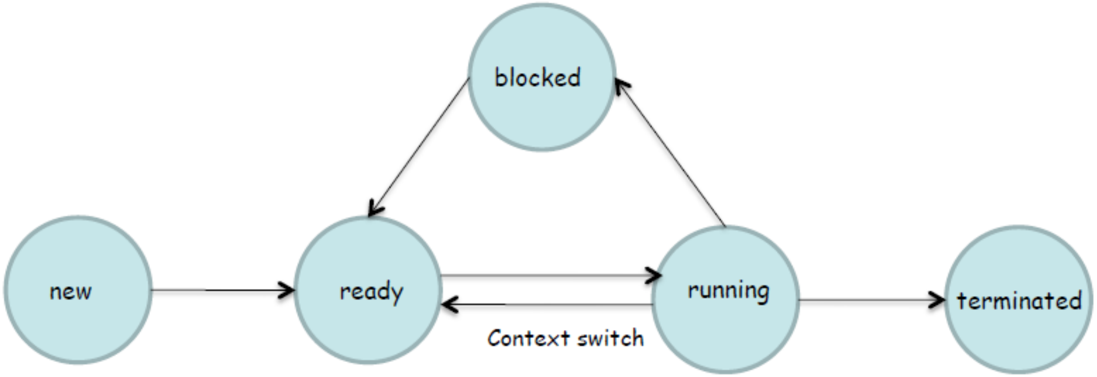
Procese in sisteme de operare

- Structura unui proces
  - Identificator de proces (ID)
  - Starea procesului (activitatea curenta)
  - Contextul procesului (valori registri, *program counter*)
  - Memorie (codul program, date globale/statice, stiva si heap)



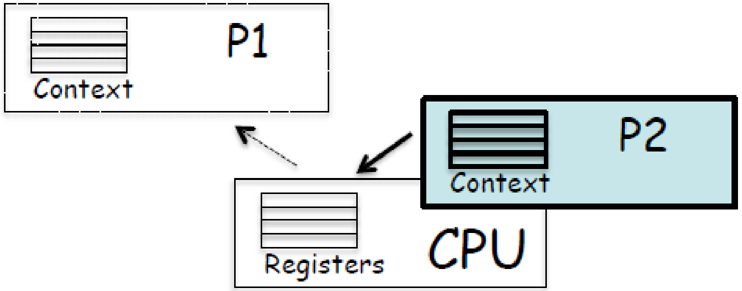
Scheduler

- Un program care controleaza executia proceselor
  - Seteaza starile procesului
    - new
    - running
    - blocked (nu poate fi selectat pt executie; este nevoie de un event extern pt a iesi din aceasta stare)
    - ready
    - terminated



Context switch

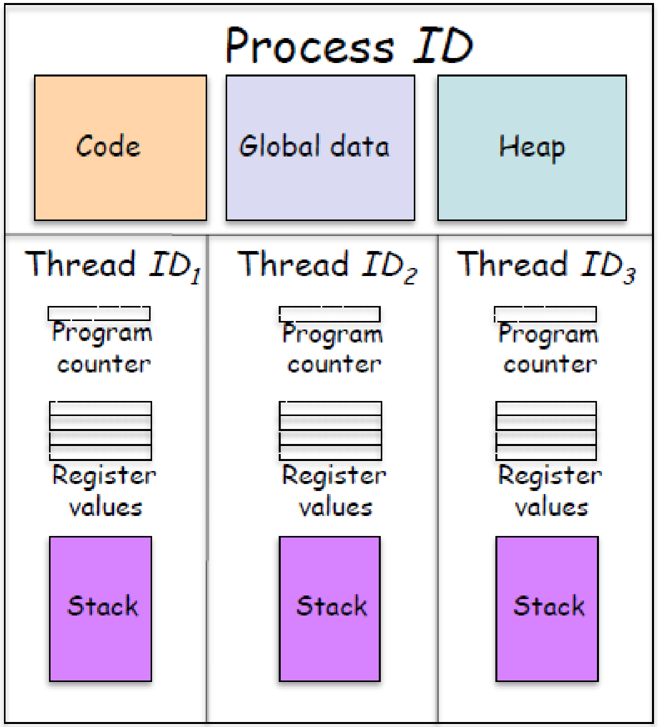
- Atunci cand *scheduler*-ul schimba procesul executat de o unitate de procesare



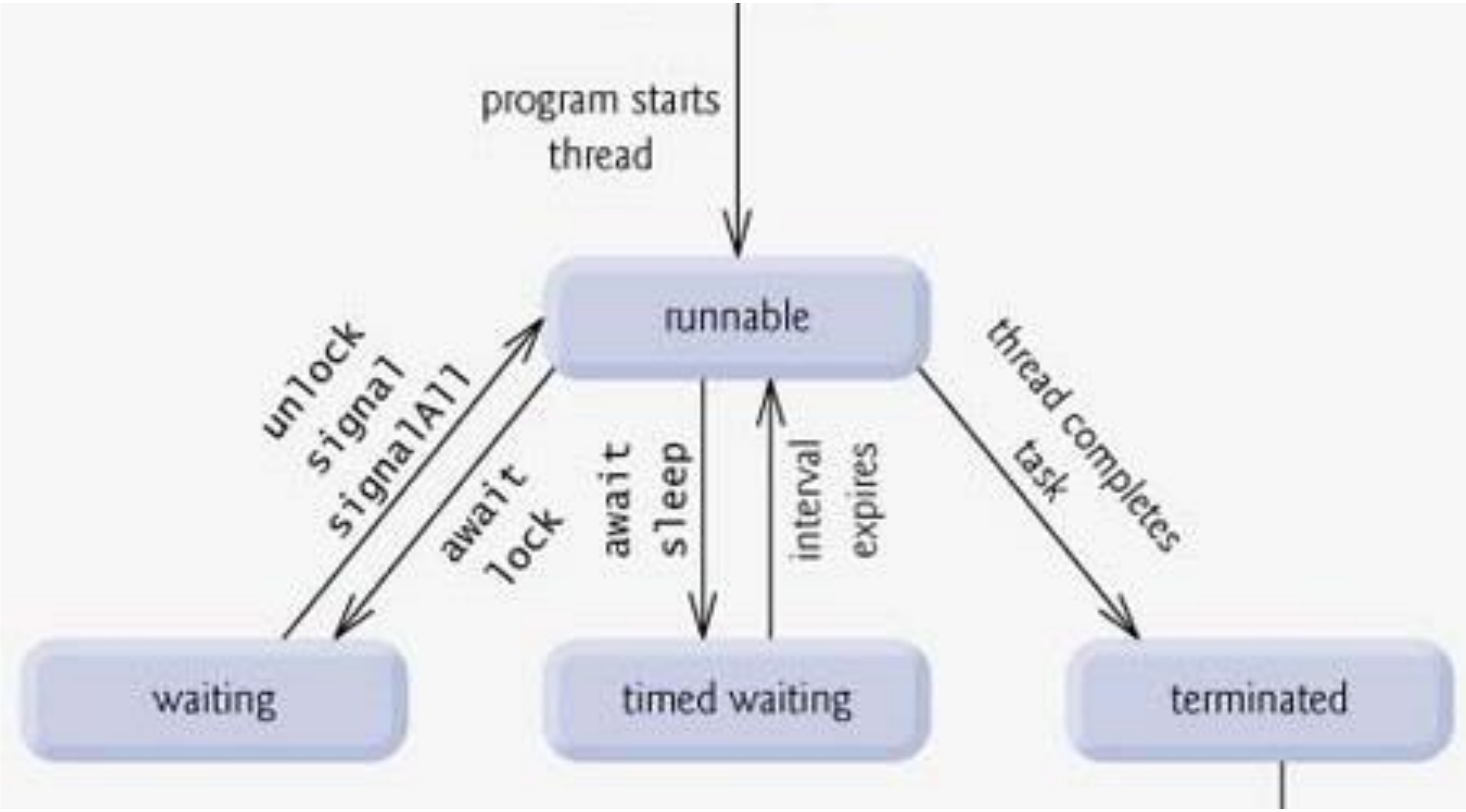
- Actiuni asociate:
  - P1.state := ready
  - Se salveaza valoarea registrilor in memorie dar si contextul lui P1
  - Se foloseste contextul lui P2 pt a seta registrii
  - P2.state := running
  - in plus -> switching memory address space:
    - include: memory addresses, page tables, kernel resources, caches

Threads

- Un thread este o parte a unui proces al sistemului de operare
- Componente private fiecarui thread:
  - Identificator
  - Stare
  - Context
  - Memorie (doar stiva)
- Componente partajate cu alte thread-uri
  - Cod program
  - Date globale
  - Heap



Threads


program starts thread
runnable
unlock
signal
signalAll
await
lock
await
sleep
interval
expires
thread completes task
waiting
timed waiting
terminated

Preemptive multitasking

- A preemptive multitasking operating system permits preemption of tasks.
- A cooperative multitasking operating system - processes or tasks must be explicitly programmed to yield when they do not need system resources.

- Preemptive multitasking involves the use of **an interrupt mechanism** which suspends the currently executing process and invokes a scheduler to determine which process should execute next.
  - Therefore, all processes will get some amount of CPU time at any given time.

- In preemptive multitasking, the operating system kernel can also initiate a context switch to satisfy the scheduling policy's priority constraint, thus preempting the active task.

- ***Non Preemptive threading model:*** Once a thread is started it cannot be stopped or the control cannot be transferred to other threads until the thread has completed its task.
- ***Preemptive Threading Model:*** The runtime is allowed to step in and hand control from one thread to another at any time. Higher priority threads are given precedence over Lower priority threads.

CPU use

- Processes could:
  - wait for input or output (called "**I/O bound**"),
  - utilize the CPU ("CPU bound").
- In early systems, processes would often "poll", or "busywait" while waiting for requested input (such as disk, keyboard or network input).
  - During this time, the process was not performing useful work, but still maintained complete control of the CPU.
- With the advent of interrupts and preemptive multitasking, these I/O bound processes could be "blocked", or **put on hold**, pending the arrival of the necessary data, allowing other processes to utilize the CPU.
- As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

Multitasking advantages

- Although multitasking techniques were originally developed to allow multiple users to share a single machine, multitasking is useful regardless of the number of users.

- Multitasking makes it possible for a single user to run multiple applications at the same time, or to run "background" processes while retaining control of the computer.

- ***Preemptive multitasking allows the computer system to more reliably guarantee each process a regular "slice" of operating time.***
  - It also allows the system to rapidly deal with important external events like incoming data, which might require the immediate attention of one or another process.

race condition

- **race condition (race hazard)**
- *a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes*

= *o conditie care poate sa apara intr-un system (electronic, software,...) in care **comportamentul este dependent de ordinea in care apar anumite evenimente** (necontrolate strict).*

> *-Daca exista una sau mai multe posibilitati de rezultat (comportament) => EROARE .*

= doua sau mai multe operatii cer ar fi executate in acelasi timp dar natura sistemului impune secventializarea lor. Daca nu este posibila orice secventializare atunci pot apare erori.

- Termenul *race condition* a aparut inainte de 1955,
  (e.g. David A. Huffman's doctoral thesis "The synthesis of sequential switching circuits", 1954)

critical race condition vs. non-critical race condition

- *critical race condition* = atunci cand ordinea in care se modifica variabilele interne determina starea finala a sistemului

- *non-critical race condition* = atunci cand ordinea in care se modifica variabilele interne NU are impact asupra starii finale a sistemului

T1:

```
update (){
    a=a+1;
    }
```

T2:

```
update (){
    b=a*2;
}
```

Data race

- **A data race occurs when two or more threads access the same variable concurrently, and at least one of the accesses is a write.**
- o situatie in care un thread executa o operatie prin care se incearca sa se acceseze o locatie de memorie care este in acelasi timp accesata pentru scriere de catre alt thread.

Deterministic versus non-deterministic computation

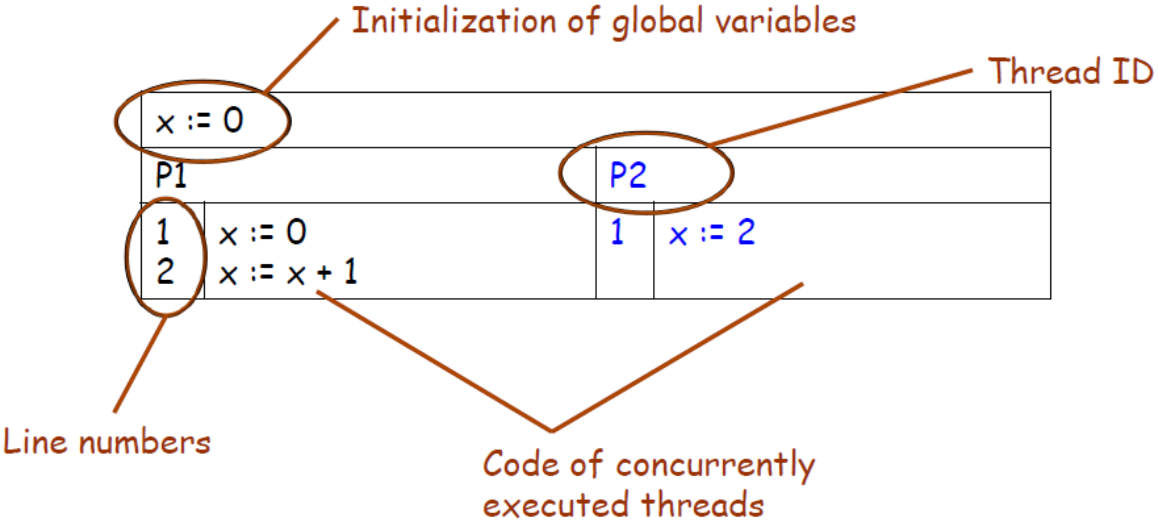
- Intr-o executie **determinista** ordinea de executie a operatiilor este complet determinata de specificatii/program
  - ex. program secvential

- Intr-o executie **NEdeterminista** ordinea de executie a operatiilor NU este complet determinata de specificatii/program –
  - ex. program parallel

- images from

Concurenta cu Threads

Un program care la executie conduce la un proces care contine mai multe threaduri



x := 0			
P1		P2	

1
2
	x := 0
	x := x + 1
		1	x := 2

Initialization of global variables

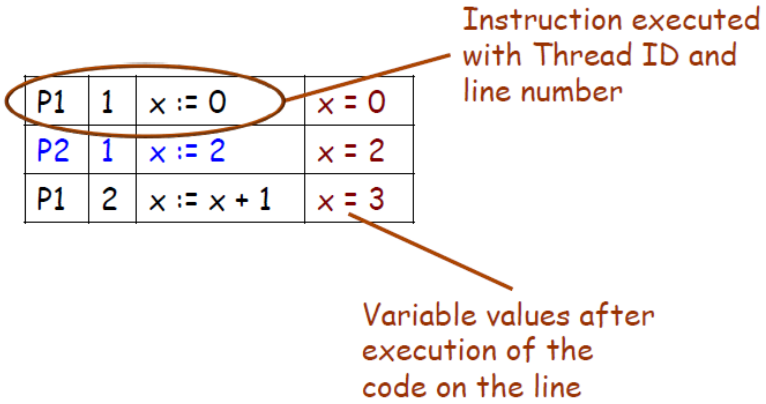
Thread ID

Line numbers

Code of concurrently executed threads

Variante de executie

- Secvente de executie



P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

Instruction executed with Thread ID and line number

Variable values after execution of the code on the line

P2	1	x := 2	x = 2
P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1
P2	1	x := 2	x = 2

Instructiuni atomice

- <instr> este atomica daca executia sa nu poate fi “interleaved” cu cea a altei instructiuni inainte de terminarea ei.

- Niveluri de atomicitate

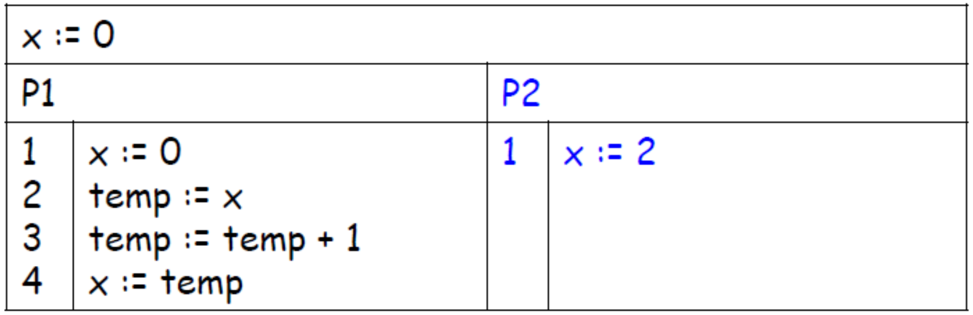
Ex: x:= x + 1

Executie:

```
temp := x                 LOAD REG, x
temp := temp + 1               ADD  REG, #1
x := temp                              STORE REG, x
```

Variante de executie

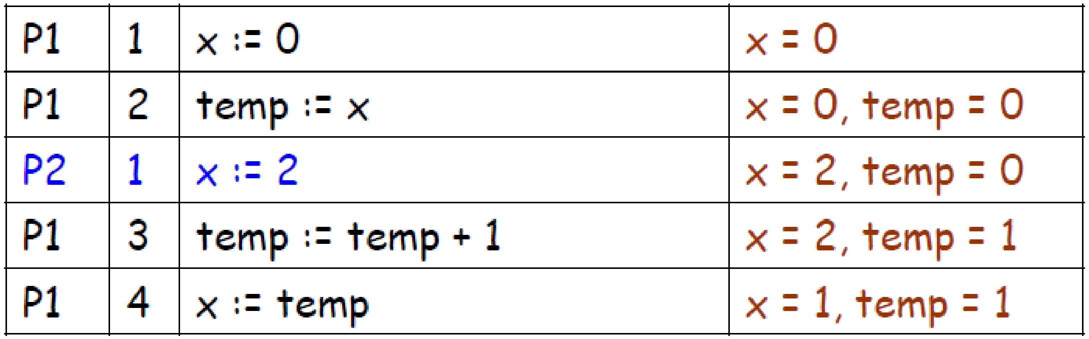
- exemplul anterior



x := 0			
P1		P2	

1
2
3
4
	x := 0
	temp := x
	temp := temp + 1
	x := temp
		1	x := 2

- o executie "interleaving"



P1	1	x := 0	x = 0
P1	2	temp := x	x = 0, temp = 0
P2	1	x := 2	x = 2, temp = 0
P1	3	temp := temp + 1	x = 2, temp = 1
P1	4	x := temp	x = 1, temp = 1

Exemplul -2

Două fire de execuţie decrementează variabila V până la 0

```
while (v>0)
     v--;
```

Thread T1

```
while (v>0)
     v--;
```

Thread T2

Ce valoare va avea variabila V după terminarea execuţiei celor două fire de execuţie?

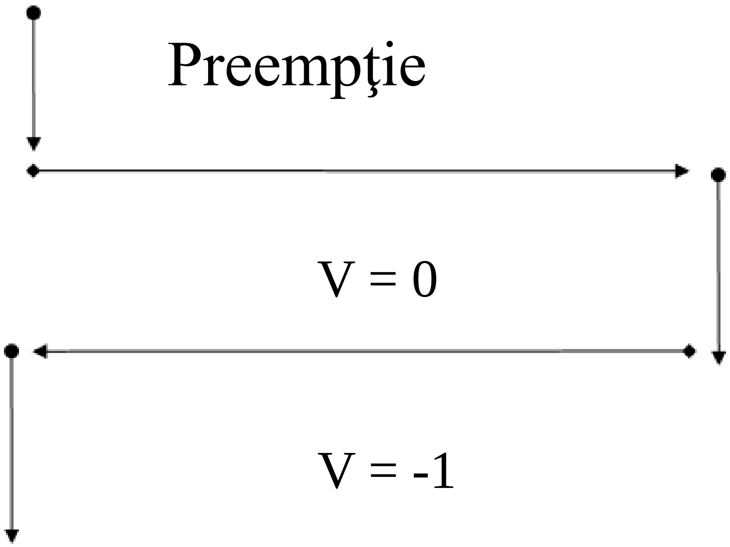
Exemplul 2- Varianta de executie

....

V = 1

Thread T1

Thread T2



```
If (v>0) {
```

Preempţie

```
If (v>0) {
      v--;
}
```

V = 0

```
    v--;
}
```

V = -1

Race condition & Critical section

- Procese / threaduri independente => executie simpla fara probleme
- Daca exista interactiune (ex. Accesarea si modificarea acelasi variabile) => pot apare probleme
- ***Nondeterministic interleaving***
- Daca rezultatul depinde de interleaving => *race condition*
- Se incearca sa se foloseasca **aceeasi resursa** si ordinea in care este folosita este importanta!
- Pot fi erori extrem de greu de depistat!!!

Race Conditions & Critical Sections

- *A **Critical Section*** = code segment where a data-race may occur

```
public class Counter {
    protected long count = 0;
    public void add(long value){
        this.count = this.count + value;
    }
  }
```

Daca un obiect de tip Counter este folosit de 2 sau mai multe threaduri!

=> Nu e *thread-safe!*

- Metoda add() este un exemplu de sectiune critica.

Counter -> Detaliere la nivel de registrii

- Codul nu este executat ca si o instructiune atomica:

```
get this.count from memory into register
add value to register
write register to memory
```

- Exemplu de intretesere

```
this.count = 0;
A:  reads this.count into a register (0)
B:  reads this.count into a register (0)
B:  adds value 2 to register
B:  writes register value (2) back to memory. this.count now equals 2
A:  adds value 3 to register
A:  writes register value (3) back to memory. this.count now equals 3
```

Solutii

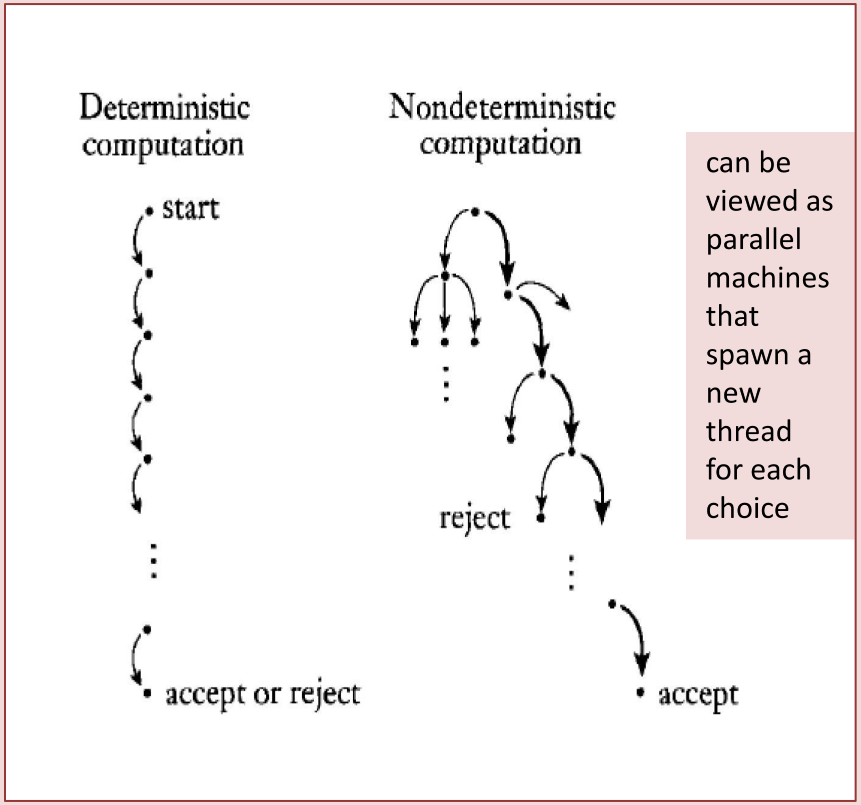
Necesar ->Accesul la date în sectiune critică făcut într-un mod ordonat si atomic, astfel încât rezultatele să fie predictibile

- Soluţii:
  - atomicizarea sectiunii critice
  - dezactivarea preempţiei în sectiunea critică
  - secvenţializarea accesului la sectiunea critică

Deterministic versus non-deterministic computation

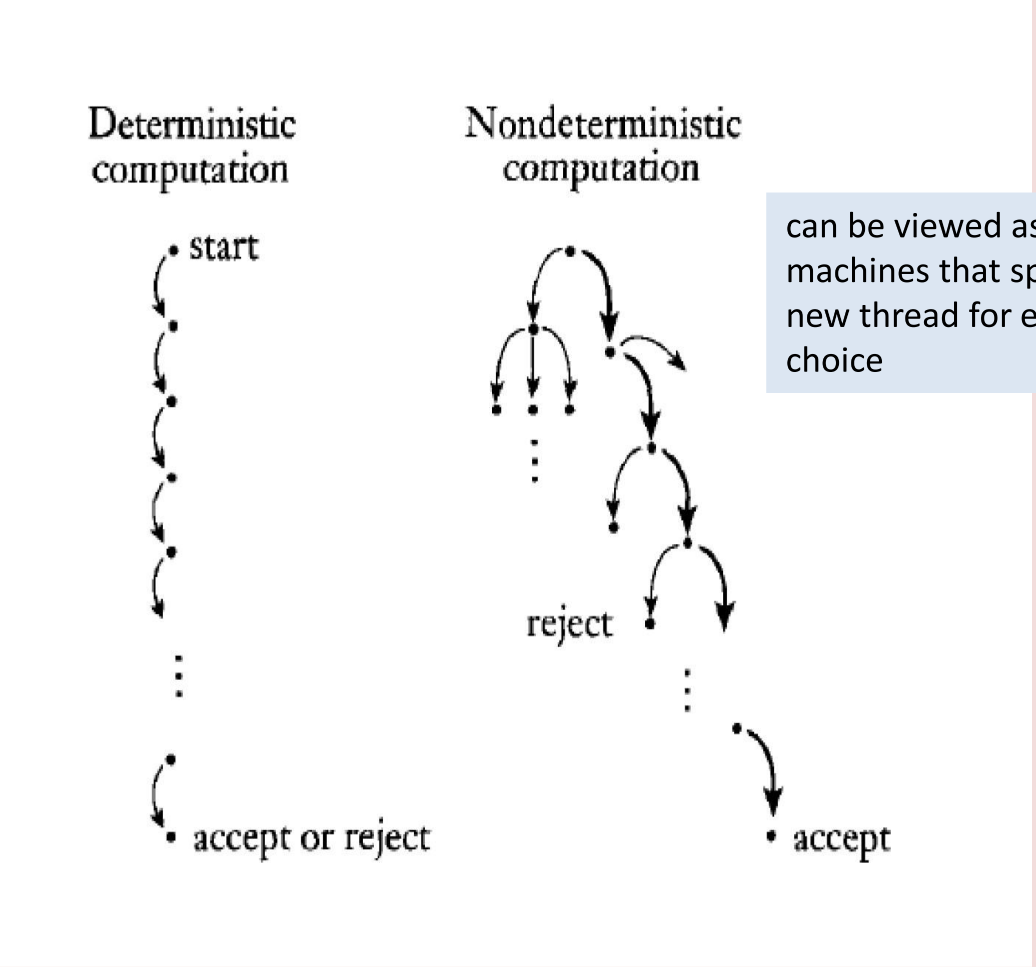
- Determinism (Computer Science)
- *A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.*

- Nondeterminism (Computer Science)
- *A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.*



can be viewed as parallel machines that spawn a new thread for each choice

Deterministic versus non-deterministic computation

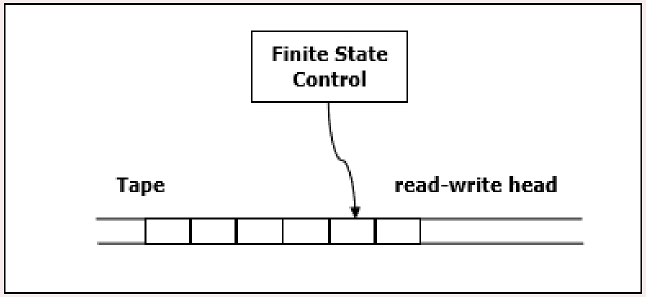


can be viewed as parallel machines that spawn a new thread for each choice

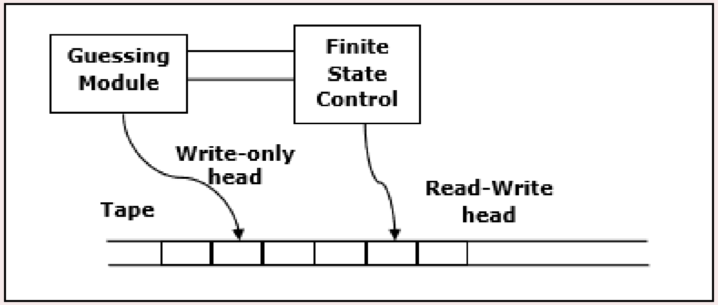
Turing machines

https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_deterministic_vs_nondeterministic_computations.htm

- **Deterministic Turing Machine** - consists of a finite state control, a read-write head and a two-way tape with infinite sequence.
- A program for a deterministic Turing machine specifies the following information −
  - A finite set of tape symbols (input symbols and a blank symbol)
  - A finite set of states
  - A transition function
- In algorithmic analysis,
  - if a problem is solvable in polynomial time by a deterministic one tape Turing machine, the problem belongs to P class.



- **Nondeterministic Turing Machine -**one additional module known as the **guessing module**, which is associated with one write-only head.
  - If the problem is solvable in polynomial time by a non-deterministic Turing machine, the problem belongs to NP class.


Guessing Module
Finite State Control
Write-only head
Tape
Read-Write head