Curs 3

Programare Paralela si Distribuita

Paralelism implicit versus paralelism explicit

Procese versus Fire de executie

Operatii atomice

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

<u>Implicit Parallelism: Parallelizing Compilers</u>

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

<u>Se poate obtine paralelizare dar nu completa si impune analize foarte dificile!</u>

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

Analiza generala a caracteristicilor

Main Features	Data-Parallel	Message-Passing	Shared-Variable
Control flow (threading)	Single	Multiple	Multiple
Synchrony	Loosely synchronous	Asynchronous	Asynchronous
Address space	Single	Multiple	Multiple
Interaction	Implicit	Explicit	Explicit
Data allocation	Implicit or semiexplicit	Explicit	Implicit or semiexplicit

Legatura Modele 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;

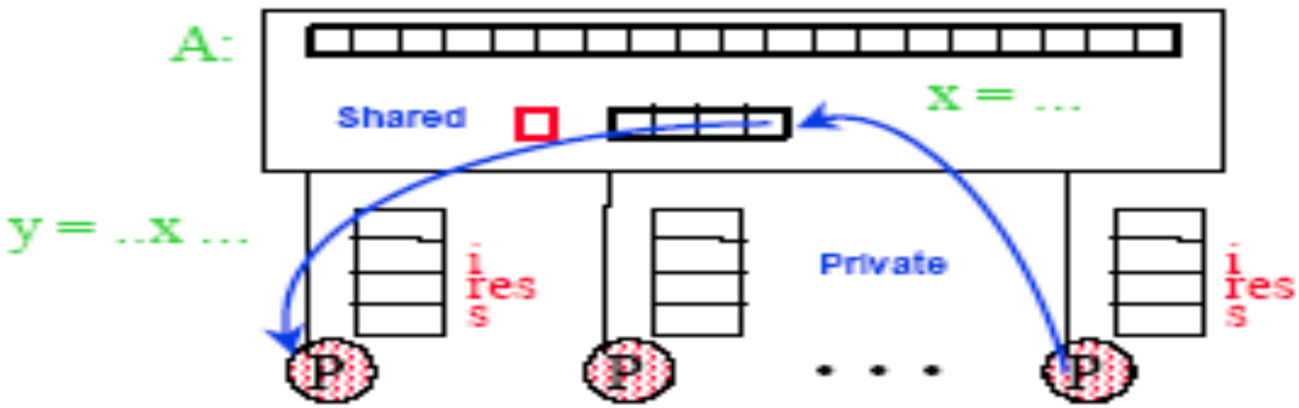
-private: evalurile individuale de functii si sumele partiale

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.

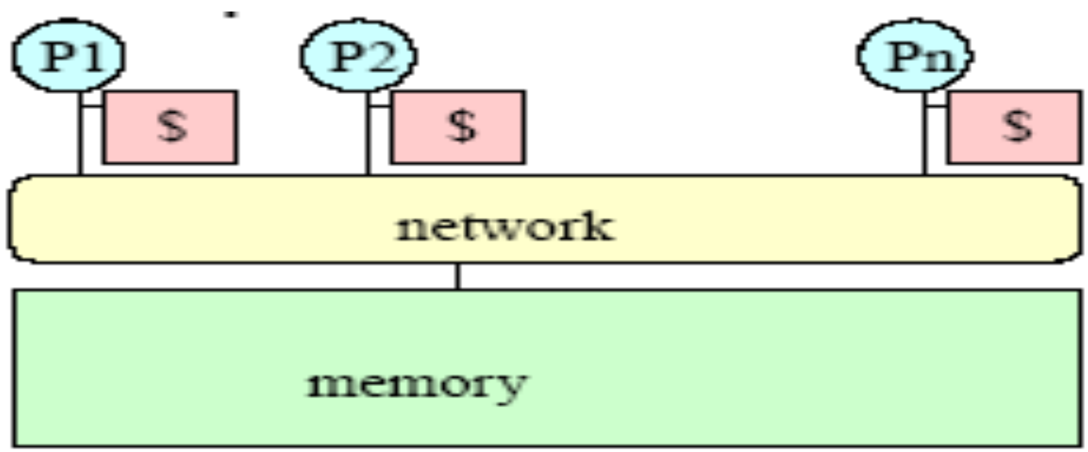
Comunicatia dintre firele de executie: prin citirea/scrierea variabilelor partajate.

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



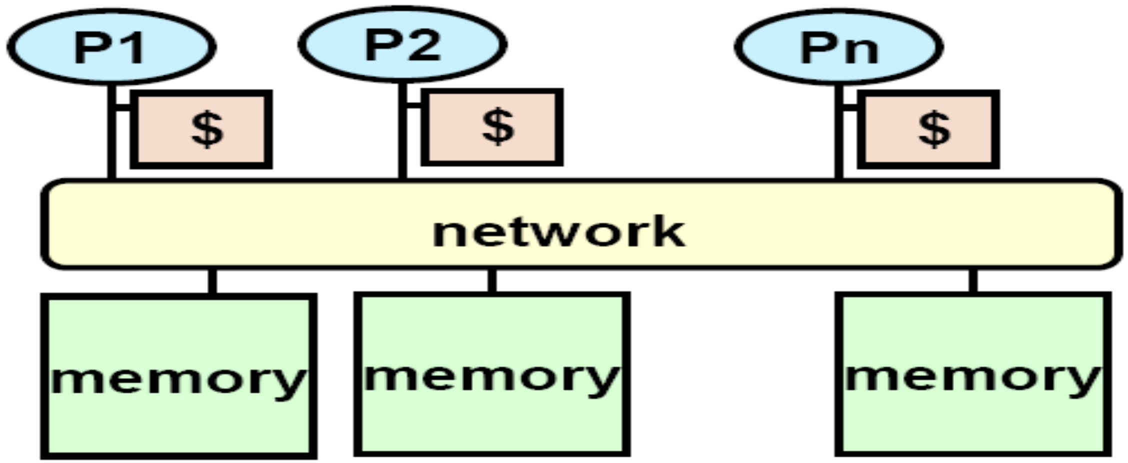
Masina paralela corespunzatoare modelului 1: masina cu memorie partajata (sistemele multiprocesor, sistemele multiprocesor simetrice sau SMP-Symmetric Multiprocessors).

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.

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):

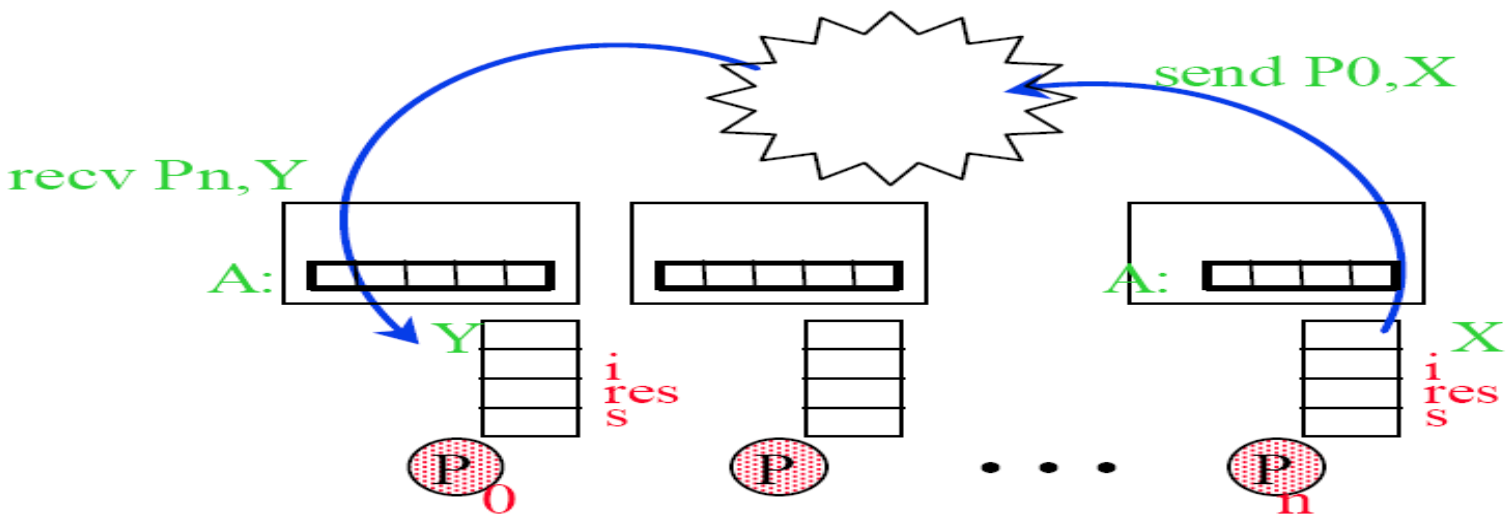
Thread 1	Thread 2
lock	lock
load s	load s
s = s+local_s1	s = s+local_s2
store s	store s
unlock	unlock

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.

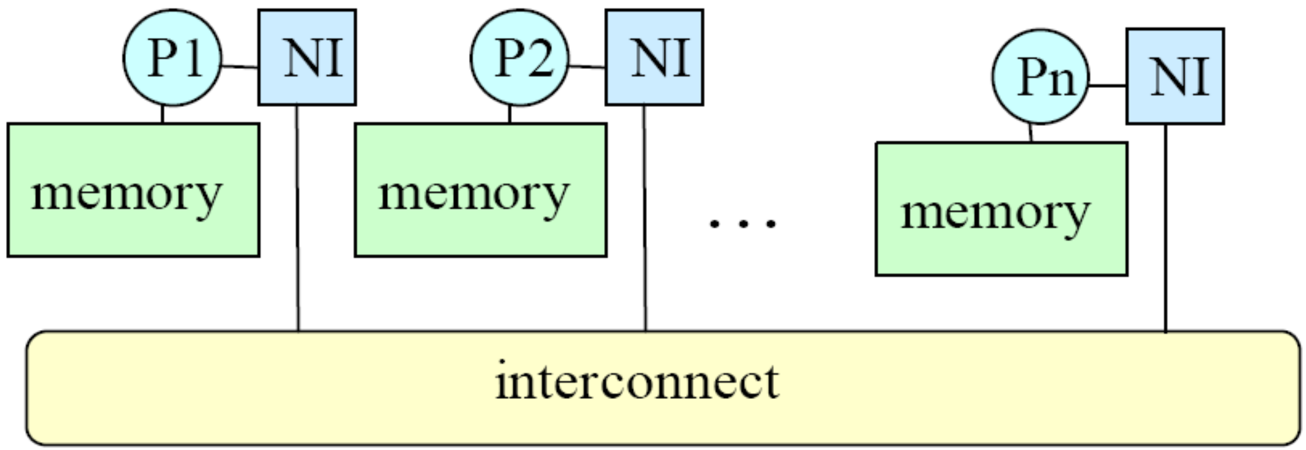
Comunicatia dintre procese: prin transfer explicit de date (perechi de operatii corespunzatoare **send** si **receive** la procesele sursa si respectiv destinatie). Datele partajate din punct de vedere logic sunt partitionate intre toate procesele.

=> asemanare cu programarea distribuita!



Exista biblioteci standard (exemplu: MPI si PVM).

Masina corespunzatoare modelului 2 este multicalculatorul (sistem cu memorie distribuita):



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])*):

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

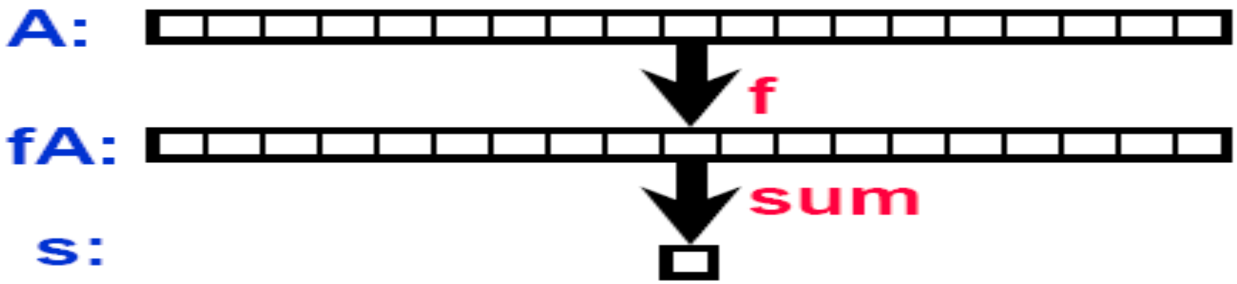
```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

Procesor 2

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

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

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

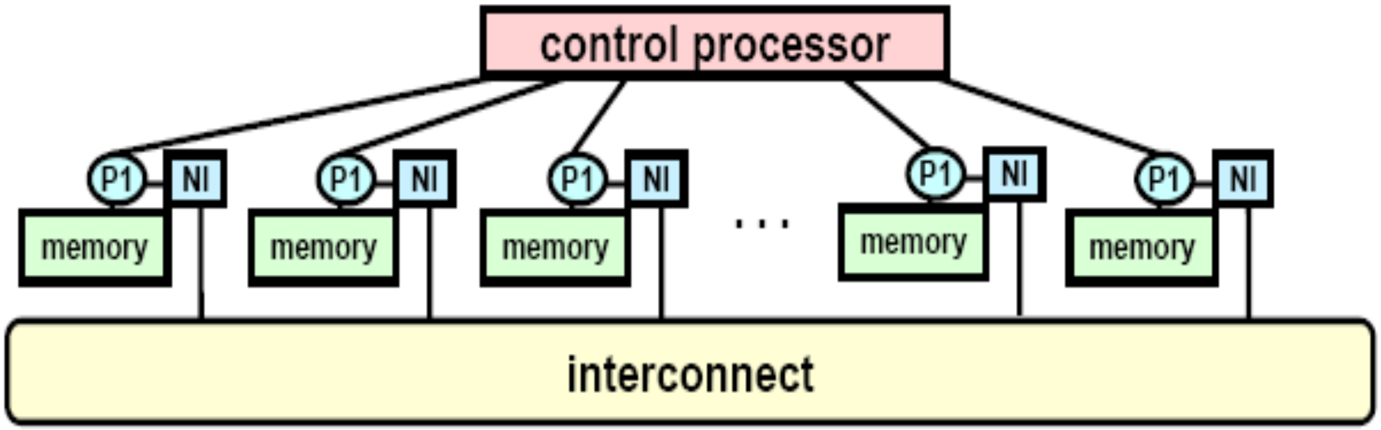


map + reduce

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).

4) Modelul 4 de masina: 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.

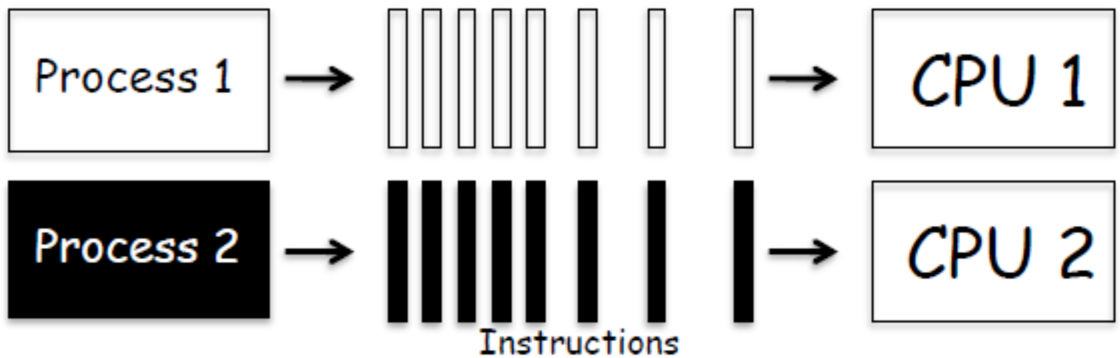
Exemple: Millennium, IBM SPx, ASCI Red (Intel).

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*

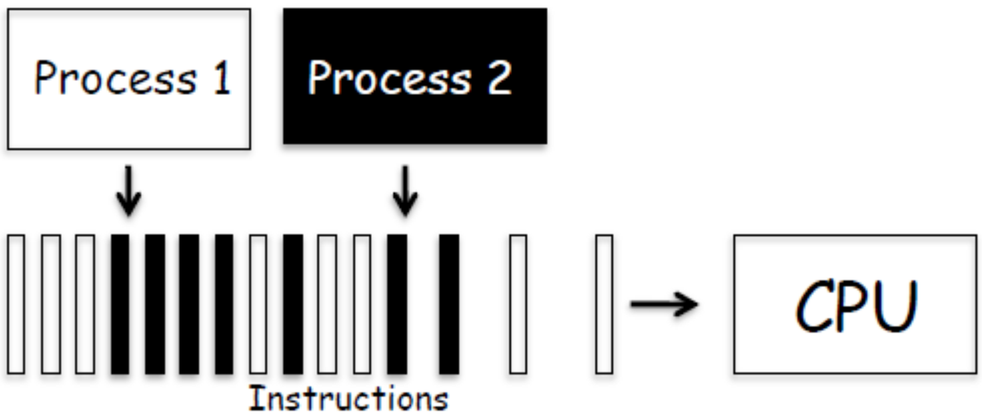


- Se folosesc mai multe unitati de procesare

=>Procesele se pot executa in acelasi timp

Multitasking-> Concurenta

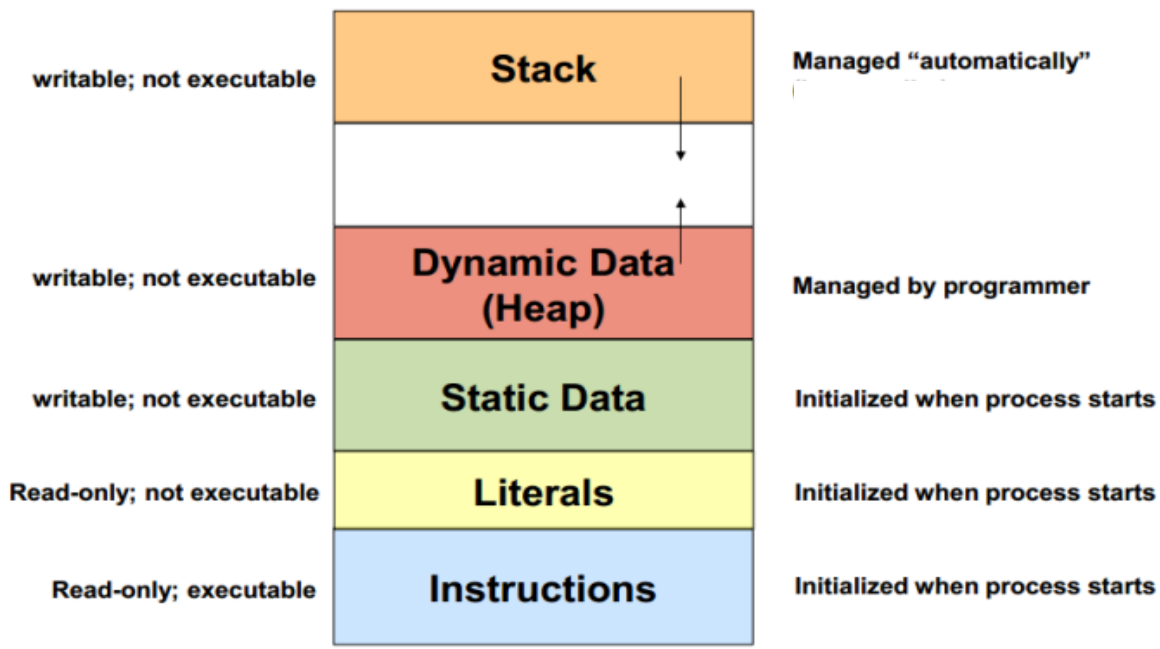
- Sistemul de operare schimba executia intre diferite taskuri
- Resursa comuna = CPU



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

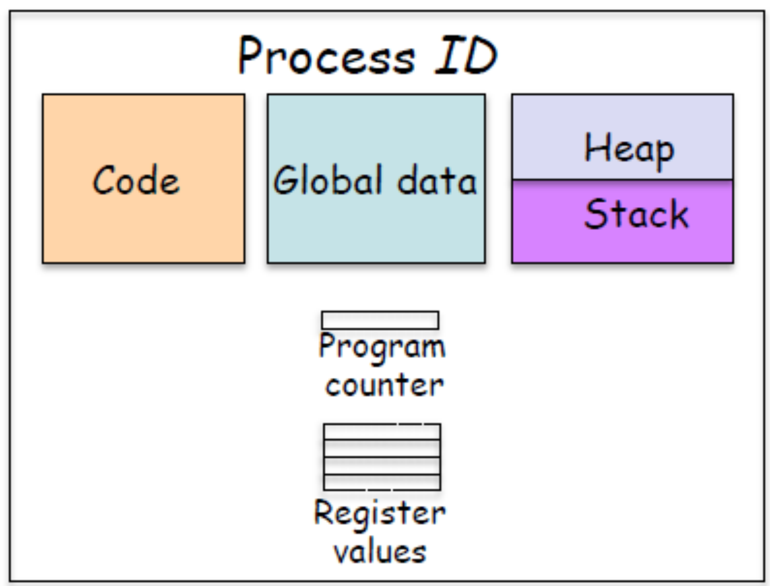
Procese

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



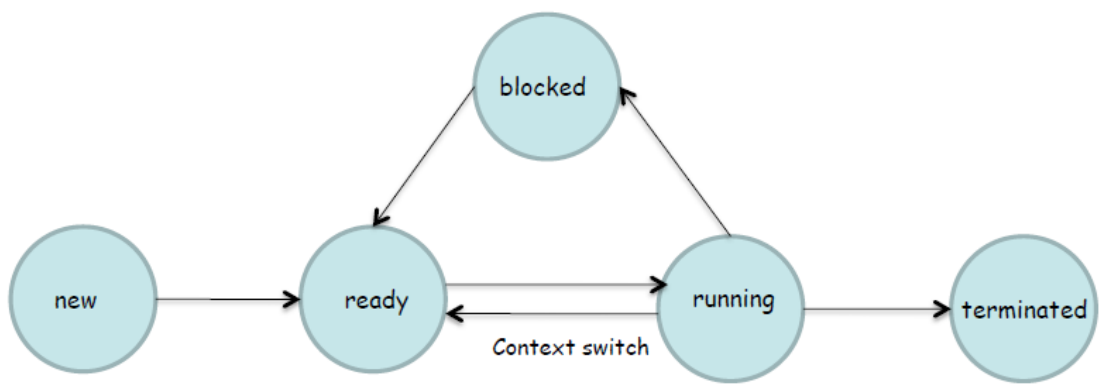
Procese in sisteme de operare

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



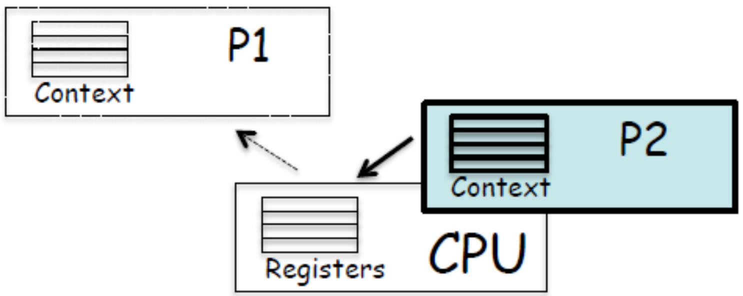
Scheduler

- 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

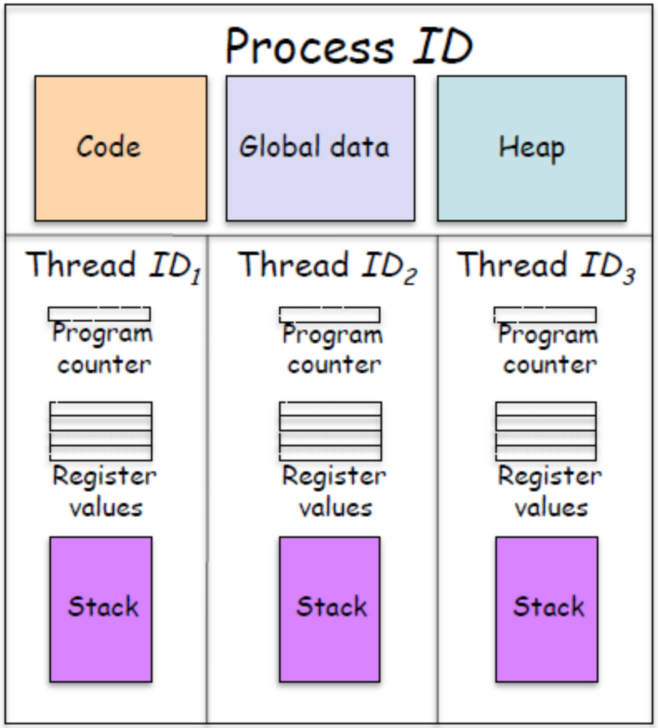
- Atunci cand *scheduler*-ul schimba procesul executat de o unitate de procesare



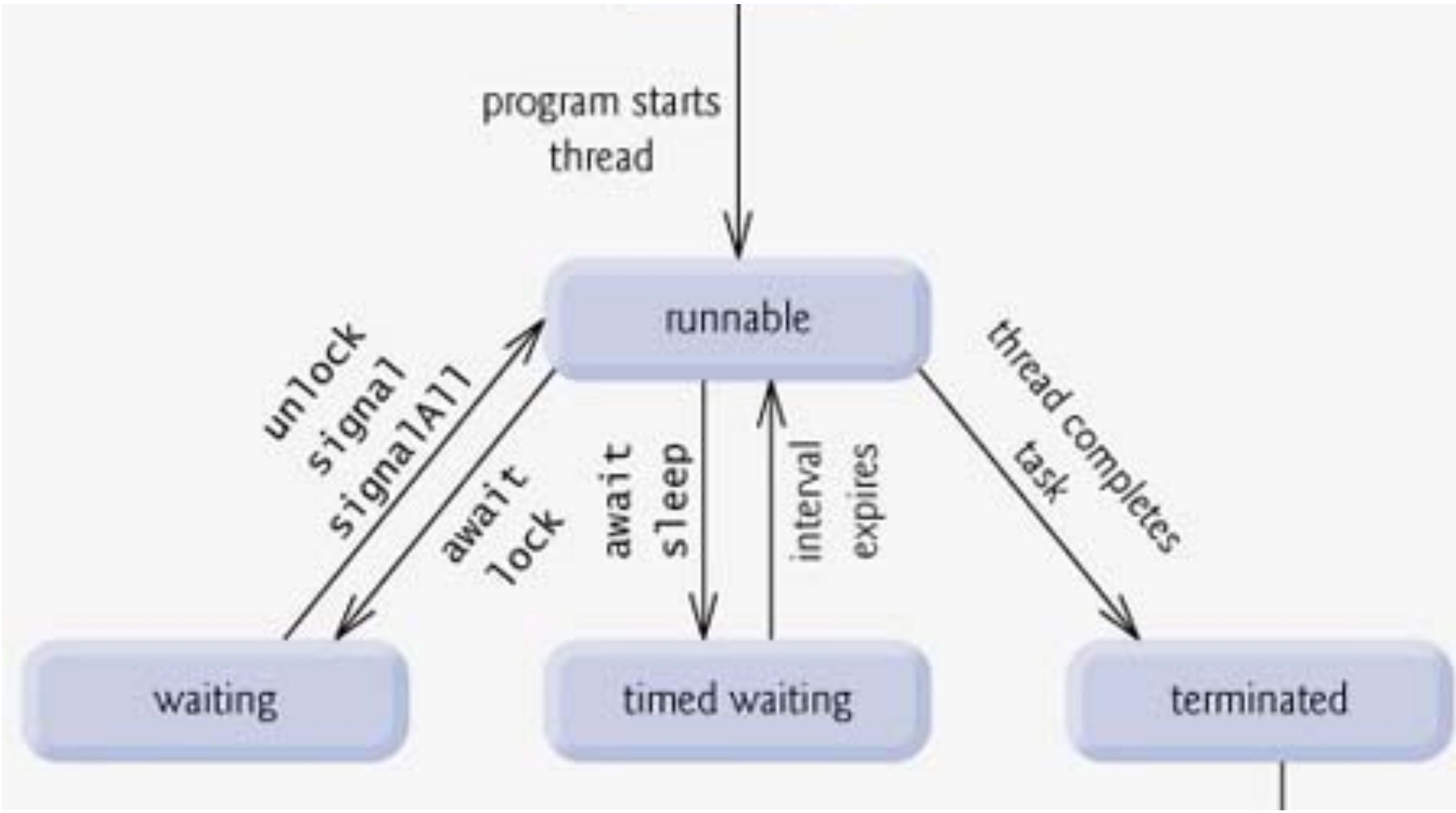
- Actiuni asociate:
  – P1.state := ready
  – Se salveaza valoarea registrilor in memorie ca si context al lui P1
  – Se foloseste contextul lui P2 pt a seta registrii
  – P2.state := running

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.

CPU use

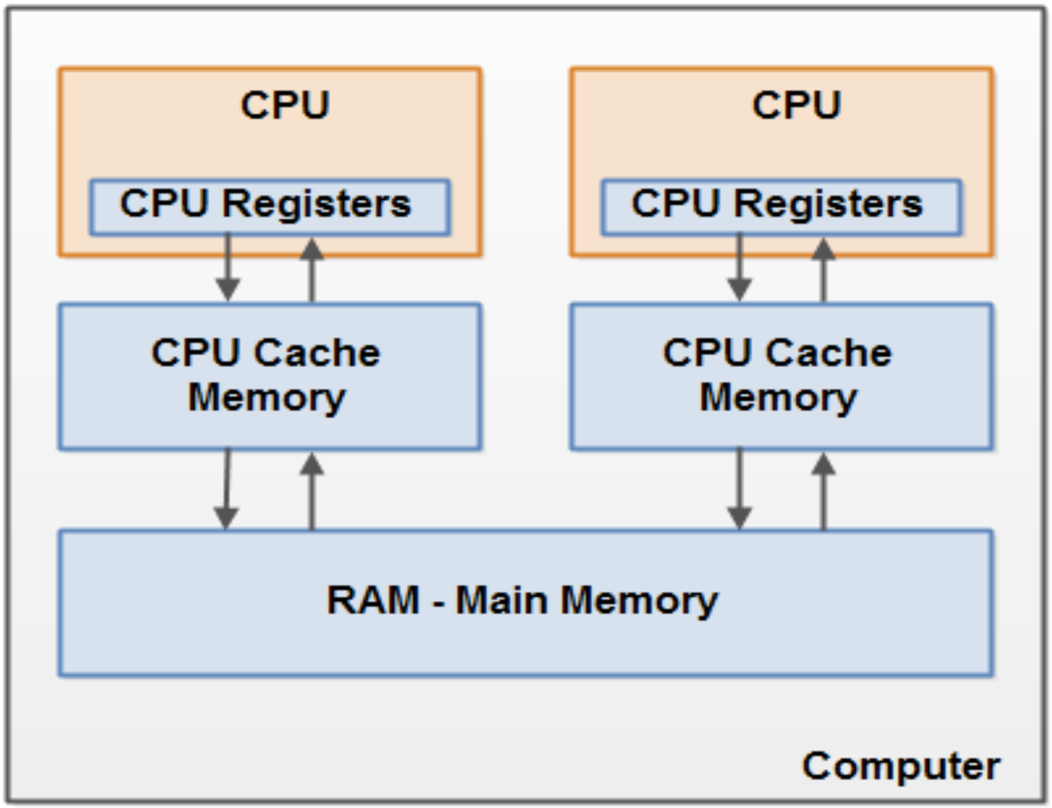
- Processes could:
  - wait for input or output (called "I/O bound"),
  - utilize the CPU ("CPU bound").
- In early systems, processes would often "poll", or "busywait" while waiting for requested input (such as disk, keyboard or network input).
  - During this time, the process was not performing useful 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.

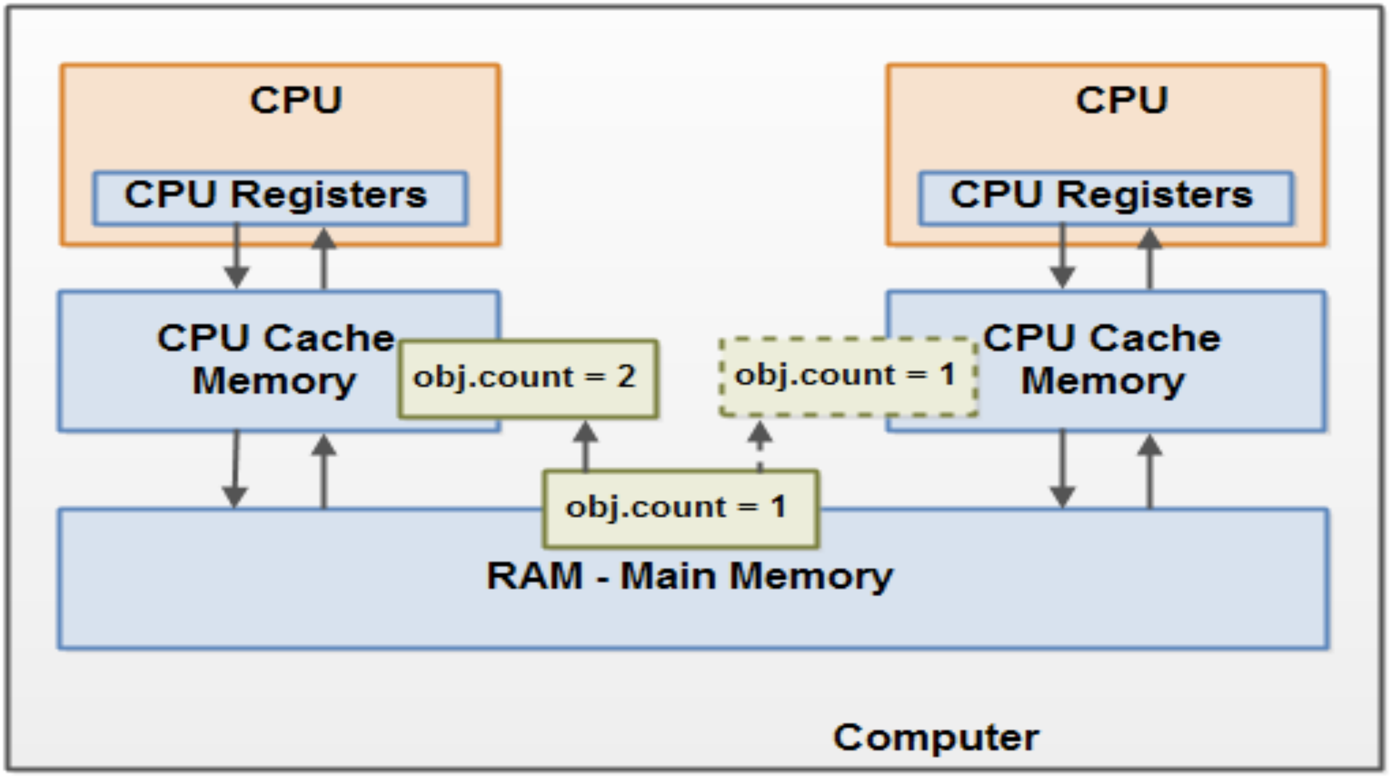
Multitasking advantages

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

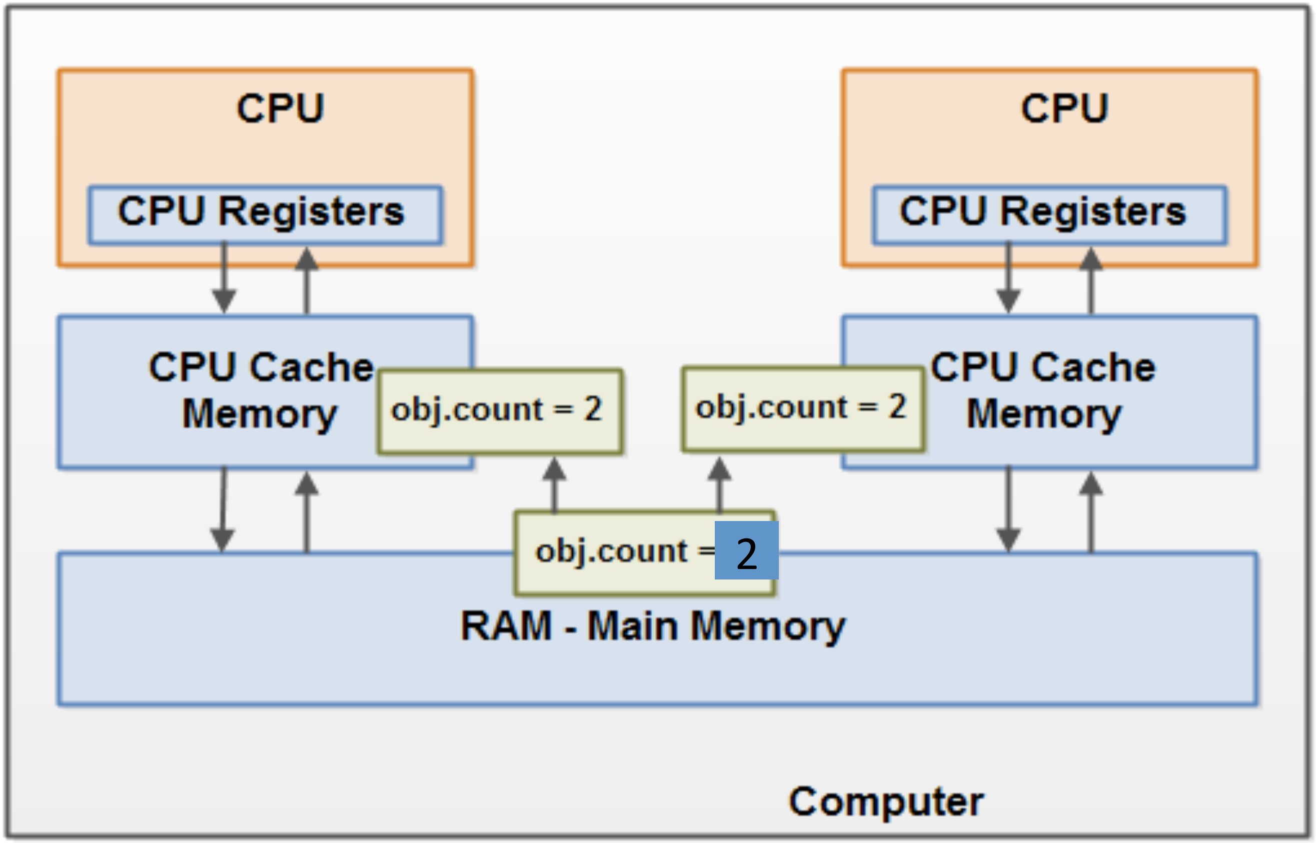
- Multitasking makes it possible for a single user to run multiple applications at the same time, or to run "background" processes while retaining control of the computer.

- 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.


CPU
CPU Registers
CPU
CPU Registers
CPU Cache Memory
CPU Cache Memory
RAM - Main Memory
Computer

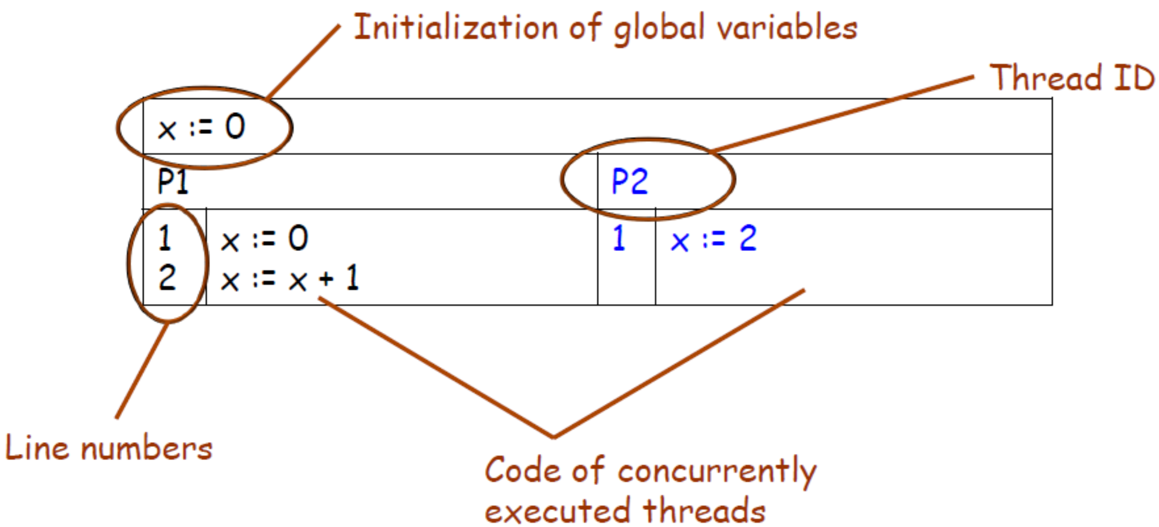

CPU
CPU Registers
CPU Cache Memory
obj.count = 2
CPU
CPU Registers
CPU Cache Memory
obj.count = 1
obj.count = 1
RAM - Main Memory
Computer

Race conditions



Concurenta cu Threads

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



Variante de executie

- Secvente de executie

Instruction executed with Thread ID and line number

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

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

Preempţie

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

```
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 (data race)*
  - 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

- O sectiune de cod care conduce la ***race conditions*** se numeste ***critical section (sectiune critica).***

```
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 care conduce la race conditions.

Solutii

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

- Soluţii:
  - atomicizarea zonei critice
  - dezactivarea preempţiei în zona critică
  - secvenţializarea accesului la zona critică

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

Thread Safety si *Shared Resources*

- Codul care poate fi apelat simultat de mai multe threaduri si produce intotdeauna rezultatul dorit/asteptat se numeste ***thread safe.***

- Daca o bucata de cod este *thread safe* atunci nu contine *race conditions*.

- *Race condition* apare doar atunci cand mai multe threaduri actualizeaza resurse partajate.
  - care sunt acestea….?

Variabile Locale

- Sunt stocate pe stiva de executie a fiecarui thread.
- Prin urmare nu sunt niciodata partajate.
  - => *thread safe.*

```
public void someMethod(){
  long threadSafeInt = 0;
  threadSafeInt++;
}
```

Local Object References

- Referinta poate fi locala si nu este partajata.
- Obiectul referit poate fi partajat – *shared heap*
- Daca obiectul este folosit doar in interiorul threadului care il defineste => *thread safe*

```
public void someMethod(){
  LocalObject localObject = new LocalObject();
  localObject.callMethod();
  method2(localObject);
}
public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

Exemplu: not thread safe

```
public class NotThreadSafe{

    StringBuilder builder =
        new StringBuilder();

    public void add(String text){
        this.builder.append(text);
    }

public static void main(String[]a){

NotThreadSafe sharedInstance =
    new NotThreadSafe();

new Thread(new
    MyRunnable(sharedInstance)).start();
new Thread(new
    MyRunnable(sharedInstance)).start();



}
```

```
public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;

  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }

  public void run(){
    this.instance.add("text LUNG");
  }
}
```

Thread Control Escape Rule

- Daca o resursa este creata, folosita si eliminata in interiorul controlului aceluiasi thread atunci folosirea acelei resurse este *thread safe*.

Operatii atomice

- Operaţii cu întregi:
  - incrementare, decrementare, adunare, scădere
  - **compare-and-swap** (**CAS**) operatii

CAS – instructiune ***atomica*** care compara continutul memoriei cu o valoare data si doar daca aceastea sunt egala modifica continutul locatiei de memorie cu noua valoare data.

The value of CAS is that it is implemented in hardware and is extremely lightweight (on most processors).

Compare and swap (CAS)

http://www.ibm.com/developerworks/library/j-jtp11234/

- The first processors that supported concurrency provided atomic test-and-set operations, which generally operated on a single bit.

- The most common approach taken by current processors, including Intel and Sparc processors, is to implement a primitive called *compare-and-swap*, or CAS.

- On Intel processors, compare-and-swap is implemented by the cmpxchg family of instructions.
- PowerPC processors have a pair of instructions called "load and reserve" and "store conditional" that accomplish the same goal; similar for MIPS, except the first is called "load linked.”

CAS operation

- A CAS operation includes three operands
  - a memory location (V),
  - the expected old value (A), and
  - a new value (B).
- The processor will atomically update the location to the new value(V==B) if the value that is there matches the expected old value (V==A), otherwise it will do nothing.
- In either case, it returns the value that was at that location prior to the CAS instruction.

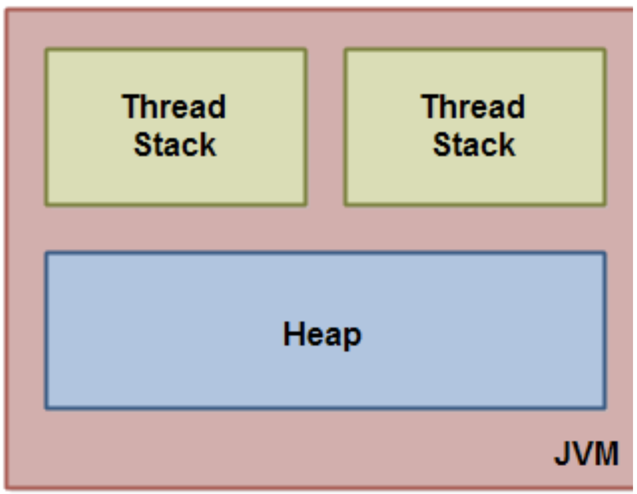
CAS synchronization

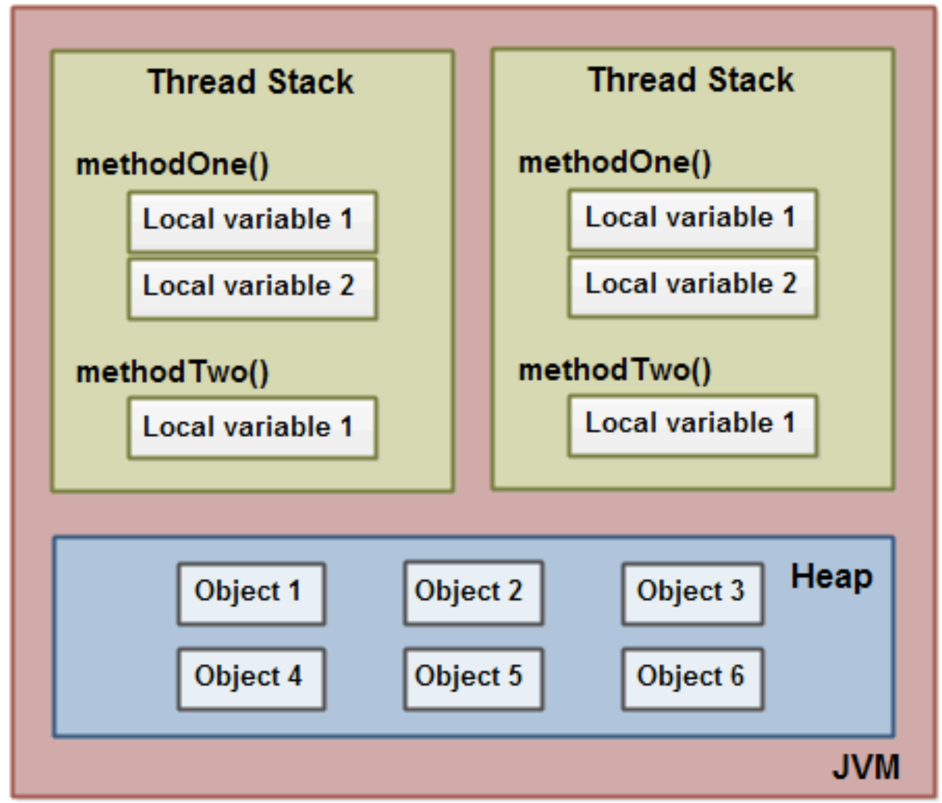
- The natural way to use CAS for synchronization is to read a value A from an address V, perform a multistep computation to derive a new value B, and then use CAS to change the value of V from A to B.
  - The CAS succeeds if the value at V has not been changed in the meantime.

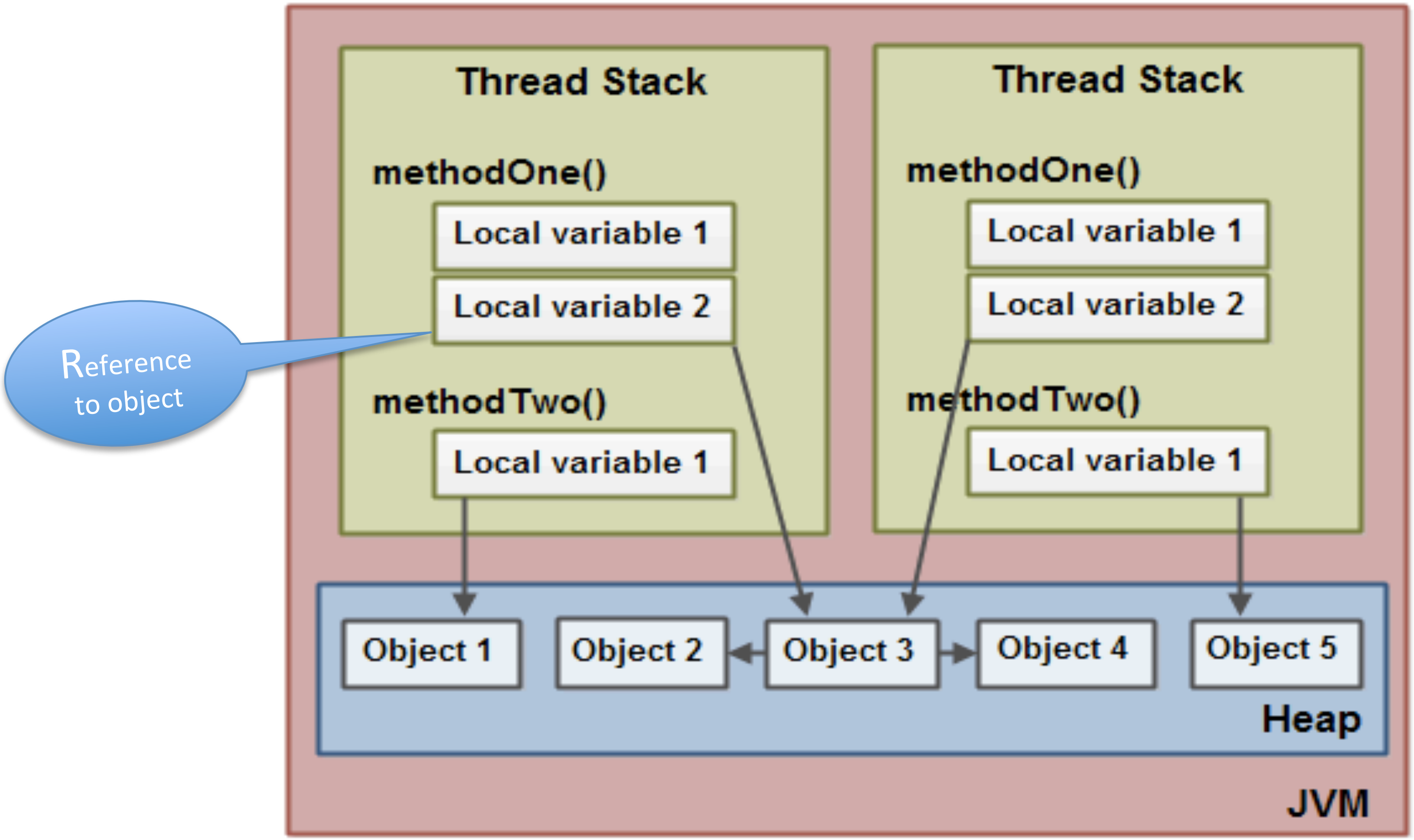
- Instructions like CAS allow an algorithm to execute a read-modify-write sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation.

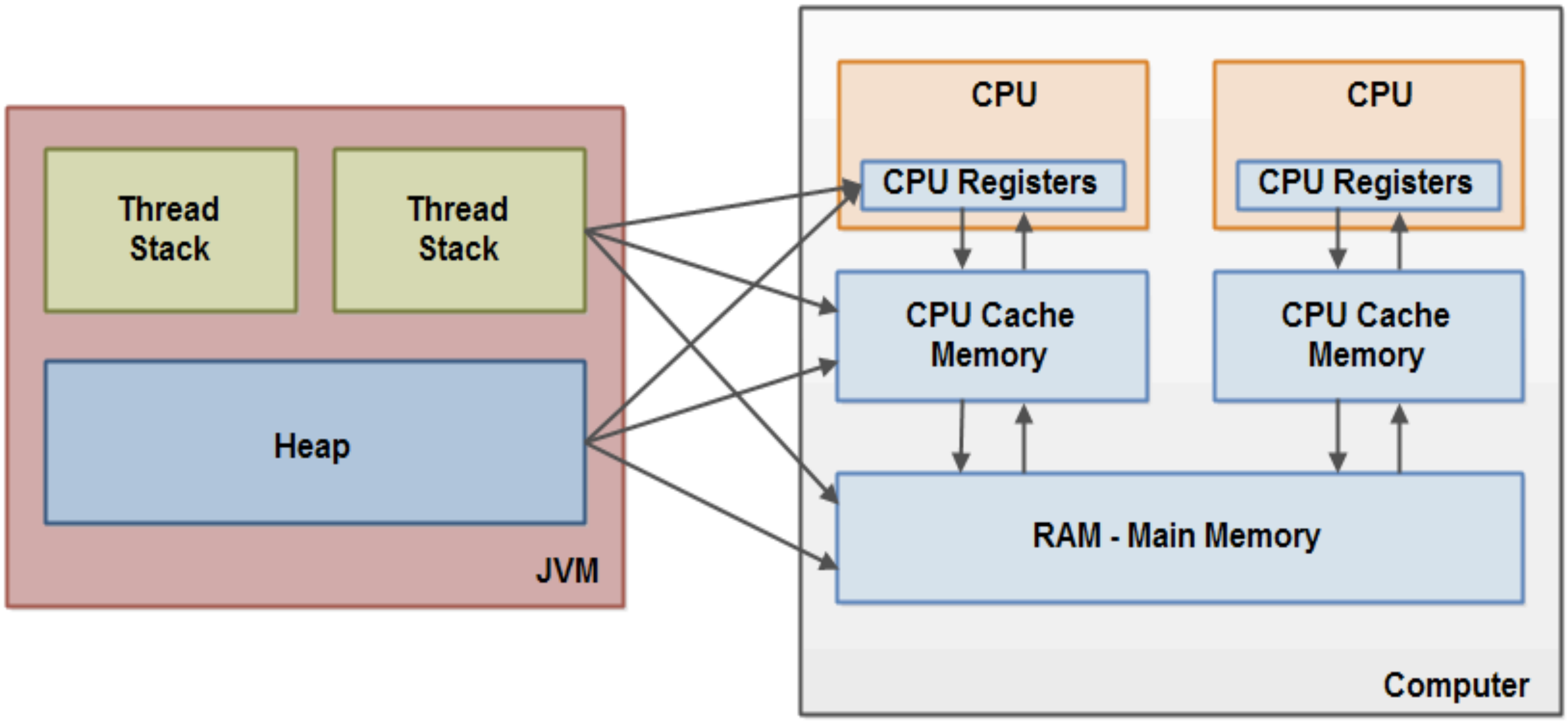
JAVA

Images from Jencov.com






Reference to object
Thread Stack
methodOne()
Local variable 1
Local variable 2
methodTwo()
Local variable 1
Thread Stack
methodOne()
Local variable 1
Local variable 2
methodTwo()
Local variable 1
Object 1
Object 2
Object 3
Object 4
Object 5
Heap
JVM


Thread
Stack
Thread
Stack
Heap
JVM
CPU
CPU Registers
CPU
CPU Registers
CPU Cache
Memory
CPU Cache
Memory
RAM - Main Memory
Computer

Java volatile

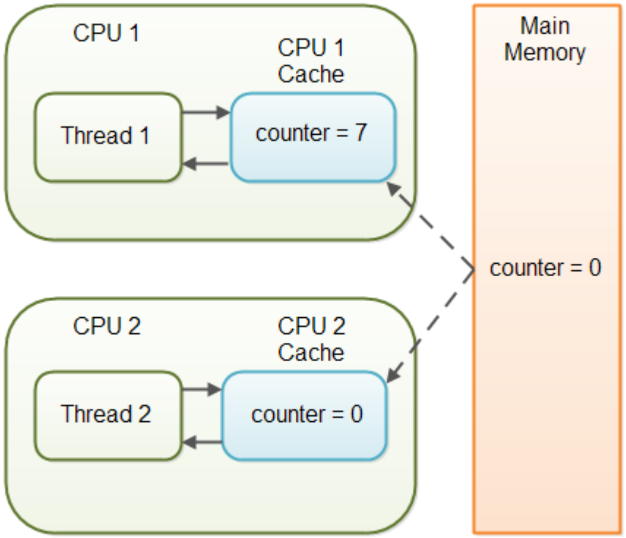
- Volatile indica faptul ca valoare unei variabila va fi modificata de catre mai multe threaduri.

- Informatii necesare pt JVM:

  -nu trebuie reordonate instructiunile corespunzatoare

- Valoarea nu va fi **cached thread-locally**:

  ⇒∀ reads & writes direct in "main memory";



http://tutorials.jenkov.com/images/java-concurrency/java-volatile-2.png

The main differences between synchronized and volatile

- a primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized);

- an access to a volatile variable **never has the potential to block**: we're only ever doing a simple read or write, so unlike a synchronized block we will never hold on to any lock;

- because accessing a volatile variable **never holds a lock**, it is **not suitable** for cases where we want to ***read-update-write*** as an atomic operation (unless we're prepared to "miss an update");

- a volatile variable that is an object reference may be null (because you're effectively synchronizing on the *reference*, not the actual object).

(Attempting to synchronize on a null object will throw a NullPointerException)

Difference between synchronized and volatile		
Characteristic	**Synchronized**	**Volatile**
Type of variable	Object	Object or primitive
`Null` allowed?	No	Yes
Can block?	Yes	No
All cached variables synchronized on access?	Yes	From Java 5 onwards
When synchronization happens	When you explicitly enter/exit a `synchronized` block	Whenever a volatile variable is accessed.
Can be used to combined several operations into an atomic operation?	Yes	Pre-Java 5, no. Atomic get-set of volatiles possible in Java 5.

Volatile

- **<u>If a variable is declared as volatile then is guaranteed that any thread which reads the field will see the most recently written value.</u>**

- From Java 5 changes to a volatile variable are always visible to other threads. What's more it also means that when a thread reads a volatile variable in java, it sees not just the latest change to the volatile variable but also the side effects of the code that led up the change.

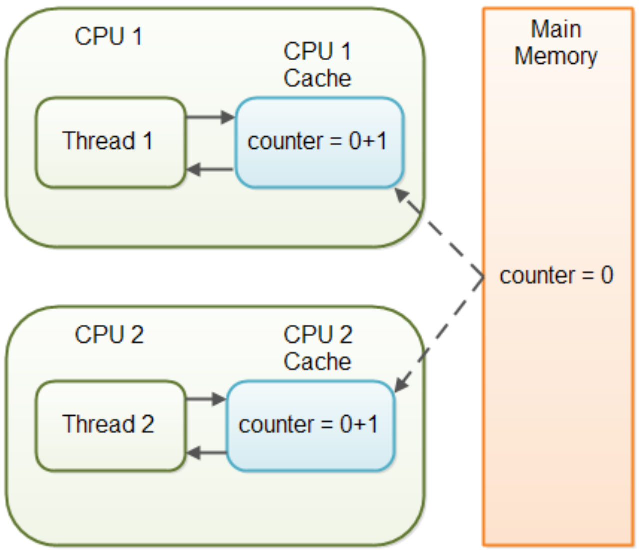
- As of Java 5 write access to a volatile variable will also update non-volatile variables which were modified by the same thread.

```
class VolatileExample {
 int x = 0;
 volatile boolean v = false;
 public void writer() {
  x = 42;
  v = true;
 }

 public void reader() {
  if (v == true) {
   //uses x - guaranteed to see 42.
  }
 }
}
```

Exemplu

```
/**
     * In this example Singleton Instance is declared as volatile variable to ensure
     * every thread see updated value for _instance.
     * * @author Javin Paul
     */



public class Singleton{
     private static volatile Singleton _instance; //volatile variable

     public static Singleton getInstance(){

       if(_instance == null){
            synchronized(Singleton.class){
             if(_instance == null)
             _instance = new Singleton();
            }

       }
       return _instance;

     }
```

volatile nu e suficient – in anumite cazuri….



v

http://tutorials.jenkov.com/images/java-concurrency/java-volatile-3.png

*volatile variable is **not suitable** for cases where we want to **read-update-write** as an atomic operation*

- *Volatile guarantees only Visibility.*
  It does'nt guarantee atomicity.
- *If We have two volatile writes in a method which is being accessed by a thread A and another thread B is accessing those volatile variables , then while thread A is executing the method it might be possible that thread A will be preempted by thread B in the middle of operations(e.g. after first volatile write but before second volatile write by the thread A).*

Read memory-barrier

- The Java memory model actually guarantees that only a read of the same variable that was written to will be guaranteed to be updated.
  - all memory cache lines are flushed upon a write memory barrier being crossed and all memory cache lines are invalidated when a read memory barrier is crossed – regardless of the variable being accessed.

- It is not the write memory-barrier which invalidates the other cache lines. It is the read memory-barrier running in the other processors which invalidates each processor's cache lines.
  - Memory synchronization is cooperative action between the thread writing and the other threads reading from volatile variables.

Exemplu

```
class Sequencer {

   private final AtomicLong sequenceNumber  = new AtomicLong(0);

   public long next() {
     return sequenceNumber.getAndIncrement();
   }
 }
```

java.util.concurrent.atomic

Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile` `int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile` `long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.

A small toolkit of classes that support lock-free thread-safe programming on single variables.

Operatii specifice

- **get** has the memory effects of reading a volatile variable.
- **set** has the memory effects of writing (assigning) a volatile variable.
- **lazySet** has the memory effects of writing (assigning) a volatile variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.
- **weakCompareAndSet** atomically reads and conditionally writes a variable but does *not* create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.
- **compareAndSet** and all other **read-and-update** operations such as **getAndIncrement** have the memory effects of both reading and writing volatile variables.

AtomicInteger

java.lang.Object
java.lang.Number
java.util.concurrent.atomic.AtomicInteger

int	**addAndGet**(int delta)
boolean	**compareAndSet**(int expect, int update)
int	**decrementAndGet**()
double	**doubleValue**()
float	**floatValue**()
int	**get**()
int	**getAndAdd**(int delta)
int	**getAndDecrement**()
int	**getAndIncrement**()
int	**getAndSet**(int newValue)
int	**incrementAndGet**()
int	**intValue**()
void	**lazySet**(int newValue)
long	**longValue**()
void	**set**(int newValue)
String	**toString**()
boolean	**weakCompareAndSet**(int expect, int update)

Intructiuni atomice – C++11

C++ Counter

```
#include <atomic>

struct AtomicCounter {
   std::atomic<int> value;

   void increment(){
      ++value;
   }

   void decrement(){
      --value;
   }

   int get(){
      return value.load();
   }
};
```

```
template <class T> struct atomic;
```

Atomic

Objects of atomic types contain a value of a particular type (`T`).

The main characteristic of atomic objects is that access to this contained value from different threads cannot cause data races.

Additionally

`atomic` objects have the ability to synchronize access to other non-atomic objects in their threads by specifying different *memory orders*.

Template parameters

`T`

Type of the contained value.

This shall be a *trivially copyable type*.

C-style atomic types

contained type	atomic type	description
bool	atomic_bool	
char	atomic_char	atomics for *fundamental integral types*. These are either typedefs of the corresponding full specialization of the `atomic` class template or a base class of such specialization.
signed char	atomic_schar	
unsigned char	atomic_uchar	
short	atomic_short	
unsigned short	atomic_ushort	
int	atomic_int	
unsigned int	atomic_uint	
long	atomic_long	
unsigned long	atomic_ulong	
long long	atomic_llong	
unsigned long long	atomic_ullong	
wchar_t	atomic_wchar_t	
char16_t	atomic_char16_t	
char32_t	atomic_char32_t	
intmax_t	atomic_intmax_t	atomics for *width-based integrals* (those defined in `<cinttypes>`). Each of these is either an alias of one of the above *atomics for fundamental integral types* or of a full specialization of the `atomic` class template with an *extended integral type*. Where *N* is one in 8, 16, 32, 64, or any other type width supported by the library.
uintmax_t	atomic_uintmax_t	
int_least*N*_t	atomic_int_least*N*_t	
uint_least*N*_t	atomic_uint_least*N*_t	
int_fast*N*_t	atomic_int_fast*N*_t	
uint_fast*N*_t	atomic_uint_fast*N*_t	
intptr_t	atomic_intptr_t	
uintptr_t	atomic_uintptr_t	
size_t	atomic_size_t	
ptrdiff_t	atomic_ptrdiff_t	

Functions for atomic objects (C-style)

atomic_is_lock_free

Is lock-free (function)

atomic_init

Initialize atomic object (function)

atomic_store

Modify contained value (function)

atomic_store_explicit

Modify contained value (explicit memory order) (function)

atomic_load

Read contained value (function)

atomic_load_explicit

Read contained value (explicit memory order) (function)

atomic_exchange

Read and modify contained value (function)

atomic_exchange_explicit

Read and modify contained value (explicit memory order) (function)

atomic_compare_exchange_weak

Compare and exchange contained value (weak) (function)

atomic_compare_exchange_weak_explicit

Compare and exchange contained value (weak, explicit) (function)

atomic_compare_exchange_strong

Compare and exchange contained value (strong) (function)

atomic_compare_exchange_strong_explicit

Compare and exchange contained value (strong, explicit) (function)

atomic_fetch_add

Add to contained value (function)

atomic_fetch_add_explicit

Add to contained value (explicit memory order) (function)

atomic_fetch_sub

Subtract from contained value (function)

atomic_fetch_sub_explicit

Subtract from contained value (explicit memory order) (function)

atomic_fetch_and

Apply bitwise AND to contained value (function)

atomic_fetch_and_explicit

Apply bitwise AND to contained value (explicit memory order) (function)

atomic_fetch_or

Apply bitwise OR to contained value (function)

atomic_fetch_or_explicit

Apply bitwise OR to contained value (explicit memory order) (function)

atomic_fetch_xor

Apply bitwise XOR to contained value (function)

atomic_fetch_xor_explicit

Apply bitwise XOR to contained value (explicit memory order) (function)