Curs 5

Programare Paralela si Distribuita

Forme de sincronizare - Java

Monitor in Java

- Fiecare obiect din Java are un mutex care poate fi blocat sau deblocat in blocurile sincronizate:
- *Bloc sincronizat*

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

:> sau *metoda* (obiectul blocat este “this”)

```
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.

Synchronized Static Methods

```
Class Counter{
static int count;
  public static synchronized void add(int value){
     count += value;
}
 public static synchronized void decrease(int value){
     count -= value;
}
}
```

-> blocare pe *class object of the class* => Counter.class

- Ce se intampla daca sunt mai multe metode statice sincronizate ?

fine-grained synchronization

```
public class Counter {
   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc1() {
      synchronized(lock1) {
         c1++;
      }
   }

   public void inc2() {
      synchronized(lock2) {
         c2++;
      }
   }
}
```

•Ce se intampla daca lock1 sau lock2 se modifica?

- Ce se intampla daca sunt metode de tip instanta sincronizate dar si metode statice sincronizate?

Exemplu

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Transformare => fine-grained synchronization

```
public class Counter {
  private long c = 0;
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void inc() {
    synchronized(lock1) {
      c++;
    }
  }
  public void dec() {
    synchronized(lock2) {
      c--;
    }
  }
}
```

- Este corect?

- Ce probleme exista?

Nonblocking Counter

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
         while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

Exemplificari

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

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

```
public class Producer extends Thread {

… ITER

   private CubbyHole cubbyhole;

   private int number; //id

   public Producer(CubbyHole c, int number) {
      cubbyhole = c;
      this.number = number;
   }

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

       }
   }
}
```

```
public class Consumer extends Thread {

… ITER

   private CubbyHole cubbyhole;

   private int number; //id

   public Consumer(CubbyHole c, int number) {
      cubbyhole = c;
      this.number = number;
   }

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

      }
   }
}
```

```
public class CubbyHole {
    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 {
                wait();           // Consumer enters a wait state until notified by the Producer
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();          // Consumer notifies Producer that it can store new contents
        return contents;
    }

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

exemplu: BlockingQueue : buffer size >1

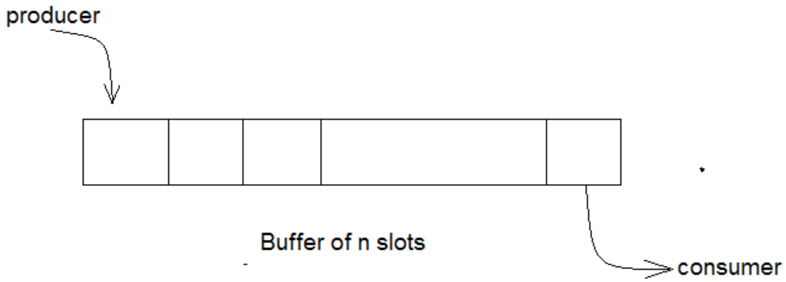
```
class BlockingQueue {
   int n = 0;
   Queue data = ...;

   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

       notifyAll();
    }
}
```



Missed Signals- Starvation

- Apelurile metodelor notify() si notifyAll() nu se salveaza in cazul in care nici un thread nu asteapta atunci cand sunt apelate.

- Astfel semnalul notify se pierde.

- Acest lucru poate conduce la situatii in care un thread asteapta nedefinit, pentru ca mesajul corespunzator de notificare se pierde.

- Propunere:
  - Evitarea problemei prin salvarea semnalelor in interiorul clasei care le trimite.
- =>analiza!

```
public class MyWaitNotify2{

 MonitorObject myMonitorObject = new MonitorObject();
 boolean wasSignalled = false;

 public void doWait(){
  synchronized(myMonitorObject){
   if(!wasSignalled){
    try{
     myMonitorObject.wait();
    } catch(InterruptedException e){...}
   }
   //clear signal and continue running.
   wasSignalled = false;
  }
 }

 public void doNotify(){
  synchronized(myMonitorObject){
   wasSignalled = true;
   myMonitorObject.notify();
  }
 }
}
```

Condition in Java

- java.util.concurrent.locks
- Interface Condition

- Imparte metodele monitorul definit pentru Object (wait, notify , notifyAll) in obiecte distincte pentru a permite mai multe *wait-sets per object.*

Exemplu

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

  final Object[] items = new Object[100];
  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();
    }
  }
}
```

Similar C++11

```
struct BoundedBuffer {
   int* buffer;   int capacity;
   int front, rear, count;
   std::mutex lock;
   std::condition_variable not_full;
   std::condition_variable not_empty;
. . .
```

```
void deposit(int data){
  std::unique_lock<std::mutex> l(lock);

//se asociaza cu un lock(mutex)
//si cu o functie booleana
  not_full.wait(l, [this](){return count != capacity; });

      buffer[rear] = data;
      rear = (rear + 1) % capacity;
      ++count;

      not_empty.notify_one();
}
```

```
int fetch(){
      std::unique_lock<std::mutex> l(lock);

      not_empty.wait(l, [this](){return count != 0; });

      int result = buffer[front];
      front = (front + 1) % capacity;
      --count;

      not_full.notify_one();

      return result;
  }
```

Semaphore

(java.util.concurrent.Semaphore)

- Semafor binar (=> excludere mutuala)

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

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

- Fair Semaphore

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

Exemplu

```
Thread loop = new Thread(
 new Runnable() {
      public void run() {
           while (true) {
                          if (Thread.interrupted()) {         break;       }
           // Continue to do what it should be done
        }
    }
  }
);
…
loop.start();
loop.interrupt();
```

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();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

Metode ale interfetei Lock

lock()
lockInterruptibly()
tryLock()
tryLock(long timeout, TimeUnit timeUnit)
unlock()

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.

Lock 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
  }
}
```

Read / Write Lock

- 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.

- Exemplu:
  - ThreadSafeArrayList

Non blocking…

```
public class ConcurrentStack<E> {
   AtomicReference<Node<E>> head =
                 new AtomicReference<Node<E>>();
 static class Node<E> {
      final E item;
      Node<E> next;
      public Node(E item) { this.item = item; }
   }
   public void push(E item) {
      Node<E> newHead = new Node<E>(item);
      Node<E> oldHead;
      do {
         oldHead = head.get();
         newHead.next = oldHead;
      }
  while
(!head.compareAndSet(oldHead, newHead));
```

```
   }

   public E pop() {
      Node<E> oldHead;
      Node<E> newHead;
      do {
         oldHead = head.get();
         if (oldHead == null)
            return null;
         newHead = oldHead.next;
      } while (!
head.compareAndSet(oldHead,newHead));
      return oldHead.item;
   }

}
```

Michael-Scott non-blocking queue algorithm: Insertion

```
public class LinkedQueue <E>

  private static class Node <E> {
    final E item;
    final AtomicReference<Node<E>> next;

    Node(E item, Node<E> next) {
      this.item = item;
      this.next = new
              AtomicReference<Node<E>>(next);
    }
  }


 private AtomicReference<Node<E>> head =
        new AtomicReference<Node<E>>(
           new Node<E>(null, null));

 private AtomicReference<Node<E>> tail =
                                  head;
```

```
public boolean put(E item) {
  Node<E> newNode = new Node<E>(item, null);
while (true) {
    Node<E> curTail = tail.get();
    Node<E> residue = curTail.next.get();
    if(curTail == tail.get()) {
    if (residue == null) /* A */ {
     if (curTail.next.compareAndSet(
                  null, newNode)) /* C */
     {
        tail.compareAndSet(curTail, newNode)
                           /* D */ ;
        return true;
      }
    } else {
        tail.compareAndSet(curTail, residue) /
* B */;
    }
   }
  }
 }
}
```