Curs 4

Programare Paralela si Distribuita

Thread-safety
Concurenta
Deadlock, Starvation, Livelock
Mutex, Monitoare, Semafoare, Lacate

Forme de interactiune intre procese/threaduri

1. **comunicarea** între procese distincte
   -transmiterea de informaţii între procese

2. **sincronizarea** astfel încât procesele să aştepte informaţiile de care au nevoie şi nu sunt produse încă de alte procese/tread-uri
   - restricţii asupra evoluţiei în timp a unui proces/thread

Thread safety

Thread-safe

- ***Thread-safe class***
  - daca comportamentul instantelor sale este corect chiar daca este accesat din threaduri multiple indiferent de executia intretesuta a lor(interleaving)
    fara sa fie nevoie de sincronizari aditionale sau alte conditii impuse codului apelant.

  - sincronizarile sunt incapsulate in interior si astfel clientii clasei nu trebuie sa foloseasca altele speciale.

- Similar ***Thread-safe code***

Variabile locale

- Variabilele locale din thread-uri sunt stocate pe ***stiva*** fiecarui thread.
- Nu sunt partajate.
- Prin urmare toate variabilele primitive locale sunt *thread safe*.

Ex:

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

Referinte Locale

- Referintele nu sunt partajate (orice obiect este accesibil printr-o referinta).
- Obiectul referit este partajat (*shared heap*).
- Daca un obiect creat local nu se foloseste decat local in metoda care il creeaza atunci este *thread safe.*
- Daca un obiect creat local este transferat altor metode dar nu este transferat altor threaduri atunci este *thread safe*.

Cum se asigura ca nu va fi transferat altor threaduri???

Ex:

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

Shared Variables

- *shared memory*

- *(Java) ->* Toate atributele claselor, campurile statice si elementele tablourilor sunt stocate in heap ➔ shared.

- Doua operatii de acces (*read or write*) la aceeasi variabila se spune ca sunt in conflict daca cel putin unul este scriere (write).

Thread-Safe shared variables

- Daca mai multe thread-uri folosesc o variabila mutabila(modificabila) fara sa foloseasca sincronizari codul ***nu este safe***.

- Solutii:
  - eliminarea partajarii valorii variabilei intre threaduri
  - transformarea variabile in variabila_imutabila (var imutabile sunt thread-safe)
  - sincronizarea accesului la starea variabilei

Thread Signaling

- Permite transmiterea de semnale/mesaje de la unul thread la altul.
- Un thread poate astepta un semnal de la altul.

Signaling via Shared Objects

- Setarea unei variabile partajate- ***comunicare prin variabile partajate.***

```
public class MySignal{

  protected boolean hasDataToProcess = false;

  public synchronized boolean hasDataToProcess(){
    return this.hasDataToProcess;
  }

  public synchronized
      void setHasDataToProcess(boolean hasData){
    this.hasDataToProcess = hasData;
  }

}
```

Busy Wait

•Thread B asteapta ca data sa devina disponibila pentru a o procesa.

- => asteapta un semnal de la threadul A

=> hasDataToProcess() to return true.

•**Busy waiting NU implica o utilizare eficienta a CPU** (cu exceptia situatiei in cate timpul mediu de asteptare este foarte mic).

•este de dorit ca asteptarea sa fie inactiva (fara folosire procesor) –

• ceva similar ~~ sleep.

```
protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){
  //do nothing... busy waiting
}
```

Deadlock – Starvation - Livelock

- ***Deadlock***
  - situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecarea 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     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 accesibilila (*unfinetely often enable*d) 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 unui resurse critice. O resursă este critică dacă poate fi utilizată doar de catre singur proces la un moment dat;

- *sincronizarea pe condiţie*: se amână execuţia unui proces până când o anumită condiţie devine adevărată;

- *arbitrarea:* se evită accesul simultan din partea mai multor procesoare la aceeaşi locaţie de memorie.

->se realizează o secvenţializare a accesului, impunând aşteptarea până când procesul care a obţinut acces şi-a încheiat activitatea asupra locaţiei de memorie.

- 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)=> s este caracterizat de

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

Operatiile semafoarelor

- Gestiunea semafoarelor: prin 2 operaţii indivizibile
  - P(s) –este apelată de către procese care doresc să acceseze o regiune critică pt a obţine acces.
    - Efect: - incercarea obtinerii accesului procesului apelant la secţiunea critică si decrementarea valorii.

    - dacă v(s) <= 0 , procesul ce doreşte execuţia sectiunii critice aşteaptă
  - V(s)
    - Efect : incrementarea valorii semaforului.
    - se apelează la sfârşitul secţiunii critice şi semnifică eliberarea acesteia pt. alte procese.

- Succesiune instrucţ.:

  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 valorea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuţiei diferitelor procese.

  -c(s) o **multime de asteptare** la semafor - conţine referinţe la procesele care aşteaptă la semaforul s.

  +

  Operatiile P(s)/down si V(s)/up

Strong Semaphore

- Un semafor ‘puternic’ se poate defini ca o pereche {v(s),c(s)} unde:
  -v(s) este valorea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuţiei diferitelor procese.
  -c(s) o **coadă de aşteptare** la semafor - conţine referinţe la procesele care aşteaptă 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
```

Analiza

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

- Invariant:

  count >= 0

  count = k + #up – #down

- Demonstratie

Apel *down*:

• if *count* > 0 => *#down* este incrementat si *count* decrementat

• if *count* ≤ 0 => *down* nu se termina si *count* nu se modifica.

Apel *up:*

• if *blocked* (*is_empty)* => *#up* si *count* sunt incrementate;

• if *blocked* (*not is_empty)* => *#up* and *#down* sunt incrementate si *count* nu se modifica.

- *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

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

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
  - 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();
         }
}
```

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 operaţii asupra variabilelor şi
  - un corp (secvenţă de instrucţiuni).
    - Corpul este apelat la lansarea ‘programului’ şi produce valori iniţiale pentru variabilele-monitor (cod de initializare).
    - Apoi monitorul este accesat numai prin procedurile sale.

- codul de iniţializare este executat înaintea oricărui conflict asupra datelor ;
- numai una dintre procedurile monitorului poate fi executată la un moment dat;
- Monitorul creează o coadă de aşteptare 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 condiţie** se poate realiza prin mijloace definite explicit de către programator prin variabile de tip condiţie şi două operaţii:
  - signal (notify)
  - wait.
- Dacă un proces care a apelat o procedură de monitor găseşte condiţia falsă, execută operaţia wait (punere în aşteptare a procesului într-un şir asociat condiţiei şi elibereaza monitorul).
- in cazul în care alt proces care execută o procedură a aceluiaşi monitor găseşte/seteaza condiţia adevărată, execută o operaţie signal
  - procesul continuă dacă şirul de aşteptare este vid, altfel este pus în aşteptare şi se va executa un alt proces extras din şirul de aşteptare al condiţiei.

Monitor – object oriented view

Monitor class :

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

Instantiere clasa Monitor = monitor

- Attribute <->*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
    a_1 : $TYPE_1$
      . . .

    -- routine declarations
    r_1 (arg_1, ..., arg_k) **do** ... **end**
      . . .

  invariant
    -- monitor invariant
end

Implementare folosind un semafor: strong semaphore

entry : SEMAPHORE

Initializare v(entry) = 1

r (arg_1, ..., arg_k)
  do
    entry.down
    $body_r$
    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

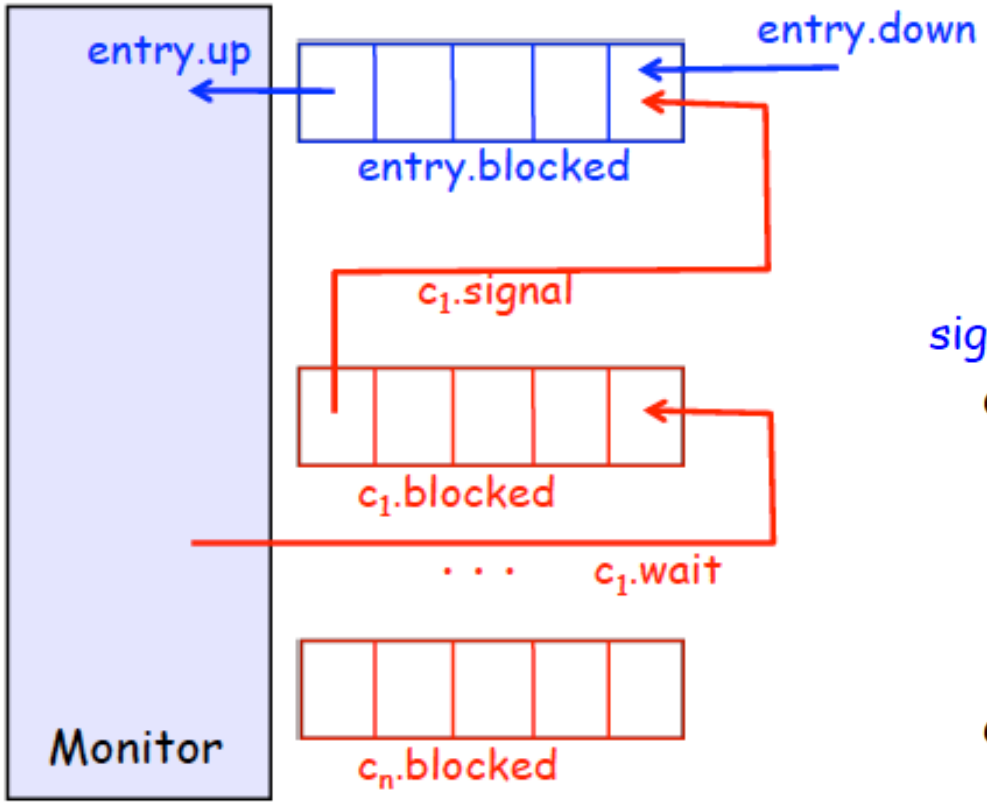
```
class CONDITION_VARIABLE
feature
   blocked: QUEUE
   wait
      do
         entry.up              -- release the lock on the monitor
         blocked.add(P)        -- P is the current process
         P.state := blocked    -- block process P
      end
   signal deferred end         -- behavior depends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

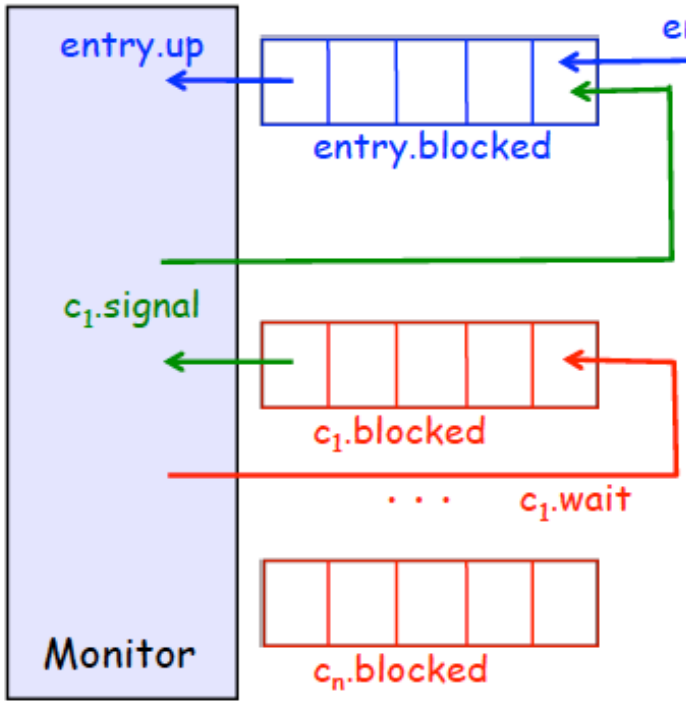


Pentru fiecare conditie => o coada

```
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 adevarat – 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 deblocat 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 semnalat 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?