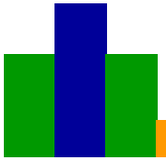


Lecţia 7:

Calcul paralel: Memorie partajată; Thread-uri în Java/PThreads

v1.0

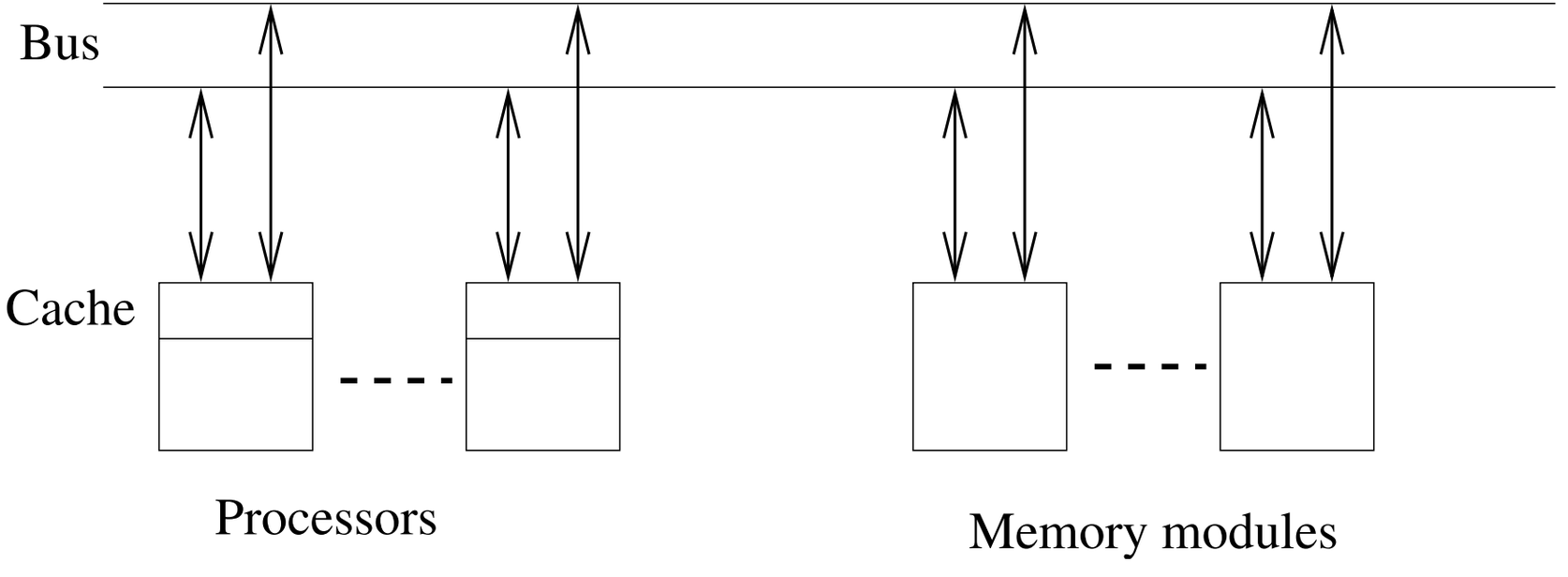
Gheorghe Stefanescu — Universitatea Bucureşti

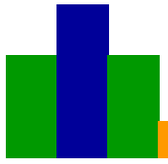
Metode de Dezvoltare Software, Sem.2
Februarie 2007— Iunie 2007

Calcul paralel cu memorie partajată

Sisteme multiprocesor cu memorie partajată:

- *orice* locaţie de memorie este *accesibilă direct* de orice procesor (comunicarea nu este prin trimitere de mesaje)
- există *un singur spaţiu de adrese*
- exemplu: *arhitectura cu o singură magistrală* (utilă pentru un număr mic de procesoare, e.g. ≤ 8)





Programare paralelă

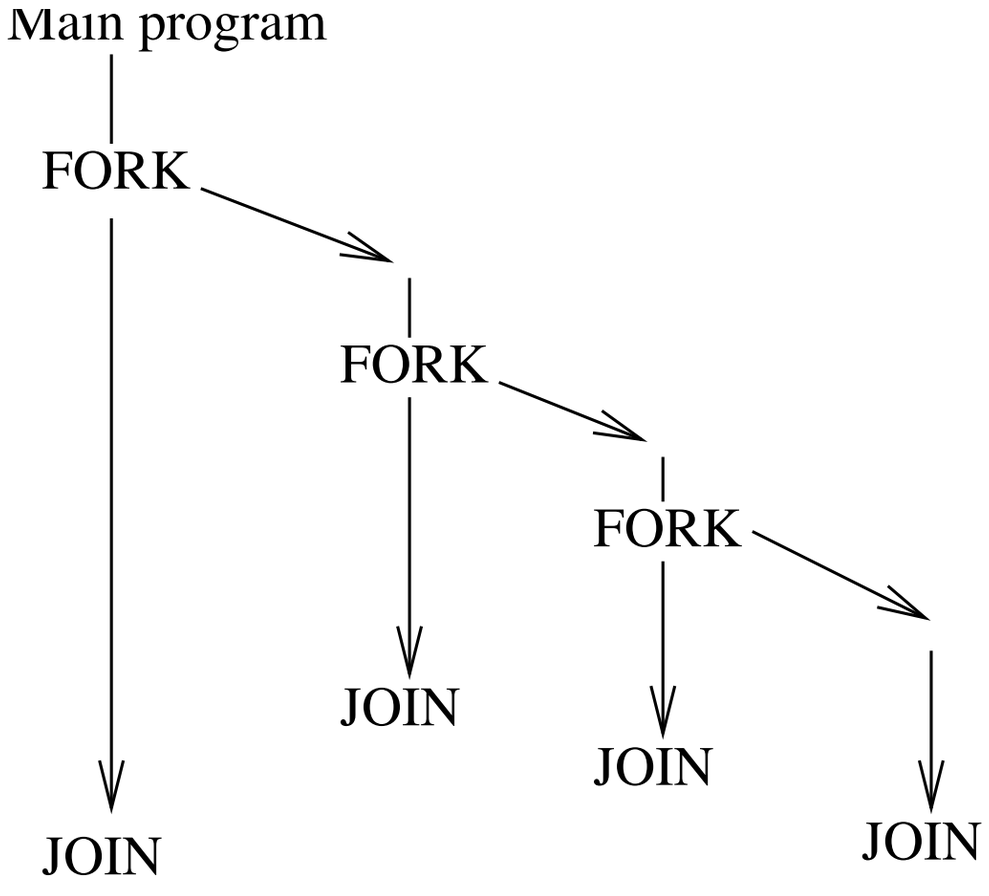
La lista anteriară de alternative de programare paralelă

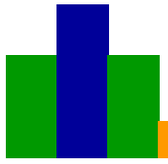
- utilizarea unor *noi* limbaje de programare
- *modificarea* unor limbaje secvenţiale uzuale
- utilizarea de *biblioteci* cu rutine specifice
- utilizarea de programe secvenţiale cu *compilatoare puternice de paralelizare*

se pot adăuga

- *procese Unix* şi
- *Thread-uri* (în: Pthreads, Java, ...)

Operaţia Fork-Join

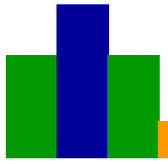




Procese Unix

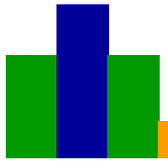
Un *fork* în Unix se obţine folosind instrucţiunea de sistem (system call) `fork()` cu următoarele caracteristici:

- *procesul fiu* nou creat este o *copie exactă* a părintelui, cu singura diferentă că are *propriul său ID*
- procesul fiu are *aceleaşi variabile*, având ca valori iniţiale valorile curente ale procesului părinte
- procesul fiu îşi începe *execuţia* din punctul unde a fost invocată instrucţiunea fork



..Procese Unix

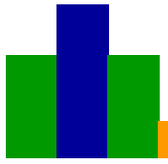
- dacă este executată cu *succes*,
  — `fork()` returnează 0 procesului fiu şi ID-ul fiului procesului părinte;
- dacă *nu are succes*,
  — returnează -1 procesului părinte şi nu se crează nici un proces fiu
- procesele sunt *reunificate* utilizând `wait()` şi `exit()`
  —`wait(statusp)` - întârzie procesul care o invocă
  —`exit(status)` - termină un proces



Fork în Unix: exemplu

Un exemplu simplu de `fork` în Unix:

```
pid = fork();
if (pid == 0) {
  code to be executed by slave
} else {
  code to be executed by parent
}
if (pid == 0) exit (0); else wait(0);
```



Thread-uri

Thread-uri vs. procese (grele) :

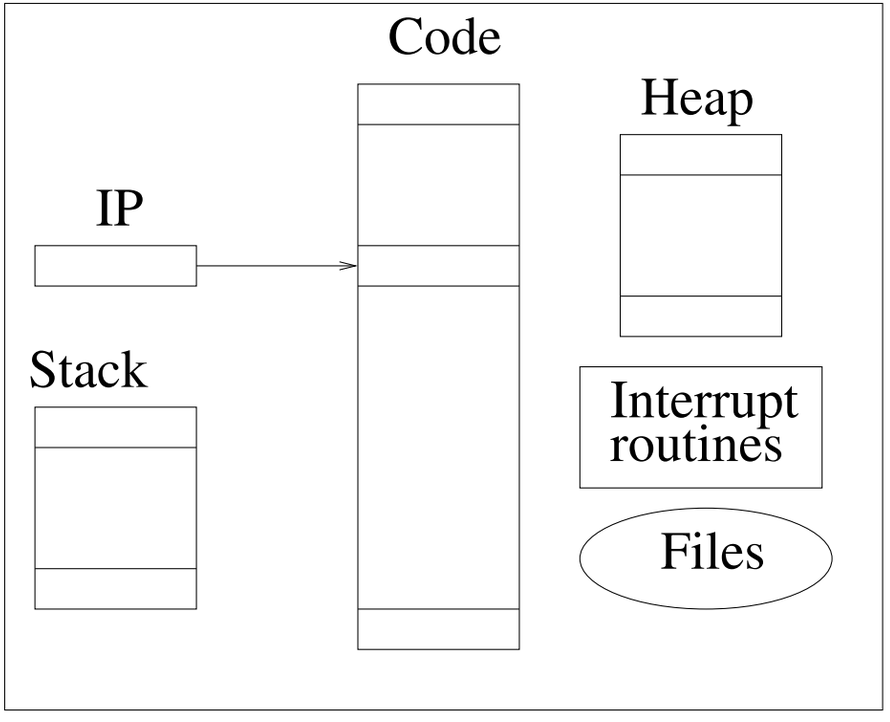
- *procesele* (uzuale, grele) sunt programe complet *separate* (cu variabile, stivă, şi alocare de memorie proprii)
- *thread-urile* folosesc *în comun* acelaşi spaţiu de memorie şi aceleaşi variabile globale (dar au propria lor stiva))

Avantaje ale thread-urilor:

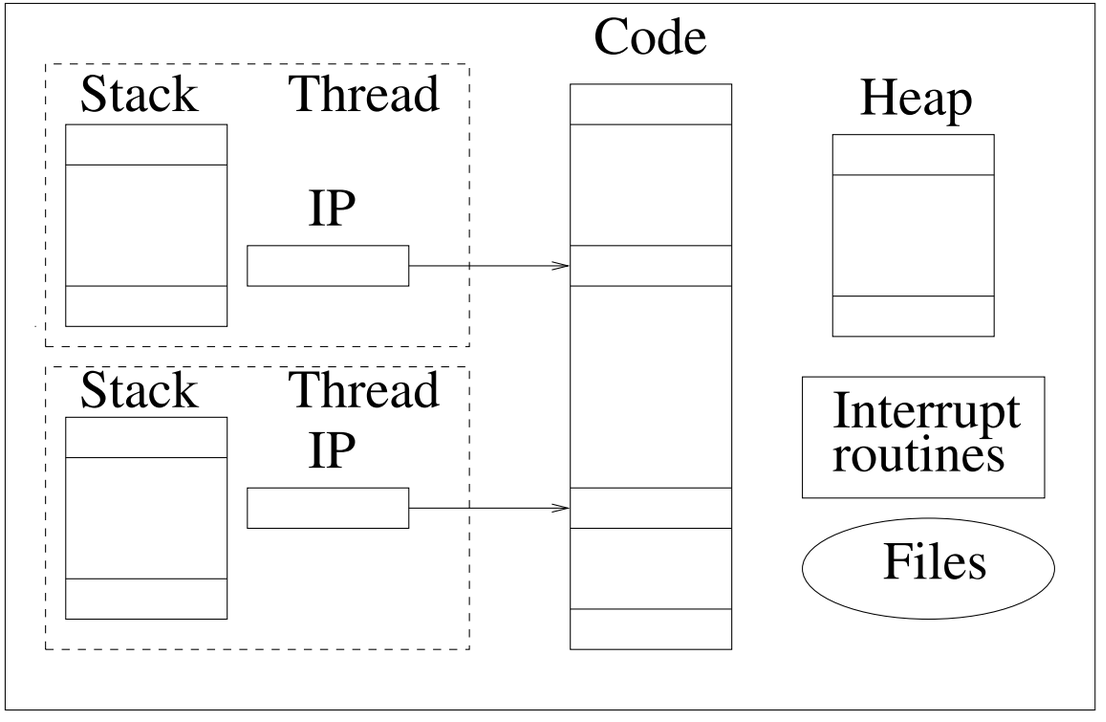
- *crearea* de thread-uri poate fi cu până la trei ordine de magnitudine *mai rapidă* decât a proceselor
- *comunicarea* (prin memoria comună) şi *sincronizarea* sunt *mai rapide* decât în cazul proceselor

Câteodată nu trebuie neapărat de facem “join” pentru un thread fiu cu al părintelui; astfel de thread-uri se numesc *thread-uri detaşate*.

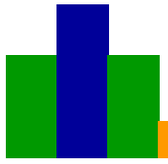
..Thread-uri



(a) Process



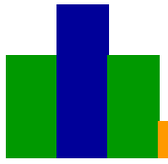
(b) Threads



Pthreads

Pthreads este un standard IEEE

```
Main program                                thread1
   ⋮
pthreads_create(&thread1,                   proc1(&arg){
    NULL, proc1, &args);
   ⋮                                           ⋮
pthreads_join(thread1, *status);            return(*status);
   ⋮                                        }
```

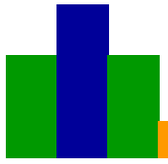


Barieră de thread-uri

O *barieră* se poate defini în Pthreads astfel

```
for (i=0; i<n; i++)
  pthread_create(&thread[i], NULL,
        (void *) slave, (void *) &arg);
for (i=0; i<n; i++)
  pthread_join(thread[i], NULL);
```

Să notăm că `pthreads_join()` *aşteaptă ca un thread specific să se termine. Thread-urile curente dispar după barieră - dacă dorim sa le reutilizăm după barieră, trebuie recreate.*



Ordinea de execuţie a instrucţiunilor

Ordinea de execuţie:

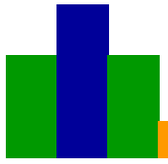
- pe un sistem cu un unic procesor, procesele paralele se *întrepătrund* (engl. “interleaving”)
- pe un sistem cu mai multe procesoare putem aveam procese real paralele, dar, cum numărul de thread-uri este în genere mult mai mare decât al procesoarelor, rămân *unele întrepătrunderi*

Intrepătrundere: Date două secvenţe

$$abc \text{ şi } ABCD$$

prin întrepătrundere orice amestec care păstrează oridinea literelor din fiecare secvenţă este possibil, e.g.,

$$aABbCDc, AabcBCD, aAbBcCD, \ldots$$



..Ordinea de execuţie

Exemple:

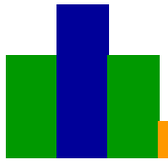
- *Printing*: când mai multe procese printează, caracterele lor se pot amesteca
- *Optimizări la compilare*: uneori ordinea instrucţiunilor este irelevantă; e.g.,

```
a := b*3;
x := sin(y);
```

şi

```
x := sin(y);
a := b*3;
```

sunt echivalente; multe compilatoare/procesoare moderne utilizează astfel de transformări pentru a creşte viteza de calcul



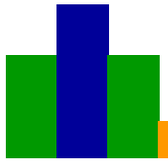
Rutine thread-safe

Rutinele thread-safe sunt instrucţiuni de sistem ori rutine de bibliotecă care pot fi invocate de multiple thread-uri şi produc întotdeauna acelaşi *rezultat corect*

Exemple

- rutina standard I/O de *print* este sigură (safe), adică nu apar amestecuri de caractere din thread-uri diferite
- *accesarea datelor* comune (shared) ori statice este, în genere, thread-safe
- rutinele de sistem care returnează *timpul* pot să nu fie thread-safe

O soluţie generală pentru thread-safeness este de a *include* astfel de rutine în *secţiuni critice* (definite mai jos), dar poate fi ineficient...



Accesarea datelor comune

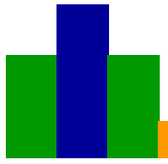
Să zicem că `x` este o variabilă comună iniţial egală cu 0. Care este rezultatul rulării în paralel

```
Process 1:                    Process 2:
x = x+1                       x = x+1
```

?

Răspunsul este: Depinde...

- dacă instrucţiunea de asignare este *atomică*, rezultaul este cel aşteptat, adică 2



..Accesarea datelor comune

- dar dacă instrucţiunea de asignare `x = x+1` este *ne-atomică*, e.g., constă din `read x; compute x+1; write x`

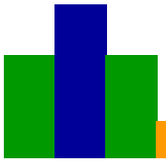
atunci programul devine

```
Process 1:              Process 2:
read x;                 read x;
compute x+1;            compute x+1;
write x;                write x;
```

cu rezultate neaşteptate, i.e.,

$r_1 c_1 w_1 r_2 c_2 w_2 \rightarrow 2$ (ok), dar şi
$r_1 r_2 c_1 c_2 w_1 w_2 \rightarrow 1$!

unde r_1, c_1, w_1 (resp. r_2, c_2, w_2) înseamnă read/compute/write în procesul 1 (resp. 2)

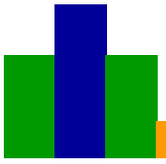


Secţiuni critice

Avem nevoie de mecanisme care să permintă la un timp dat *accesul unui singur proces* la o resursă; secţiuea de cod care gestionează accesul la resursă se numeşte *secţiune critică*. Mecanismul trebuie să îndeplinească următoarele condiţii:

- primul proces care ajunge la secţiune critică pentru o resursă particulară *intră şi execută* secţiunea critică
- în acelaşi timp, *interzice tuturor celorlaltor procese* să intre în secţiunea lor critică pentru a acesa resursa
- când procesul a ieşit din zona critică, un *alt* proces este lăsat să intre în secţiunea lui critică pentru a accesa resursa

Acest mecanism se numeşte mecanism de *excludere mutuală*.

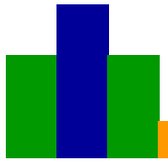


Incuietori (locks)

Incuietorile (locks) reprezintă cel mai simplu mecanism care asigură excluderea mutuală în secţiunile critice şi operează astfel:

- o *încuietore* este o variabilă 0-1 care este
  —1 - spre a indică că *un proces* a *intrat* în secţiunea critică şi
  —0 - spre a indică că *nici un proces* nu este *în* secţiunea critică

- în mare, operează ca o încuietoare de uşă:
  —un proces care *ajunge la uşa* unei resurse critice şi găseşte *uşa deschisă* poate *intra* în secţiunea critică, *încuind* uşa după el pentru a preveni intrarea altor procese;
  —când procesul a *terminat* execuţia din secţiunea critică, *deschide uşa* şi pleacă



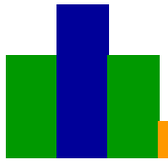
..Incuietori (locks)

"*Spin lock*":

- un proces care încearcă accesul la secţiunea critică poate *verifica continuu* dacă poate intra *stând degeaba*; o astfel de încuietoare se cheamă *spin lock*, iar mecanismul de aşteptare *busy waiting*
- este desigur *ineficient*, fiind preferabil ca procesul să facă ceva

Verificare atomică a încuietorii:

- este important să avem o verificate atomică a încuietorii: e.g.,
  —nu se permite ca două procese să verifice uşa şi să intre *simultan*;
  —dacă un proces găseşte uşa deschisă, nici un alt proces nu este capabil să verifice uşa înainte ca procesul curent să intre şi să o încuie



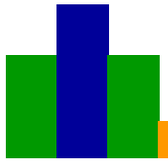
Incuietori în Pthreads

In Pthreads excluderea mutuală se implementează cu variabile *mutex*; ele se *declară* şi iniţializează în programul principal

```
pthread_mutex_t myMutex;
   :
pthread_mutex_intit(&myMutex, NULL);
```

şi se folosesc astfel

```
pthread_mutex_lock(myMutex)
  critical section
pthread_mutex_unlock(myMutex);
```



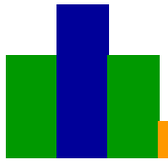
Blocare (deadlock)

Blocarea apare când procesele necesită accesul la resurse deţinute de alte procese şi *nu se mai poate face nici un progres*.

Exemplu (circularitate):

P1	deţine	R1	şi necesită	R2
P2	deţine	R2	şi necesită	R3
⋮				
P(n-1)	deţine	R(n-1)	şi necesită	Rn
Pn	deţine	Rn	şi necesită	R1

O soluţie în Pthread este `pthread_mutex_trylock()` care *testează* încuietoarea *fără a bloca thread-ul*, anume:
— încuie o variabilă mutex neîncuiată şi returnează 0; ori
— ori returnează EBUSY dacă variabila este încuiată.



Semafoare

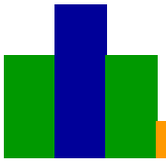
Semafor: introdus de Dijskstra (1968) pentru a controla accesul la secţiunile critice.

Un semafor este un *întreg* pozitiv s (iniţializat cu 1) cu două operaţii

- $P(s)$ - aşteaptă până ce s este mai mare ca 0 şi apoi descreşte s cu 1 ("wait to pass" — consumă o resursă
- $V(s)$ - creşte s cu unu şi eliberează unul din procesele care aşteaptă ("release" — produce o resursă

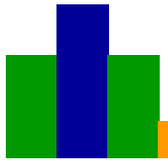
Orice proces are o secvenţă

$$\ldots \text{necritic} \rightarrow P(s) \rightarrow \text{critic} \rightarrow V(s) \rightarrow \text{necritic} \ldots$$



..Semafoare

- *semafoarele binare* se comportă ca încuietorile, dar operaţiile P/V dau şi un proces de *planificare (scheduling)* pentru selecţia proceselor
- *semafoarele generale* (cu numere pozitive arbitrare) pot fi folosite pentru a rezolva probleme de tip producător/consumator, contorizând numărul de resurse libere
- *nu* există semafoare în *Pthreads*, dar *există* în *Unix*
- semafoarele sunt suficient de *puternice* spre a rezolva multe din problemele de acces la secţiunile critice, dar reprezintă un *mecanism de nivel-scăzut* care poate fi dificil de gestionat



Monitor

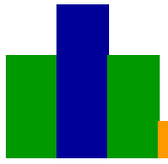
Hoare (1974) a introdus noţiunea de *monitor* care este

- o tehnică la *nivel mai-înalt* pentru a gestiona accesul la secţiunile critice, şi
- încapsulează *date* şi *operaţii de acces* într-o singură structură

Un monitor se poate implementa cu ajutorul semafoarelor astfel

```
monitor_proc1{
  P(monitor_semaphore);
    monitor body
  V(monitor_semaphore);
  return;
}
```

Java are acest tip de "monitoroare".



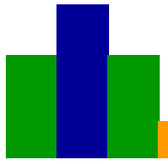
Variabile condiţionale

Problemă:

- *intrarea* într-o secţiune *critică* poate necesita *verificarea frecventă* a unor condiţii de acces
- *verificarea* lor *continuă* poate fi foarte *ineficientă*

O soluţie care a fost propusă este de a folosi *variabile condiţionale* (utilizate, e.g., în cazul monitoarelor):

- o astfel de variabilă condiţională este întovarăşită de trei operaţii
  —`Wait(condVar)` - *aşteaptă* ca respectiva condiţie să apară
  —`Signal(condVar)` - trimite un *semnal* că respectiva condiţie a apărut
  —`Status(condVar)` - returnează *numărul* de procese care *aşteaptă* ca respectiva condiţie să apară

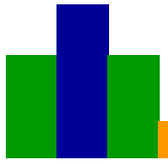


..Variabile condiţionale

Exemplu:

```
action(){                    counter(){
    ⋮                            ⋮
  lock();                      lock();
  while(x != 0)                x--;
    wait(s);                     if (x == 0) signal(s);
  unlock(s);                   unlock();
  doAction;                      ⋮
    ⋮                        }
}
```

`wait(s)` deschide încuietoarea şi aşteaptă semnalul `s`; bucla `while...` se foloseşte pentru a reverifica condiţia; de notat că semnalele nu au memorie [dacă `counter` ajunge primul în secţiunea critică, semnalul `s` se pierde]



Variabile condiţionale în Pthread

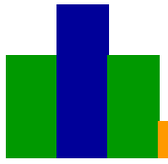
Pthread are *variabile condiţionale* asociate variabilelor *mutex*;
Mecanismele de semnalizare şi aşteptare au următorul format:

—în partea `action`

```
   ⋮
pthread_mutex_lock(&mutex1);
while (c <> 0)
  pthread_mutex_wait(cond1,mutex1);
pthread_mutex_unlock(&mutex1);
```

—în partea `counter`

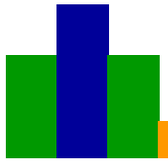
```
   ⋮
pthread_mutex_lock(&mutex1);
c--;
  if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
```



Instrucţiuni specifice pentru paralelism

Dăm mai jos câteva instrucţiuni de programare *specifice* pentru modelul de paralelism cu memorie partajată:

- *Shared data* - se pot specifica prin
  ```
  shared int;
  ```
- *Concurrent statements (diferite)* - se pot introduce cu instrucţiunea `par`
  ```
  par{S1;S2; ...,Sn;}
  ```
- *Concurrent statements (similare)* - se pot introduce cu instrucţiunea `forall`
  ```
  forall (i=0; i<n; i++){body}
  ```
  care generează *n* blocuri concurente {`body1; body2; ...; bodyn;`}, câte o instanţă a lui `body` pentru fiecare `i`



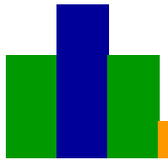
Analiza dependenţelor

Analiza dependenţelor este o tehnică *automată* utilizată intensiv de *compilatoarele de paralelizare* spre a detecta procese/instrucţiuni care se pot executa în *paralel*

Exemple (primul - evident; al doilea - uşor, de nu aşa de evident)

```
forall(i=0; i<5; i++)
  a[i] = 0
```

```
forall(i=2; i<6; i++){
  x = i - 2*i + i*i;
  a[i] = a[x];
}
```



Condiţiile lui Bernstein

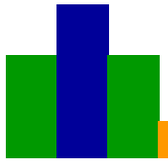
Condiţiile lui Bernstein reprezintă un set de condiţii *suficiente* ca două procese să se poată executa simultan în situaţia de *"read concurent"/"write exclusiv"*:

- se definesc două mulţimi de locaţii de memorie
  I_i - mulţimea locaţiilor de memorie citite de procesul P_i
  O_i - mulţimea locaţiilor de memorie în care procesul P_i scrie

- două procese P_1, P_2 se pot *executa simultan* dacă

$$O_1 \cap O_2 = \emptyset \quad \& \quad I_1 \cap O_2 = \emptyset \quad \& \quad I_2 \cap O_1 = \emptyset$$

(In vorbe: locaţiile de *scriere* sunt *disjuncte* şi nici un proces *nu citeşte* dintr-o locaţie folosită pentru *scriere* de *celălalt* proces.)

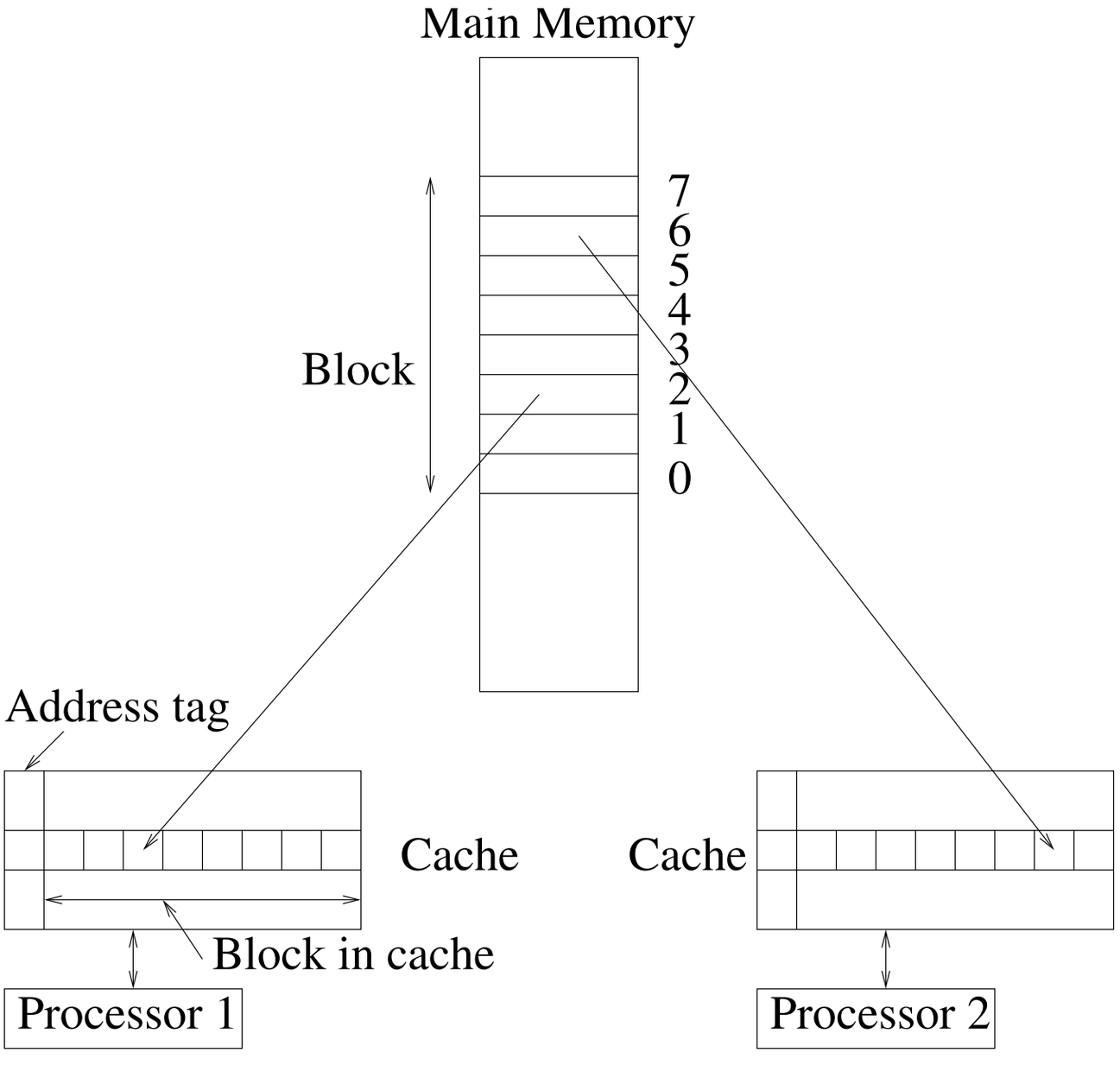


Sisteme cu cache-uri

- *memoria cache* este o memorie performantă, conectată la procesor, în care se reţin datele şi codul recent folosite
- *protocoale de coerenţă a cache-urilor*: trebuie să asigure coerenţa memoriilor cache ale procesoarelor; două soluţii sunt:
  —*update policy* - copiile din toate cache-urile sunt actualizate de fiecare dată când una se modifică
  —*invalidate policy* - când o dată dintr-o copie se modifică într-un cache, dată respectivă se invalidează în celelalte cache-uri (resetând un bit); noua dată de actualizează numai dacă procesorul are nevoie de ea
- un *sharing fals* poate apare când diverse procesoare alterează zone diferite dintr-un bloc (soluţie - rearanjarea memoriei)

Thread-uri

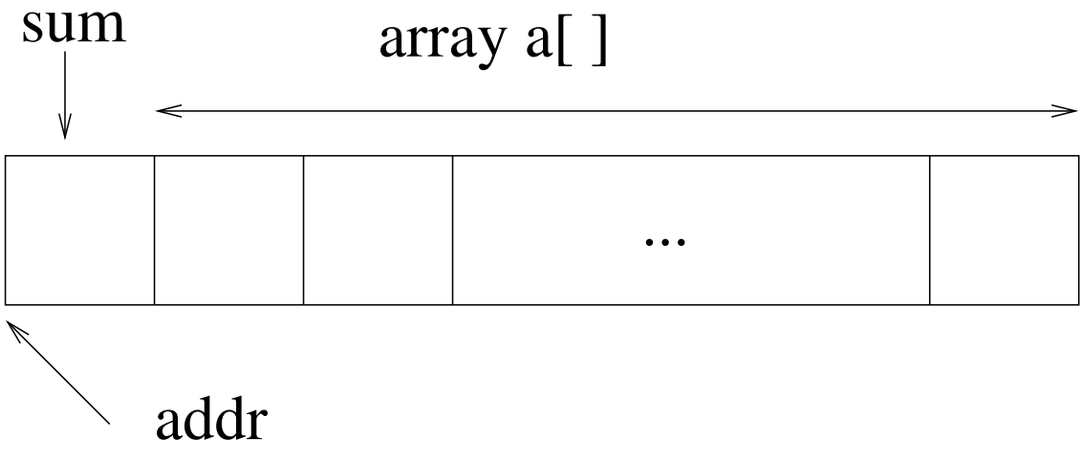
Sharing fals:

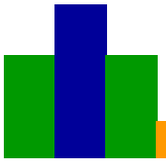


Exemple (adună 1000 numere): 1 Unix

Adunare de numere: Scopul este de a aduna 1000 numere cu două procese, anume `P1` adună numerele cu index par din vectorul `a[ ]`, iar `P2` pe cele cu index impar din `a[ ]`.

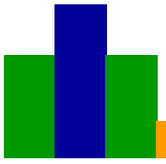
Folosim structura de memorie





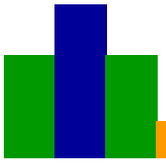
..Exemple / Unix

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
   int shmid, s, pid;
   char *shm;
   int *a, *addr, *sum;
   int partial_sum;
   int i;
```



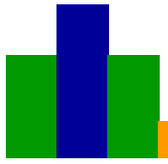
..Exemple / Unix

```
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT));
if (s == -1){
   perror("semget");
   exit(1);
}
if (semctl(s,0,SETVAL, init_sem_value) < 0){
   perror("semctl");
   exit(1);
}

shmid = shmget(IPC_PRIVATE, (array_size*sizeof(int)+1),
      (IPC_CREAT|0600));
if (shmid == -1){
   perror("semget");
   exit(1);
}
```



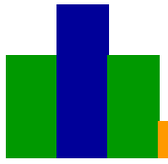
..Exemple / Unix

```
shm = shmat(shmid, NULL, 0);

if (shm == (char*)-1){
   perror("shmat");
   exit(1);
}

addr = (int*)shm;
sum = addr;
addr++;
a = addr;

*sum = 0;
for (i = 0; i < array_size; i++)
   *(a+i) = i+1;
```



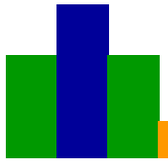
..Exemple / Unix

```
pid = fork();
if (pid == 0){
   partial_sum = 0;
   for (i = 0; i < array_size; i = i + 2)
      partial_sum += *(a + i);
} else {
   partial_sum = 0;
   for (i = 1; i < array_size; i = i + 2)
      partial_sum += *(a + i);
}
P(&s);
*sum += partial_sum;
V(&s);

printf("\nprocess pid = %d, partial sum = %d\n",
      pid, partial_sum);
if (pid == 0) exit(0); else wait (0);
printf("\nThe sum of 1 to %i is %d\n",
      array_size, *sum);
```



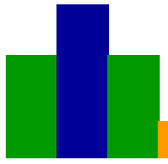
..Exemple / Unix

```
    if (semctl(s, 0, IPC_RMID, 1) == -1) {
       perror("semctl");
       exit(1);
    }

    if (shmctl(shmid, IPC_RMID, NULL) == -1) {
       perror("shmctl");
       exit(1);
    }

    exit(0);
}
```



..Exemple / Unix

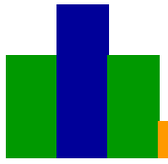
```
void P(int *s)
   {
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = -1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
   return;
}
```



..Exemple / Unix

```
void V(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = 1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
   return;
}
```

```
process pid = 0, partial sum = 250000
process pid = 2073, partial sum = 250500

The sum of 1 to 1000 is 500500
```



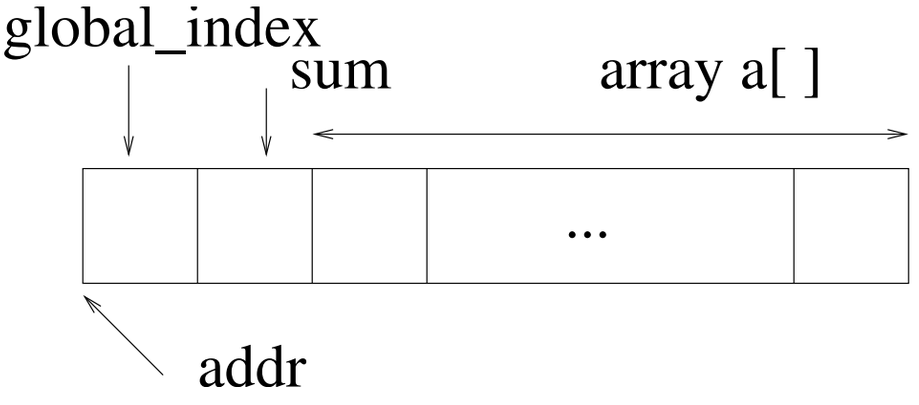
Exemple (adună 1000 numere): 2 Pthreads

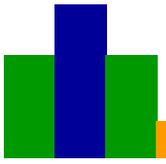
Adunare de numere: Scopul este acelaşi (de a aduna 1000 de numere), dar acum folosind 10 thread-uri:

- fiecare thread citeşte un index global `global_index` pentru a accesa vectorul `a[]` şi adună elementul corespunzător la suma sa locală `partial_sum`
- apoi, fiecare thread va aduna valoarea sa `partial_sum` la o variabilă globală `sum`
- ambele, accesul şi actualizarea pentru variabilele `global_index` şi `sum` sunt secţiuni critice implementate cu vabiabile mutex

..Exemple / Pthreads

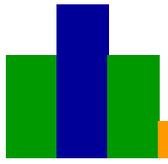
Folosim următoare structură de memorie





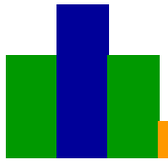
..Exemple / Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;
```



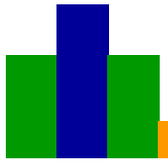
..Exemple / Pthreads

```
void *slave(void *ignored)
{
   int local_index, partial_sum = 0;
   do {
      pthread_mutex_lock(&mutex1);
         local_index = global_index;
         global_index++;
      pthread_mutex_unlock(&mutex1);

      if (local_index < array_size)
         partial_sum += *(a + local_index);
   } while (local_index < array_size);
   pthread_mutex_lock(&mutex1);
      sum += partial_sum;
   pthread_mutex_unlock(&mutex1);
   return();
}
```



..Exemple / Pthreads

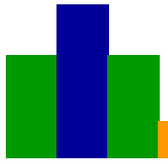
```
main()
{
   int i;
   pthread_t thread[10];
   pthread_mutex_init(&mutex1, NULL);

   for (i = 0; i < array_size; i++)
      a[i] = i+1;


   for (i = 0; i < no_threads; i++)
      if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
         perror("Pthread_create fails");

   for (i = 0; i < no_threads; i++)
      if (pthread_join(thread[i], NULL) != 0)
         perror("Pthread_join fails");

   printf("The sum of 1 to %i is %d\n", array_size, sum);
}
```



Exemple (adună 1000 numere): 3 Java

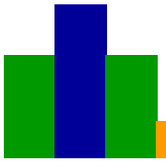
Adunare de numere: Ca mai sus, adunăm 1000 de numere cu 10 thread-uri, dar acum folosind Java.

Gestionarea accesul la secţiunile critice în Java se face folosind metode sincronizate, anume *synchronized methods*.



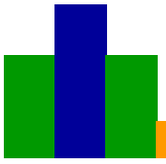
..Exemple / Java

```
public class Adder
{
   public int[] array;
   private int sum = 0;
   private int index = 0;
   private int number_of_threads = 10;
   private int threads_quit;

   public Adder()
   {
      threads_quit = 0;
      array = new int[1000];
      initializeArray();
      startThreads();
   }

   public synchronized int getNextIndex()
   {
      if (index < 1000) return (index++); else return (-1);
   }
```



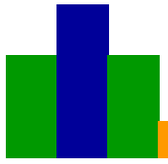
..Exemple / Java

```
public synchronized void addPartialSum(int partial_sum)
{
   sum = sum + partial_sum;
   if (++threads_quit == number_of_threads)
      System.out.println("The sum of the numbers is " + sum);
}

private void initializeArray()
{
   int i;
   for (i = 0; i < 1000; i++)
      array[i] = i;
}
```



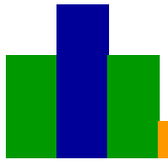
..Exemple / Java

```
    public void startThreads()
    {
       int i = 0;
       for (i = 0; i < 10; i++)
          {
             AdderThread at = new AdderThread(this,i);
          at.start();
          }
    }

    public static void main(String args[])
    {
       Adder a = new Adder();
    }
}
```



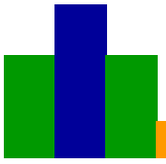
..Exemple / Java

```
class AdderThread extends Thread
{
   int partial_sum = 0;
   Adder parent;
   int number;

   public AdderThread(Adder parent, int number)
   {
      this.parent = parent;
      this.number = number;
   }
```



..Exemple / Java

```
    public void run()
    {
       int index = 0;

       while(index != -1) {
          partial_sum = partial_sum + parent.array[index];
          index = parent.getNextIndex();
       }

       System.out.println("Partial sum from thread "
            + number + " is " + partial_sum);
       parent.addPartialSum(partial_sum);
    }
}
```



..Exemple / Java

```
Partial sum from thread 8 is 0
Partial sum from thread 7 is 910
Partial sum from thread 1 is 165931
Partial sum from thread 2 is 51696
Partial sum from thread 9 is 10670
Partial sum from thread 6 is 54351
Partial sum from thread 0 is 98749
Partial sum from thread 3 is 62057
Partial sum from thread 5 is 45450
Partial sum from thread 4 is 9686
The sum of the numbers is 499500
```