Curs 7

Programare Paralela si Distribuita

Java: Sincronizare conditionata

Lock, Condition, Semaphore, ReadWriteLock, Exchange
Intreruperi

Exemple: deadlock, producator-consumator

Java Monitor

Operatii

- wait()
- notify() // signal
- notifyAll() // signal_all

Exemplul 1 –> Producator- Consumator / Buffer de dimensiune = 1

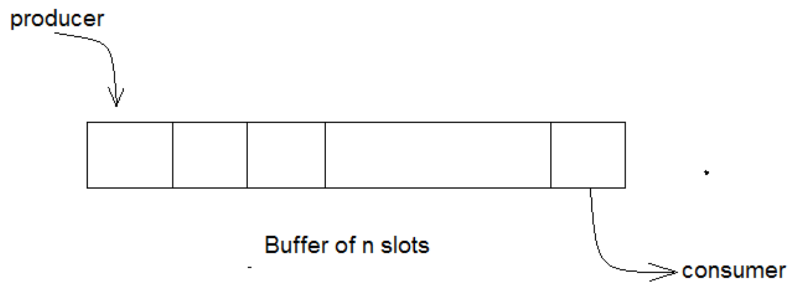
```
public class Producer extends Thread {

… ITER

    private Location loc;

    private int number; //id

    public Producer(Location c, int number) {
         loc = c;
         this.number = number;
    }

    public void run() {
        for (int i = 0; i < ITER; i++) {
                  loc.put(i);

         }
    }
}
```

```
public class Consumer extends Thread {

… ITER

    private Location loc;

    private int number; //id

    public Consumer(Location c, int number) {
       loc = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < ITER; i++) {
            value = loc.get();

        }
    }
}
```

```
public class Location {
   private int contents;                // shared data : didactic
   private boolean available = false;

/* Method used by the consumer to access the shared data */
   public synchronized int get() {
      while (available == false) {
         try {
            this.wait();          // Consumer enters a wait state until notified by the Producer
         } catch (InterruptedException e) { }
      }
      available = false;
      this.notifyAll();           // Consumer notifies Producers that it can store new contents
      return contents;
   }

/* Method used by the producer to store the shared data */
   public synchronized void put (int value) {
      while (available == true) {
         try {
            this.wait();          // Producer who wants to store contents enters
                       // a wait state until notified by the Consumer
         } catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      this.notifyAll();           // Producer notifies Consumer to come out
                       // of the wait state and consume the contents
   }
}
```

Exemplul 2: BlockingQueue : buffer size >1

```
class BlockingQueue {
  int n = 0;
  Queue data = ...;

  public synchronized Object remove() {
    // wait until there is something to read
    while (n==0)
            this.wait();

    n--;
    // return data element from queue
   }

   public synchronized void write(Object o) {
    n++;
    // add data to queue (consider that there is unlimited space)

     notifyAll();
   }
}
```



Lock

Oracle docs:

public interface **Lock**

- Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
- They allow ***more flexible structuring***, may have quite different properties, and may support multiple associated Condition objects.

Modifier and Type	Method and Description
void	lock()Acquires the lock.
void	lockInterruptibly()Acquires the lock unless the current thread is interrupted.
Condition	newCondition()Returns a new Condition instance that is bound to this Lock instance.
boolean	tryLock()Acquires the lock only if it is free at the time of invocation.
boolean	tryLock(long time, TimeUnit unit)Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
void	unlock()Releases the lock.

Lock (java.util.concurrent.locks.Lock)

```
public class Counter{

  private int count = 0;

  public int inc(){
    synchronized(this)
{
      return ++count;
    }
  }
}
```

```
public class Counter{
  private
    Lock lock = new ReentrantLock();
  private int count = 0;

  public int inc(){
    lock.lock();
        try{
                int newCount = ++count;  }
        finally{
                lock.unlock();
        }
    return newCount;
  }
}
```

Metode ale interfetei Lock

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

lockInterruptibly() – daca threadul este intrerupt in timp ce asteapta atunci apelul metodei se termina

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

Diferente Lock vs synchronized

- Nu se poate trimite un parametru la intrarea intr-un bloc *synchronized* => nu se poate preciza o valoare timp corespunzatoare unui interval maxim de asteptare-> timeout.

- Un bloc *synchronized* trebuie sa fie complet continut in interiorul unei metode
  - lock() si unlock() pot fi apelate in metode separate.

Reentrance

- Blocurile sincronizate in Java au proprietatea de a permite ‘reintrarea’ (*reentrant Lock*).
- Daca un thread intra intr-un bloc sincronizat si blocheaza astfel monitorul obiectului corespunzator, atunci threadul poate intra in alt cod sincronizat prin monitorul aceluiasi obiect.

```
public class Reentrant{
 public synchronized outer(){
  inner();
 }
 public synchronized inner(){
  //do something
 }
}
```

Class ReentrantLock
(java.util.concurrent.locks.ReentrantLock)

public class **ReentrantLock** extends Object implements Lock, Serializable

- A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

- A ReentrantLock is *owned* by the thread last successfully locking, but not yet unlocking it.
- A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread.
- The method will return immediately if the current thread already owns the lock.
  - This can be checked using methods isHeldByCurrentThread(), and getHoldCount().
-

Conditions in Java

- java.util.concurrent.locks
- Interface Condition

- Avantaj fata de “wait-notify” din monitorul definit pentru Object
- Imparte metodele (wait, notify , notifyAll) in obiecte distincte pentru diferite conditii
  - permite mai multe *wait-sets per object.*

Exemplu – Prod-Cons FIFO Buffer

```
class BoundedBuffer {
   static final MAX = 100;
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[MAX];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException
{
    lock.lock();
    try {
      while (count == items.length)
                         notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

```
public Object take() throws InterruptedException {
   lock.lock();
   try {
     while (count == 0)
       notEmpty.await();
     Object x = items[takeptr];
     if (++takeptr == items.length) takeptr = 0;
     --count;
     notFull.signal();
     return x;
   } finally {
     lock.unlock();
   }
  }
}
```

Intreruperi

- O intrerupere (*interrupt*) este o indicatie pentru un thread ca ar trebui sa se opreasca si … sa faca altceva (de ex. sa se termine).

public void interrupt()

public static boolean interrupted()

public boolean isInterrupted()

"There is no way in Java to terminate a thread unless the thread exits by itself."

Intreruperi

- mecanismul de intrerupere foloseste un flag intern -> the *interrupt status*.
- Atunci cand se apeleaza Thread.interrupt se seteaza acest flag.
- Atunci cand se verifica intreruperea prin metoda statica Thread.interrupted, <*interrupt status*> este sters.
- Metoda nestatica isInterrupted, care este folosita de catre un thread pt a verifica statusul (*interrupt status*) al altuia nu schimba flagul.
- Prin conventie, orice metoda care se termina (exit) aruncand o exceptie de tip InterruptedException sterge "interrupt status".
- Totusi este posibil ca acesta sa fie imediat setat din nou de catre alt thread care invoca o metoda *interrupt*.

Exemplu

```
public class SimpleThreads {

static void threadMessage(String message) {
      String threadName =          Thread.currentThread().getName();
      System.out.format("%s: %s  %n",      threadName,        message);
   }

   private static class MessageLoop      implements Runnable {
      public void run() {
         String importantInfo[] = {
            "Studentii sunt prezenti.",
            "Examenul este greu.",
            "Vacanta este asteptata.",
            "Exista concurenta."
         };
         try {
            for (int i = 0;   i < importantInfo.length;   i++) {
               // Pause for 4 seconds
               Thread.sleep(4000);
               // Print a message
               threadMessage(importantInfo[i]);
            }
         } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
         }
      }
   }
```

```
    public static void main(String args[])
     throws InterruptedException {

     // Delay, in milliseconds before
     // we interrupt MessageLoop
     // thread (default one hour).

     long patience = 1000 * 60 * 60;



 threadMessage(
    "Starting MessageLoop thread");

 long startTime = System.currentTimeMillis();

 Thread t = new Thread(new MessageLoop());

  t.start();

  threadMessage(
    "Waiting for MessageLoop thread to finish");
```

```
while (t.isAlive()) {
   threadMessage("Still waiting...");

       t.join(1000);

   if ((
(System.currentTimeMillis() - startTime) > patience)
       && t.isAlive())
 {
             threadMessage("Tired of waiting!");
             t.interrupt();
             // Shouldn't be long now
             // -- wait indefinitely

             t.join();
          }
        }
        threadMessage("Finally!");
    }
}
```

Interactiuni: Waits, Notification, Interruption

- Notificarile nu se pot pierde din cauza intreruperilor.

- Presupunem ca un set de threaduri **s** este in *wait set* a lui ***m***, si alt thread executa notificare pe **m**.

Atunci fie:

  - cel putin un thread din **s** iese normal din *wait*, sau
  - toate threadurile din **s** ies din wait aruncand exceptie InterruptedException

- Daca un thread este atat intrerupt cat si trezit prin notificare si el iese din wait aruncand o exceptie atunci un alt thread din *wait set* va fi notificat.

- Daca thread ***t*** a fost sters din ***wait*** *set* a lui **m** din cauza unei intreruperi atunci *interruption status* al lui t este setat la false si se iese din wait cu aruncarea unei exceptii de tip InterruptedException.

Conditionare - reguli

- folosirea operatiilor wait doar in cicluri care se termina atunci cand anumite conditii logice sunt indeplinite!

- fiecare thread trebuie sa determine o ordine intre evenimentele care pot cauza ca el sa fie sters din *wait set*.

De exemplu: daca *t* este in *wait set* al obiectului ***o*,** atunci cand apare atat o intrerupere a lui ***t*** cat si o notificare a lui **o**, trebuie sa existe o ordine intre aceste evenimente.

- ÷ Daca *interrupt* este considerata prima, atunci pana la urma t iese din wait aruncand InterruptedException, si alt thread din *wait set* a lui *o* (daca mai exista vreunul) va primi notificarea.
- ÷ Daca sunt invers ordonate atunci **t** iese normal din wait si intreruperea este in asteptare (pana se verifica starea).

Concluzii – legate de interactiune intreruperi cu mecanismul wait-notify

Un thread t poate fi sters din *wait set* al unui obiect *o* ca urmare a uneia din urm. actiuni (si apoi isi va relua executia*)* :

- ◆ O actiune *notify* asupra lui *o* in care *t* este selectat spre a fi sters din *wait set*.
- ◆ O actiune notifyAll asupra lui *o* .
- ◆ O actiune de intrerupere realizata de *t*.
- ◆ Trecerea timpului (argument) specificat la apelul lui *wait*.
- ◆ Operatii "spurious wake-ups" (*no apparent reason*) –/rare.

(https://en.wikipedia.org/wiki/Spurious_wakeup)

Nested monitor lockout

- Thread 1 blocheaza obiectul A
  - Thread 1 blocheaza B (in timp se ramane blocajul pe A)
    - Thread 1 decide sa astepte un semnal de la un alt thread
    - Thread 1 apeleaza B.wait() => elibereaza B dar nu si pe A.

- Thread 2 trebuie sa blocheze atat pe A cat si pe B (in ordine) pentru a trimite un semnal catre Thread 1.
- Thread 2 nu poate bloca pe A (pt ca Thread 1 are blocajul lui A).
- Thread 2 ramane blocat indefinit asteptand ca A sa fie eliberat.

- Thread 1 ramane blocat indefinit asteptand un semnal de la Thread 2, si astfel nu elibereaza pe A, etc.

Semaphore
(java.util.concurrent.Semaphore)

- Semafor binar (=> excludere mutuala)

```
Semaphore semaphore = new Semaphore(1);

//critical section
semaphore.acquire();
…
semaphore.release();
```

- Fair/Strong Semaphore

```
Semaphore semaphore = new Semaphore(1, true);
```

ReadWriteLock

- Read Access -> daca nici un thread nu scrie si nici nu cere acces pt scriere.
- Write Access -> daca nici un thread nici nu scrie nici nu citeste.

public interface **ReadWriteLock**

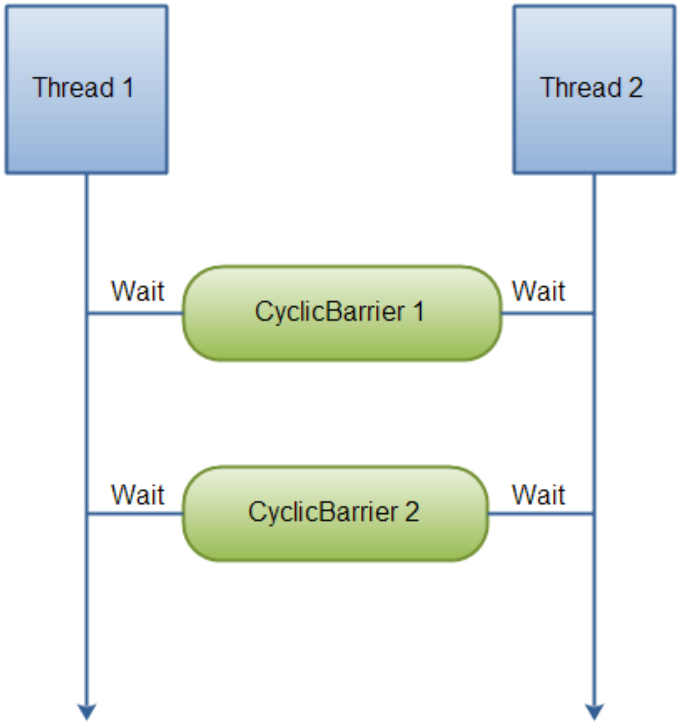
- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
- The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.
- The write lock is exclusive.

Bariera de sincronizare

```
CyclicBarrier barrier = new CyclicBarrier(2);
// 2 = no_of_threads_to_wait_for

barrier.await();

barrier.await(10, TimeUnit.SECONDS);
```

Bariera de sincronizare:

- Bariera secventiala – pt implementare se fol. in general 2 variabile – {no_threads(0..n), state(stop/pass)}
- Bariera ierarhica (tree-barrier)



Jacob Jenkov Tutorial

java.util.concurrent>

java.util.concurrent.CyclicBarrier (java documentation)

- A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other.
- The barrier is called cyclic because it can be re-used after the waiting threads are released.
- A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.
  - CyclicBarrier(int parties)
  - CyclicBarrier(int parties, Runnable barrierAction)

java.util.concurrent.CountDownLatch (java documentation)

- A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

- A CountDownLatch is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately.
  - This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier.

- A CountDownLatch is a versatile synchronization tool and can be used for a number of purposes.
  - A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate: all threads invoking await wait at the gate until it is opened by a thread invoking countDown().
  - A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.

- A useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass.

CountDownLatch (Java)

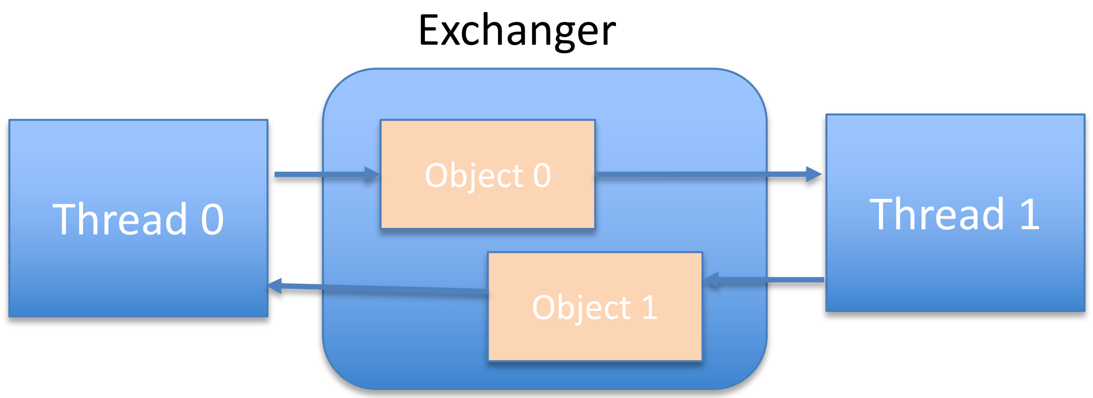
```
final CountDownLatch latch = new CountDownLatch(5);

        // making two threads for 2 services
        Thread serviceOneThread = new Thread(new ServiceOne(latch, 2));
        Thread serviceTwoThread = new Thread(new ServiceTwo(latch, 3));

        serviceOneThread.start();
        serviceTwoThread.start();

        // latch waits till the count becomes 0
        // this way it can make sure that the execution of main thread only
        // finishes once 2 services have executed
        try {
            latch.await(); …
```

Conceptul de intalnire (**Rendez-vous**)

- Conceptul de întâlnire (rendez-vous) a fost introdus initial in limbajul Ada pentru a facilita comunicarea între două task-uri.

a) Procesul B este gata să transmită informaţiile, dar procesul A nu le-a cerut încă. In acest caz, procesul B rămâne în aşteptare până când procesul A i le cere.
b) Procesul B este gata să transmită informaţiile cerute, iar procesul A cere aceste date. In acest caz, se realizează un *rendez-vous*, cele două procese lucrează sincron până când îşi termină schimbul, după care fiecare îşi continuă activitatea independent.
c) Procesul A a lansat o cerere, dar procesul B nu este în măsură să-i furnizeze informaţiile solicitate. In acest caz, A rămâne în aşteptare până la întâlnirea cu B.

Java Exchanger <-> Rendez-Vous



Jacob Jenkov Tutorial

Thread-0 exchanged Object0 for Object1
Thread-1 exchanged Object1 for Object0

Exchanger
(java doc)

public V exchange(V x) throws InterruptedException

- Waits for another thread to arrive at this exchange point (unless the current thread is interrupted), and then transfers the given object to it, receiving its object in return.
- If another thread is already waiting at the exchange point then it is resumed for thread scheduling purposes and receives the object passed in by the current thread.
  - The current thread returns immediately, receiving the object passed to the exchange by that other thread.

If no other thread is already waiting at the exchange then the current thread is disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread enters the exchange; or
- Some other thread interrupts the current thread.

If the current thread:

- has its interrupted status set on entry to this method; or
- is interrupted while waiting for the exchange,
  - then InterruptedException is thrown and the current thread's interrupted status is cleared.

```
Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =    new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =    new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();
new Thread(exchangerRunnable2).start();
```

```
public class ExchangerRunnable implements Runnable{
  Exchanger exchanger = null;
  Object    object    = null;

  public ExchangerRunnable(Exchanger exchanger, Object object) {
    this.exchanger = exchanger;
    this.object = object;
  }

  public void run() {
    try {
      Object previous = this.object;

         Object received =
                          this.exchanger. exchange (this.object);
         this.object = received;

     System.out.println(
           Thread.currentThread().getName() +
           " exchanged " + previous + " for " + this.object
      );
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

Class SynchronousQueue (->*Rendez-vous*) Java doc

java.lang.Object
java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
        java.util.concurrent.SynchronousQueue<E>

A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. ***A synchronous queue does not have any internal capacity, not even a capacity of one.*** You cannot peek at a synchronous queue because an element is only present when you try to remove it; ***you cannot insert an element (using any method) unless another thread is trying to remove it***; you cannot iterate as there is nothing to iterate. The *head* of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements [https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html]

- boolean **offer(E e)**
  - Inserts the specified element into this queue, if another thread is waiting to receive it.
- void **put(E o)**
  - Adds the specified element to this queue, waiting if necessary for another thread to receive it.

- **E** **poll()**
  - Retrieves and removes the head of this queue, if another thread is currently making an element available.

Exemplu

```
public class SynchronousQueueDemo{ public static void main(String args[]) {
final SynchronousQueue<String> queue = new SynchronousQueue<String>();
            Thread producer = new Thread("PRODUCER") {
                  public void run() {
                        String event = "FOUR";
                        try { queue.put(event); // thread will block here
                              System.out.printf("[%s] published event : %s %n”,
                                                Thread .currentThread().getName(), event); }
                        catch (InterruptedException e) {
                              e.printStackTrace(); } } };
                  producer.start(); // starting publisher thread

            Thread consumer = new Thread("CONSUMER") {
                  public void run() {
                        try { String event = queue.take(); // thread will block here
                              System.out.printf("[%s] consumed event : %s %n",
                                                Thread .currentThread().getName(), event); }
                        catch (InterruptedException e) { e.printStackTrace(); }
                        }
                  };
                  consumer.start(); // starting consumer thread
            }
```

Sincronizare <-> comunicare

Programele comunică între ele nu numai pentru a-şi comunica informaţii sub formă de mesaje ci şi pentru a se sincroniza.

=>

Un semnal de sincronizare poate fi considerat şi el că este un mesaj fără conţinut ce se transmite între programe.

Cum se realizează acest lucru?

Rendez- vous

Process-view

Procese secvenţiale comunicante. Rendez-vous simetric.

task-ul A: o comandă de emitere mesaj → SUSPENDARE ←
task-ul B: o comandă de recepţie de mesaj;

task-ul B: o comandă de recepţie de mesaj → SUSPENDARE ←
task-ul A:o comandă de emisie de mesaj;

task-uri sincronizate→ datele (mesajul) sunt transferate

Procese distribuite. Rendez-vous asimetric

- Comunicarea şi sincronizarea între programele concurente se realizează în acest caz similar cu apelarea prin nume de către programul emiţător a unei proceduri incluse în programul receptor,
  - lista cu parametrii asociaţi acestui apel fiind folosită ca un "canal" de comunicare pentru transmiterea de date între cele două programe.

- Doar programul apelant trebuie să cunoască numele programului apelat, nu şi invers.