

Curs 4

Programare Paralela si Distribuita

Concurenta

Deadlock, Starvation, Livelock

Semafoare, Mutex, Monitoare, Variabile Conditionale

Forme de interactiune intre procese/threaduri

1. **comunicarea** între procese distincte
-transmiterea de informații între procese
2. **sincronizarea** astfel încât procesele să aștepte informațiile de care au nevoie și nu sunt produse încă de alte procese/thread-uri
- restricții asupra evoluției în timp a unui proces/thread

Deadlock – Starvation - Livelock

- ***Deadlock***
 - situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecare proces asteapta dupa o resursa care este retinuta de alt proces care la randul lui asteapta dupa alta resursa.
- ***Starvation***
 - Daca unui thread nu i se aloca timp de executie *CPU time* pentru ca alte threaduri folosesc CPU
 - Thread este "*starved to death*" pentru ca alte threaduri au acces la CPU in locul lui.
 - Situatia corecta "*fairness*" toate threadurile au sanse egale la folosire CPU.
- ***Livelock***
 - Situatia in care un grup de procese/threaduri nu progreseaza datorita faptului ca isi cedeaza reciproc executia

Exemplu - Deadlock

```
public class TreeNode {  
  
    TreeNode parent = null;  
    List<TreeNode> children = new ArrayList();  
  
    public synchronized void addChild(TreeNode child){  
        if(! this.children.contains(child)) {  
            this.children.add(child);  
            child.setParentOnly(this);  
        }  
    }  
  
    public synchronized void addChildOnly(TreeNode child){  
        if(!this.children.contains(child)){  
            this.children.add(child);  
        }  
    }  
  
    public synchronized void setParent(TreeNode parent){  
        this.parent = parent;  
        parent.addChildOnly(this);  
    }  
  
    public synchronized void setParentOnly(TreeNode parent){  
        this.parent = parent;  
    }  
}
```

Cum apare deadlock?

In ce situatii?

Safety - Liveness

- *Safety*
 - "nothing bad ever happens"
 - *a program never terminates with a wrong answer*
- **Fairness**
 - presupune o rezolvare corecta a nedeterminismului in executie
 - Weak fairness
 - daca o actiune este in mod continuu accesibila (*continuously enabled*) (stare-ready) atunci trebuie sa fie executata infinit de des (*infinitely often*).
 - Strong fairness
 - daca o actiune este infinit de des accesibila (*unfinetely often enabled*) dar nu obligatoriu in mod continuu atunci trebuie sa fie executata infinit de des (*infinetely often*).
- Liveness
 - "something good eventually happens"
 - *a program eventually terminates*

Forme de sincronizare

- excluderea mutuală: se evită utilizarea simultană de către mai multe procese a unei resurse critice.
 - O resursă este critică dacă poate fi utilizată doar de către singur proces la un moment dat
 - arbitrarea: se evită accesul simultan din partea mai multor procesoare la aceeași locație de memorie.
 - se realizează o secvențializare a accesului, impunând așteptarea până când procesul care a obținut acces și-a încheiat activitatea asupra locației de memorie.
- sincronizarea pe condiție: se amână execuția unui proces până când o anumită condiție devine adevărată;

Mecanisme de sincronizare

- Semafoare
- Variabile conditionale
- Monitoare

Ref.: **Bertrand Meyer. Sebastian Nanz. *Concepts of Concurrent Computation***

Semafoare

- Primitiva de sincronizare de nivel inalt (nu cel mai inalt)
- Foarte mult folosita
- Implementarea necesita operatii atomice
- Inventata de E.W. Dijkstra in 1965

Definitie

Semafor (general) \Rightarrow s este caracterizat de

- O variabila $\rightarrow count = v(s)$ (valoarea semaforului)
- 2 operatii $P(s)/down$ si $V(s)/up$:

Operatiile semafoarelor

- Gestiunea semafoarelor: prin 2 operații indivizibile
 - $P(s)$ –este apelată de către procese care doresc să acceseze o regiune critică pt a obține acces.
 - Efect: - incercarea obtinerii accesului procesului apelant la secțiunea critică si decrementarea valorii.
 - dacă $v(s) \leq 0$, procesul ce dorește execuția secțiunii critice așteaptă
 - $V(s)$
 - Efect : incrementarea valorii semaforului.
 - se apelează la sfârșitul secțiunii critice și semnifică eliberarea acesteia pt. alte procese.
- Succesiune instrucț.:
 - $P(s)$
 - regiune critică
 - $V(s)$
 - Rest. procesului

- Cerinte de atomicitate:
 - Testarea
 - Incrementare/decrementarea valorii
- Un semafor general se numeste si semafor de numarare (*Counting semaphore*)
- Valoarea unui semafor = valoarea *count*

```
class SEMAPHORE feature
  count : INTEGER
  down
  do
    await count > 0
    count := count - 1
  end
  up
  do
    count := count + 1
  end
end
```

Semafor Binar

- Valoarea semaforului poate lua doar valorile 0 si 1

Valoarea => poate fi de tip boolean

```
b : BOOLEAN
down
  do
    await b
    b := false
  end
up
  do
    b := true
  end
```

Starvation-free

- Daca semaforul se foloseste fara a se mentine o evidenta a proceselor care asteapta intrarea in sectiunea critica nu se poate asigura *starvation-free*
- Pentru a se evita aceasta problema, procesele (referinte catre ele) blocate sunt tinute intr-o colectie care are urmatoarele operatii:
 - add(P)
 - Remove (P)
 - is_empty

Weak Semaphore

- Un semafor 'slab' se poate defini ca o pereche $\{v(s), c(s)\}$ unde:
 - $v(s)$ este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese.
 - $c(s)$ o **multime de așteptare** la semafor - conține referințe la procesele care așteaptă la semaforul s .

+

Operatiile $P(s)/\text{down}$ si $V(s)/\text{up}$

Strong Semaphore

- Un semafor 'puternic' se poate defini ca o pereche $\{v(s), c(s)\}$ unde:
 - $v(s)$ este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese.
 - $c(s)$ o **coadă de așteptare** la semafor - conține referințe la procesele care așteaptă la semaforul s (FIFO).

+

Operatiile P/down si V/up

Schita de implementare

```
count : INTEGER
blocked: CONTAINER
down
  do
    if count > 0 then
      count := count - 1
    else
      blocked.add(P)      -- P is the current process
      P.state := blocked -- block process P
    end
  end
up
  do
    if blocked.is_empty then
      count := count + 1
    else
      Q := blocked.remove -- select some process Q
      Q.state := ready    -- unblock process Q
    end
  end
end
```

Analiza

- Invariant:

$$count \geq 0$$

$$count = k + \#up - \#down$$

- Demonstratie

Apel down:

- if $count > 0 \Rightarrow \#down$ este incrementat si $count$ decrementat
- if $count \leq 0 \Rightarrow down$ nu se termina si $count$ nu se modifica.

Apel up:

- if $blocked(is_empty) \Rightarrow \#up$ si $count$ sunt incrementate;
- if $blocked(not\ is_empty) \Rightarrow \#up$ and $\#down$ sunt incrementate si $count$ nu se modifica.

- $k \geq 0$: valoarea initiala a semaforului
- $count$: valoarea curenta a semaforului
- $\#down$: nr. de op. down terminate
- $\#up$: nr. de op. up terminate

- *Starvation*
 - este posibila pt semafoarele de tip *weak semaphores*:
Pentru ca procesul de selectie este de tip random

Semafoare Binare

- Count ia doar 2 valori
 - 0->false
 - 1->true

=> excludere mutuala

- Mutex - Un semafor binar

Simulare semafor general prin semafoare binare

- mutex protejeaza
modificarile var count

```
mutex.count := 1 -- binary semaphore  
delay.count := 1 -- binary semaphore  
count := k
```

```
general_down  
do  
  delay.down  
  mutex.down  
  count := count - 1  
  if count > 0 then  
    delay.up  
  end  
  mutex.up  
end
```

Primele k-1 procese
nu asteapta;
Urmatoarele DA.

```
general_up  
do  
  mutex.down  
  count := count + 1  
  if count = 1 then  
    delay.up  
  end  
  mutex.up  
end
```

Varianta de bariera de sincronizare folosind semafoare

2 procese

2 semafoare

<code>s1.count := 0</code> <code>s2.count := 0</code>			
P1		P2	
1	code before the barrier	1	code before the barrier
2	s1.up	2	s2.up
3	s2.down	3	s1.down
4	code after the barrier	4	code after the barrier

s1 furnizeaza bariera pentru P2,
s2 furnizeaza bariera pentru P1

Java

`java.util.concurrent.Semaphore` package

- Constructors:
 - `Semaphore(int k)`, weak semaphore
 - `Semaphore(int k, boolean b)`, strong semaphore if `b=true`
- Operations:
 - `acquire()`, (down)→ throws `InterruptedException`
 - `release()`, (up)

Dezavantaje - semafoare

- Nu se poate determina utilizarea corecta a unui semafor doar din bucata de cod in care apare; intreg programul trebuie analizat.
- Daca se pozitioneaza incorect o operatie P sau V atunci se compromite corectitudinea.
- Este usor sa se introduca *deadlocks* in program.
- => o varianta mai structurata de nivel mai inalt => Monitor

Monitor

- Un monitor poate fi considerat un tip abstract de dată (poate fi implementat ca si o clasa) care constă din:
 - un set permanent de variabile ce reprezintă resursa critică,
 - un set de proceduri ce reprezintă operații asupra variabilelor și
 - un corp (secvență de instrucțiuni).
 - Corpul este apelat la lansarea ‘programului’ și produce valori inițiale pentru variabilele-monitor (cod de initializare).
 - Apoi monitorul este accesat numai prin procedurile sale.
- codul de inițializare este executat înaintea oricărui conflict asupra datelor ;
- numai una dintre procedurile monitorului poate fi executată la un moment dat;
- Monitorul creează o coadă de așteptare a proceselor care fac referire la anumite variabile comune.

Monitor

- Excluderea mutuală este realizată prin faptul că la **un moment dat poate fi executată doar o singură procedură a monitorului!**
- **Sincronizarea pe condiție** se poate realiza prin mijloace definite explicit de către programator prin variabile de tip condiție și două operații:
 - **signal** (notify)
 - **wait**.
- Dacă un proces care a apelat o procedură de monitor găsește **condiția falsă**, execută operația **wait** (punere în așteptare a procesului într-un șir asociat condiției și eliberează monitorul).
- în cazul în care alt proces care execută o procedură a aceluiași monitor găsește/setează **condiția adevărată**, execută o operație **signal**
 - procesul continuă dacă șirul de așteptare este vid, altfel este pus în așteptare și se va executa un alt proces extras din șirul de așteptare al condiției.

Monitor – object oriented view

Monitor class :

- toate attributele sunt private
- rutinele sale se executa prin excludere mutuala;

Instantiere clasa Monitor = monitor

- Attribute \leftrightarrow *shared variables*, (thread-urile le acceseaza doar via monitor)
- Corpurile rutinelor corespund sectiunilor critice – doar o rutina este activa in interiorul monitorului la orice moment).

Schita Implementare

```
monitor class MONITOR_NAME
  feature
    -- attribute declarations
    a1: TYPE1
    ...

    -- routine declarations
    r1 (arg1, ..., argk) do ... end
    ...

  invariant
    -- monitor invariant
end
```

Implementare folosind un semafor: strong semaphore

`entry` : SEMAPHORE

Initializare $v(\text{entry}) = 1$

```
r (arg1, ..., argk)  
do  
    entry.down  
    bodyr  
    entry.up  
end
```

Variabile conditionale in monitoare

- O abstractizare care permite sincronizarea conditionala;
- Variabile conditionale sunt asociate cu lacatul unui monitor (monitor lock);
- Permit threadurilor sa astepte in interiorul unei sectiuni critice eliberand lacatul monitorului.

Variabile conditionale -> sincronizare conditionala

Monitoarele ofera variabile conditionale.

O variabila conditionala consta dintr-o coada de blocare si 3 operatii atomice:

- **wait** elibereaza lacatul monitorului, blocheaza threadul care se executa si il adauga in coada
- **signal** – daca coada este empty nu are efect;
- altfel deblocheaza un thread
- **is_empty** returneaza ->true, daca coada este empty,
-> false, altfel.
- Operatiile **wait** si **signal** pot fi apelate doar din corpul unei rutine a monitorului (=> acces sincronizat).

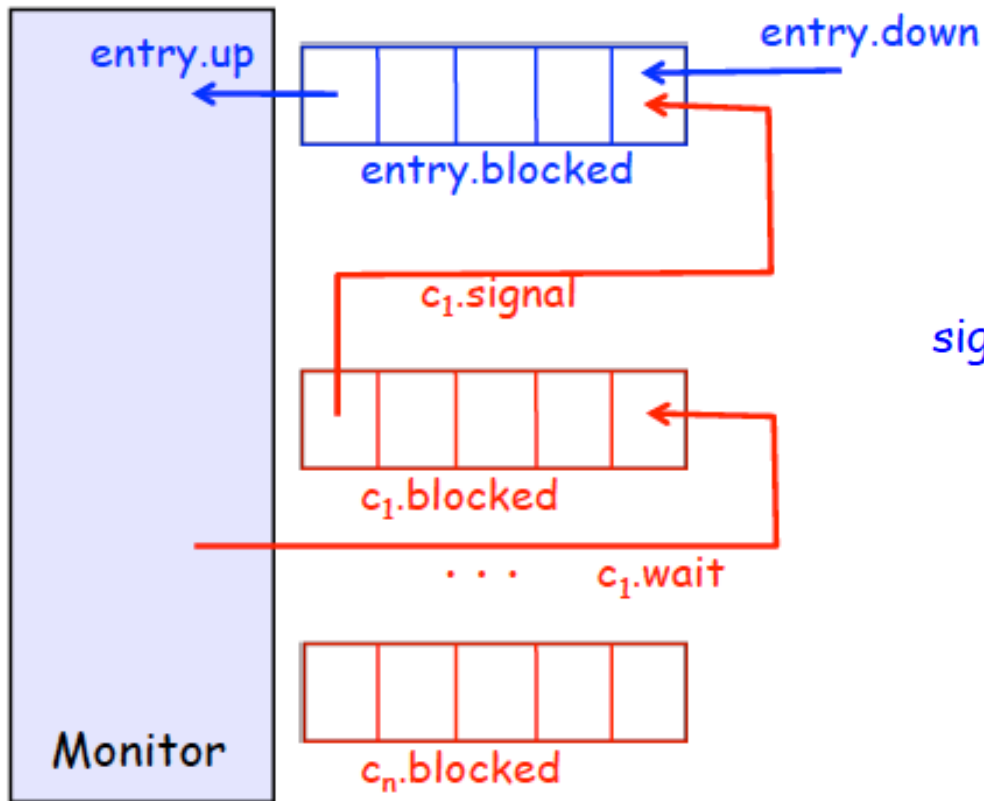
Schita Implementare

```
class CONDITION_VARIABLE
feature
  blocked: QUEUE
  wait
  do
    entry.up          -- release the lock on the monitor
    blocked.add(P)    -- P is the current process
    P.state := blocked -- block process P
  end
  signal deferred end -- behavior depends on signaling discipline
  is_empty: BOOLEAN
  do
    result := blocked.is_empty
  end
end
```

Disciplina de semnalizare (*signal*)

- Atunci cand un proces executa un semnal/*signal* pe o conditie el se executa inca in interiorul monitorului;
- Doar un proces se poate executa in interiorul monitorului => un proces neblocat nu poate intra in monitor imediat
- Doua solutii:
 - Procesul de semnalizare continua si procesul notificat este mutat la intrarea monitorului;
 - Procesul care semnalizeaza lasa monitorul si procesul semnalizat continua.

Signal & Continue



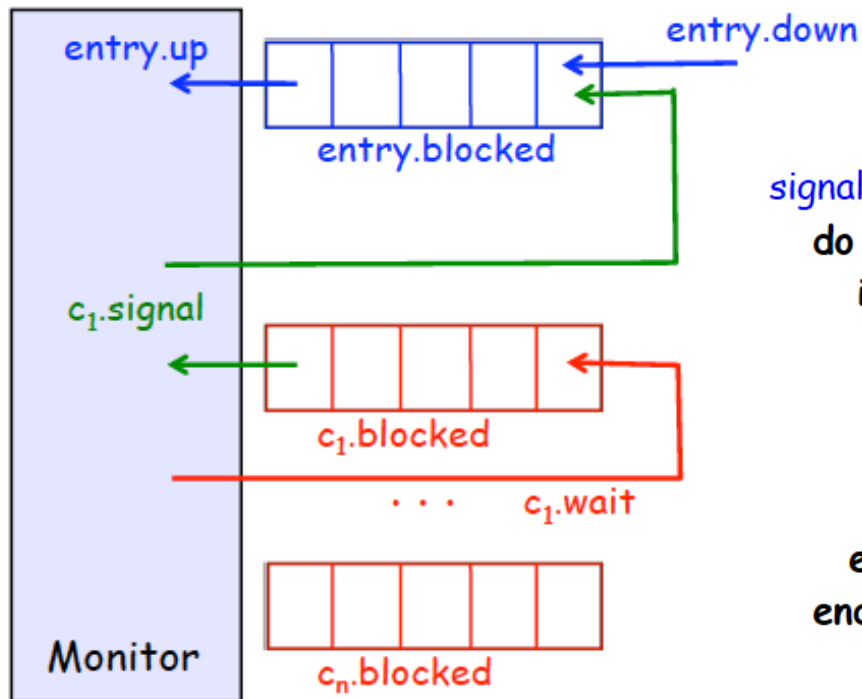
Pentru fiecare
conditie => o
coada

```

signal
do
  if not blocked.is_empty then
    Q := blocked.remove
    entry.blocked.add(Q)
  end
end
    
```

Q se introduce in
coada semaforului

Signal & wait



signal

do

if not blocked.is_empty then

entry.blocked.add(P) -- P is the current process

Q := blocked.remove

Q.state := ready -- unblock process Q

P.state := blocked -- block process P

end

end

- 'Signal and Continue', -> `signal` este doar un "hint" ca o conditie ar putea fi adevarata – dar alte threaduri ar putea intra si seta conditia la false
- Pt. 'Signal and Continue' este si operatia `signal_all`

`while not blocked.is_empty do signal end`

Alte discipline

- Urgent Signal and Continue: caz special pt 'Signal and Continue' prin care thread-ului deblocaat prin [signal](#) i se da o prioritate mai mare in [entry.blocked](#) (trece in fata)
- Signal and Urgent Wait: caz special pt 'Signal and Wait', prin care thread-ului care a semnalizat i se da o prioritate mai mare in [entry.blocked](#) (trece in fata)

Monitor in Java

- Fiecare obiect din Java are un monitor care poate fi blocat sau deblocat in blocurile sincronizate:

```
Object lock = new Object();  
synchronized (lock) {  
    // critical section  
}
```

:

```
synchronized type m(args) {  
    // body  
}
```

- echivalent

```
type m(args) {  
    synchronized (this) {  
        // body  
    }  
}
```

Monitor in Java

Prin metodele `synchronized` monitoarele pot fi emulate

- nu e monitor original
- variabilele conditionale nu sunt explicit disponibile , dar metodele
 - `wait()`
 - `notify()` // signal
 - `notifyAll()` // signal_all

pot fi apelate din orice cod `synchronized`

- Disciplina = 'Signal and Continue'
- Java "monitors" nu sunt starvation-free – `notify()` deblocheaza un proces arbitrar.

Avantaje ale folosirii monitoarelor

- Abordare structurata
 - Implica mai putine probleme pt programator pentru a implementa excluderea mutuala;
- *Separation of concerns:*
 - *mutual exclusion for free,*
 - *condition synchronization -> condition variables*

Probleme

- *trade-off* -> suport pt programator si performanta
- •Disciplinele de semnalizare – sursa de confuzie;
 - *Signal and Continue* – conditia se poate schimba inainte ca procesul semnalizat sa intre in monitor
- *Nested monitor calls*:
 - Rutina r1 din M1 apeleaza rutina r2 din monitorul M2.
 - Daca r2 contine o operatie wait atunci excluderea mutuala trebuie relaxata si pentru M1 dar si pentru M2, ori doar pentru M2?

Variabile conditionale (CV)

- O abstractizare care permite sincronizarea conditionala;

Operatii: **wait**; **signal** ; [broadcast]

- O variabila conditionala **C** este asociata cu
 - o variabila de tip **Lock – m**
 - o coada
- Thread **t** apel **wait** =>
 - suspenda **t** si il adauga in coada lui **C** + deblocheaza **m** (op atomica)
- Atunci cand **t** isi reia executia **m** se blocheaza
- Thread **v** apel **signal** =>
 - se verifica daca este vreun thread care asteapta si il activeaza

Legatura cu monitor:

- Variabile conditionale pot fi asociate cu lacatul unui monitor (monitor lock);
 - Permit threadurilor sa astepte in interiorul unei sectiuni critice eliberand lacatul monitorului.

CV implementare orientativa

(Lock implementat ca si un semafor binar initializat cu 1)

```
class CV {  
    Semaphore s, x;  
    Lock m;  
    int waiters = 0;  
public CV(Lock m) {  
    // Constructor  
    this.m = m;  
    s = new Semaphore();  
    s.count = 0;    s.limit = 1;  
    x = new Semaphore();  
    x.count = 1;    x.limit = 1;  
    }  
    // x protejeaza accesul la variabila 'waiters'
```

```
    public void Wait() {  
        // Pre-condition: this thread holds "m"  
        //=> Wait se poate apela doar dintr-un cod  
        //sincronizat (blocat ) cu "m"  
        x.P(); {  
            waiters++; }  
        x.V();  
        m.Release();  
    (1)  
        s.P();  
        m.Acquire();  
    }  
    public void Signal() {  
        x.P(); {  
            if (waiters > 0)  
            { waiters--; s.V(); }  
            x.V();  
        }  
    }
```

Java and C++

Java

Interface Condition

Methods

[await\(\)](#)

The current thread suspends its execution until it is signalled or interrupted.

[await\(long time, TimeUnit unit\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.

[awaitNanos\(long nanosTimeout\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.

[awaitUninterruptibly\(\)](#)

The current thread suspends its execution until it is signalled (cannot be interrupted).

[await\(long time, TimeUnit unit\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified deadline elapses.

[signal\(\)](#)

This method wakes a thread waiting on this condition.

[signalAll\(\)](#)

This method wakes all threads waiting on this condition.

C++

[std::condition_variable](#)

[\(constructor](#)

`condition_variable();`

`condition_variable(const condition_variable&) = delete;`

Notification

[notify_one](#)

notifies one waiting thread
(public member function)

[notify_all](#)

notifies all waiting threads
(public member function)

Waiting

[wait](#)

template< class [Predicate](#) >
void wait([std::unique_lock](#)<[std::mutex](#)>& lock, [Predicate](#) pred);

[wait_for](#)

template< class Rep, class Period, class [Predicate](#) >
bool wait_for([std::unique_lock](#)<[std::mutex](#)>& lock,
const [std::chrono::duration](#)<Rep,Period>& rel_time,
[Predicate](#) pred);

[wait_until](#)

template< class Clock, class Duration >
wait_until([std::unique_lock](#)<[std::mutex](#)>& lock,
const [std::chrono::time_point](#)<Clock,
Duration>& timeout_time);

C++ example

```
#include <condition_variable>
#include <iostream>
#include <thread>

std::mutex a_mutex;
std::condition_variable condVar;
bool dataReady = false;

void waitingForWork(){
    std::cout << "Waiting\n ";
    std::unique_lock<std::mutex>
        lck(a_mutex);
    condVar.wait(lck, []{
        return dataReady; } );
    std::cout << "Running\n " ;
}
```

```
void setDataReady(){
    {
        std::lock_guard<std::mutex>
            lck(a_mutex);
        dataReady = true;
    }
    std::cout << "Data prepared\n";
    condVar.notify_one();
}

int main(){

    std::thread t1(waitingForWork);

    std::thread t2(setDataReady);

    t1.join();  t2.join();

}
```