Unit 5.- Other Synchronization Tools

Concurrency and Distributed Systems

Teaching Unit Objectives

- Know other synchronization tools
  - Semaphores
  - Barriers
- Make use of the *java.util.concurrent* library, that offers high-level synchronization tools
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (Executors)

Bibliography

- Java concurrency: *High-level concurrency objects → Lock Objects, Executors, Concurrent Collections, Atomic Variables*

  http://docs.oracle.com/javase/tutorial/essential/concurrency/

- Also available at Poliformat: **Java Concurrency** document

- In Spanish:
  - Chapter 5, course book ("Concurrencia y Sistemas Distribuidos").

Disadvantages of Java basic primitives

- **Activity**: List different disadvantages of Java basic primitives (e.g. monitors) for task synchronization.
  - a) Limitations of Java basic primitives for mutual exclusion
  - b) Limitations of Java basic primitives for conditional synchronization

Disadvantages of Java basic primitives

- a) Limitations of the Java basic primitives, related with **mutual exclusion:**
  1. We cannot specify a maximum time of waiting when requesting the entry to the monitor.
     - If the monitor is busy, the thread remains waiting and it cannot be interrupted voluntarily.
  2. We cannot ask about the state of the monitor before requesting its entry.
     - Sometimes it might be interesting to check whether the lock (or monitor) is free or busy, without blocking if busy (e.g. using *tryLock*).
       - Specially useful to avoid possible deadlocks.
       - With *synchronized* methods, this check is not possible.

Disadvantages of Java basic primitives

3. The tools that ensure mutual exclusion are oriented to blocks.
   - i.e. they are oriented to complete methods or to smaller sections of code.
   - We cannot acquire a lock inside a method and release it inside another method.
4. We cannot extend its semantics
   - Example: in the readers-writers problem, we cannot employ these constructions to define mutual exclusion among writer threads or among writers and readers, but not among multiple readers.

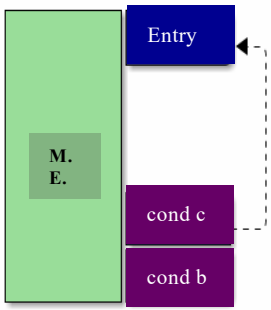
Disadvantages of Java basic primitives

- b) Limitations of the Java basic primitives, related with **conditional synchronization**:

1. There can be one single condition in each monitor.
   - All threads that are suspended in a monitor go to the same queue, independently of the motive (condition) for which they get suspended.

2. Java employs the Lampson and Redell variant
   - The programmer is forced to use a structure of type:

     while (logical expression) wait();



A thread has to check the state of the monitor and suspend (if required).

java.util.concurrent library

- J2SE 5.0 includes the **java.util.concurrent** library
  - This library provides high-level constructions
  - They guarantee:
    - **More productivity**: facilitates development/maintenance of concurrent applications of quality
    - **More features**: more efficient and scalable than typical implementations
    - **More reliability**: extensive checks against deadlocks, race conditions, starvation

Components of *java.util.concurrent*

- This library includes several useful components
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (*Executors*)

Locks

- ***java.util.concurrent.locks*** provides different classes and interfaces for the management and development of multiple types of *locks*.
  - ReentrantLock
  - ReentrantReadWriteLock
  - …..

Locks

- Features:
  - Allows specifying if a **fair** management of the waiting queue of the lock is required.
    - i.e. reactivate suspended threads in FIFO
  - Provides several types of locks with different semantics
    - Example: locks oriented to mutual exclusion, locks that resolve the readers-writers problem.
  - Offers a **tryLock**() method that does not suspend the invoker if the lock has been closed by another thread.
    - This breaks the "hold and wait" Coffman's condition (seen in Unit 4 – deadlocks).

Locks: ReentranLocks

- Example of *lock* class provided: ***ReentrantLock***
  - Implements a reentrant lock:
    - Inside the protected section by the lock we can employ again this same lock without problems of getting blocked by it.
  - Solves limitations of the *synchronized* sentence
    - Very efficient implementation
    - Allows specifying a **maximum waiting time** when acquiring a lock → *tryLock()* with *timeout*
    - Allows defining different condition variables → ***newCondition()*** method
    - Allows open and close *locks* in different methods of the application → lock(), unlock()
      - It does not restrict the use of lock objects to the construction of monitors.
    - Provides support for interruptions on threads waiting to acquire a lock
      - We can interrupt waits using *Thread.interrupt ()*

Locks: ReentrantLocks

- Drawbacks:
  - With a basic monitor in Java, the management of *locks* is implicit
    - The programmer does not need to care about opening and closing locks.
    - But with ***ReentrantLock***, the programmer must ensure to open the related lock when releasing a resource that is being used in mutual exclusion.
  - All exceptions that can be generated inside a critical section protected by a ***ReentrantLock*** must be controlled.
    - The code associated to the exception must ensure to execute the *unlock*() method on this *ReentrantLock*.

Locks: ReentrantLocks

- *Recommendation*: follow this structure for input and output protocolos

```
Lock x = new ReentrantLock();
x.lock();
try {
.. //Critical Section (where the state of the object is updated)
} finally {
   x.unlock()
}
```

Input protocol

Output protocol

Condition Variables

- Remember that the ***Object*** class includes special methods for synchronization between threads: *wait*, *notify*, *notifyAll*.
  - It is easy to use them incorrectly
  - There is an interaction between notification and locking
    - To *wait* or *notify* we need to previously close the "lock" of the Object.

- With *java.util.concurrent* we have higher level constructions, and this reduces the need to use *wait/notify/notifyAll*.

Condition Variables

- The ***Condition*** interface allows declaring any number of condition variables inside a lock
  - In many cases, employing multiple condition variables allows developing a more readable and efficient code
    - For example, in the Producer/Consumer problem

- The ***Lock*** object acts as a factory for the condition variables related to it
  - We can only define **condition** variables inside a lock

Condition Variables

- ***newCondition***() method of ***ReentrantLock*** class
  - allows generating all required waiting queues (i.e. conditions)
  - returns an object that implements the ***Condition*** interface

- Methods of **Condition** interface:
  - **await**(): suspends a thread in the condition
    - Includes variants for specifying maximum waiting times, deadlines, interumption management.
  - **signal**(): notifies the occurrence of the expected event to **one** of the threads who was waiting it.
  - **signalAll**(): notifies the occurrence of the expected event to **all** threads waiting in the condition.
- Note: in basic monitors, similar methods are *wait*(), *notify*(), *notifyAll*()

Example.- Lock and Condition

```
class BufferOk implements Buffer {
    private int elems, head, tail, N;
    private int[] data;
    Condition notFull, notEmpty;
    ReentrantLock lock;

    public BufferOk(int N) {
        data= new int[N];
        this.N=N;
        head = tail = elems = 0;
        lock= new ReentrantLock();
        notFull=lock.newCondition();
        notEmpty=lock.newCondition();
    }
```

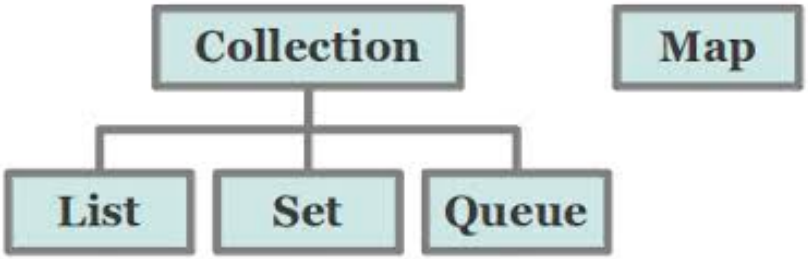
```
public int get() {
    int x;
    try {
        lock.lock();
        while (elems==0) {
            System.out.println("consumer waiting ..");
            try {notEmpty.await();}
             catch(InterruptedException e) {}
        }
        x=data[head];  head= (head+1)%N;  elems--;
        notFull.signal();
    } finally {lock.unlock();}
    return x;
}

public void put(int x) {
    try{
        lock.lock();
        while (elems==N) {
            System.out.println("producer waiting ..");
            try {notFull.await();}
             catch(InterruptedException e) {}
        }
        data[tail]=x; tail= (tail+1)%N;  elems++;
        notEmpty.signal();
    } finally {lock.unlock();}
}
```

Components of *java.util.concurrent*

- This library includes several useful components
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (*Executors*)

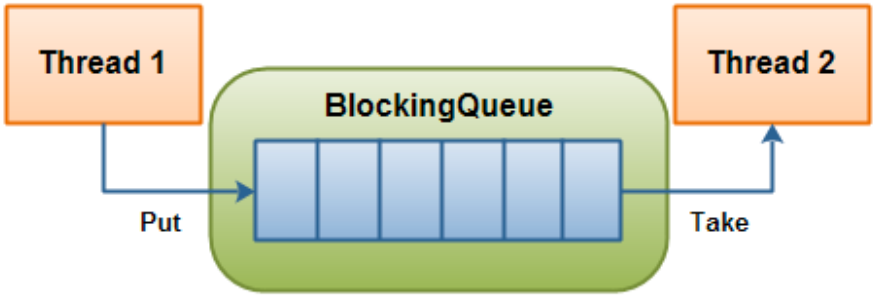
Concurrent Collections

- Many applications require access to collections of objects:
  - ***List*** *interface (dynamic list)* → *ArrayList, LinkedList, Vector…*
  - ***Map*** *interface (associative array)* → EnumMap, HashMap, TreeMap
  - ***Queue*** *interface (FIFO policy)*
- They are **not *thread-safe*** *(they cannot be shared in a safe way by several threads)*



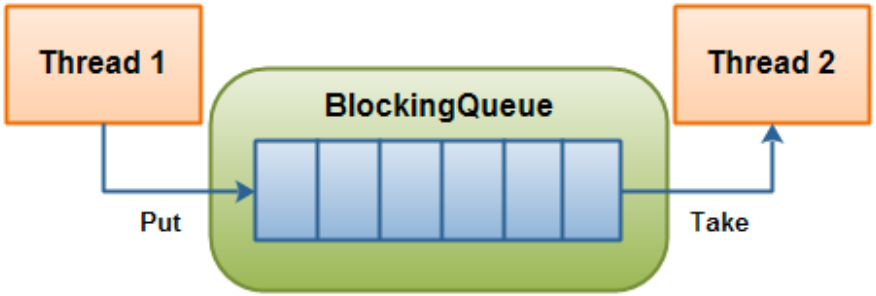
Concurrent Collections

- *Java.util.concurrent* includes **Thread-safe** versions:
  - ***ConcurrentHashMap***, ***ConcurrentSkipListMap*** *classes* implementing ***Map*** interface
  - ***CopyOnWriteArrayList***, to implement ***List*** interface
  - ***BlockingQueue*** interface, extends ***Queue***.



Concurrent Collections

- *Java.util.concurrent* includes **Thread-safe** versions:
  - Efficient concurrent implementations of *Queue*:
    - ArrayBlockingQueue
    - ConcurrentLinkedQueue
    - LinkedBlockingQueue
    - PriorityBlockingQueue
    - DelayQueue
    - SynchronousQueue



Concurrent Collections.- Example: BlockingQueue

- Methods of **BlockingQueue** interface:

	Generates Exception	Returns Value (true/false)	Blocks	TimeOuts
Insert	*add(o)*	*offer(o)*	***put(o)***	*offer(o, timeout, timeunit)*
Extract	*remove(o)*	*poll(o)*	***take(o)***	*poll(timeout,timeunit)*
Check	*element(o)*	*peek(o)*		

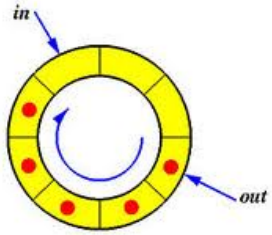
- Other methods:
  - **remainingCapacity**(): number of elements that can still be inserted in the queue.
  - **contains**(): returns true if the queue contains a specific object.
  - **drainTo**(): deletes all elements of the queue and adds them to the specified collection.

Concurrent Collections.- Example: BlockingQueue

- Methods of **BlockingQueue** interface:
  - For inserting elements in the queue:
    - **add**(): if there is not enough space, an exception is generated.
    - **offer**(): if there is not enough space, it returns false.
    - **put**(): if there is not enough space, waits.
  - For extracting elements from the queue:
    - **take**(): extracts and deletes the first element. Waits if needed until there is any element to extract.
    - **poll**(): same as *take*, but if there are no elements in the queue waits as maximum as the specified interval.
    - **remove():** deletes the specified object
    - **peek**(): returns first element of the queue, but does not delete it.

Concurrent Collections.- Example: BlockingQueue

- Example: **Producer/consumer problem**
  - Buffer is a ***BlockingQueue:***



BlockingQueue queue;

  - The thread that wants to consume an item has to wait if the queue is empty **consume(queue.take())**
  - The thread that puts and item hast to wait if there is not any free space **queue.put(produce())**

Conditional Synchronization controlled by the BlockingQueue methods:
- take()
- put()

Concurrent Collections.- Example: BlockingQueue

```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while(true) { queue.put(produce()); }
     } catch (InterruptedException ex) {...}
   }
   Object produce() { ... }
 }
```

```
class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while(true) { consume(queue.take()); }
     } catch (InterruptedException ex) {...}
   }
   void consume(Object x) { ... }
 }
```

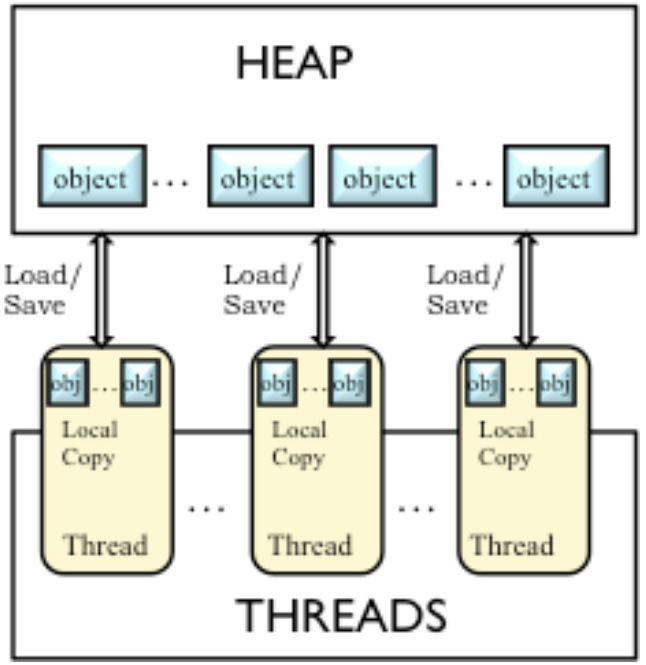
```
class Setup {
   void main() {
     BlockingQueue q = new
             someQueueImpl();
     Producer p = new Producer(q);
     Consumer c1 = new Consumer(q);
     Consumer c2 = new Consumer(q);
     new Thread(p).start();
     new Thread(c1).start();
     new Thread(c2).start();
   }
 }
```

Components of *java.util.concurrent*

- This library includes several useful components
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (*Executors*)

Atomic Variables (*java.util.concurrent.atomic*)

- Java Memory Model:



- Non atomic instructions→ int i=0; i++;
  - Load int value from heap (local copy=0);
  - Increment value in 1 (local copy=1);
  - Load value to heap (i=1);

Critical Section

requires *synchronized*

protect it with ReentrantLock

Atomic Variables (*java.util.concurrent.atomic*)

- Defines classes that support safe concurrent access to simple variables
  - Make use of low level atomic instructions supported by processors
  - Ensure: if a thread reads a specific value of a variable, any other thread can lately read an older value of same variable
- Primitive types (*AtomicBoolean*, *AtomicInteger*, *AtomicLong*)
- References (*AtomicReference*)
- Allows considering different operators in an atomic way
  - Operation ++ on an integer is not atomic, but *AtomicInteger* has an atomic increment operation (*getAndIncrement*, *incrementAndGet*)
  - Also operations like *getAndSet*, *compareAndSet*, etc.

Atomic Variables (*java.util.concurrent.atomic*)

- Useful for:
  - implementing concurrent algorithms efficiently
  - implementing counters
  - generating sequences of numbers
- Implementation is very efficient
  - More than we can obtain using *synchronized*

Atomic Variables - Example: **AtomicLong**

- ***AtomicLong**: allows updating atomically a* long value
- *Methods of this class:*
  - *addAndGet() → adds given value to current one, atomically*
  - *decrementAndGet(), getAndDecrement() → decrements 1, atomically*
  - *incrementAndGet(), getAndIncrement() → increments 1, atomically*
  - *getAndSet()*
  - *compareAndSet()*
  - *toString()*
  - ...

Atomic Variables.- Example

- With our own class

```
class ID {
  private static long nextID = 0;
  public static synchronized
    long getNext() {
      return nextID++;
  }
}
public class ExCounter extends Thread{
  ID counter;
  public ExCounter(ID c) {counter=c;}
  public void run() {
    System.out.println("counter value: "+
      (counter.getNext()));
  }
  public static void main(String[] args) {
    ID counter= new ID();
    new ExCounter(counter).start();
    new ExCounter(counter).start();
    new ExCounter(counter).start();
  }
}
```

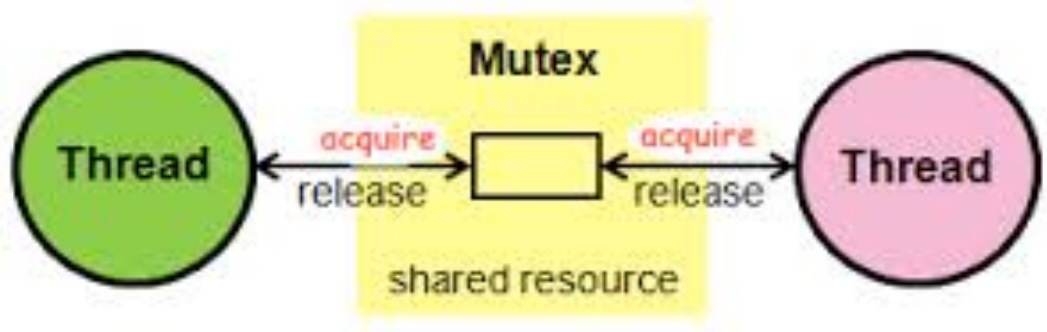
- With AtomicLong
  - Methods *addAndGet, compareAndSet, decrementAndGet, incrementAndGet, getAndDecrement, getAndIncrement, getAndSet, toString, get, …*

```
public class ExCounter extends Thread{
  AtomicLong counter;
  public ExCounter(AtomicLong c) {counter=c;}
  public void run() {
    System.out.println("counter value: "+
      (counter.getAndIncrement()));
  }
  public static void main(String[] args) {
    AtomicLong counter= new AtomicLong(0);
    new ExCounter(counter).start();
    new ExCounter(counter).start();
    new Ex Counter(counter).start();
  }
}
```

Components of *java.util.concurrent*

- This library includes several useful components
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (*Executors*)

- The **Semaphore** class enables synchronization among thread
  - Instead of P() and V(), in Java we use **acquire**() and **release**() method
    - ***acquire():*** blocks the thread until a permit is available, and then takes it.
    - ***release():*** adds a permit, potentially releasing a blocking acquirer.



Semaphores

- We can employ semaphores to protect critical sections (e.g. mutex) or to synchronize threads.
  - They include a **counter** initialized in the creation of the semaphore.
  - If initialized to **ONE** → ensures mutual exclusion

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

```
sem.acquire();
```

Critical Section

```
sem.release();
```

Semaphores

- We can employ semaphores to protect critical sections (e.g. mutex) or to synchronize threads.
  - They include a **counter** initialized in the creation of the semaphore.
  - If initialized to a **positive value >1** → restricts the grade of concurrence

```
Semaphore sem = new Semaphore(4, true);
```

```
sem.acquire();
```

Restricted Section

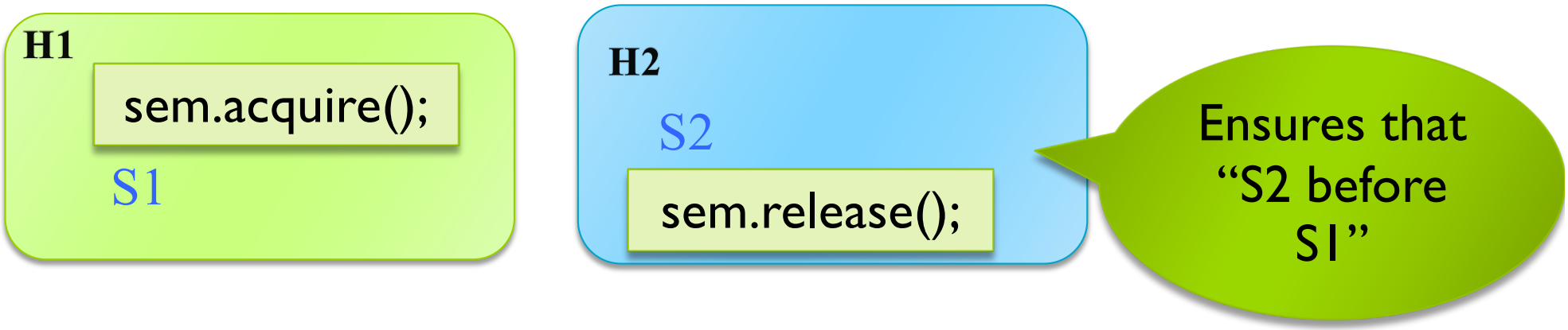
```
sem.release();
```

Maximum 4 threads at the same time

Semaphores

- We can employ semaphores to protect critical sections (e.g. mutex) or to synchronize threads.
  - They include a **counter** initialized in the creation of the semaphore.
  - If initialized to **Zero** → ensure a specific order of execution between two or more threads

Semaphore sem = new Semaphore(0, true);



Synchronization: Semaphore class

- **Example 1:** protecting critical sections

```
import java.util.concurrent.Semaphore;
class Process2 extends Thread {
        private int id;
        private Semaphore sem;

        public Process2(int i, Semaphore s) {
            id = i; sem = s;}

       private void busy() {
          try {
            sleep(new
java.util.Random().nextInt(500));
           } catch (InterruptedException e) {}
       }
       private void msg(String s) {
              System.out.println("Thread " + id + s);}

       private void noncritical() {
              msg(" is NON critical");
              busy();
       }
```

```
private void critical() {
       msg(" entering CS");
       busy();
       msg(" leaving CS");}

public void run() {
     for (int i = 0; i < 2; ++i) {
                    noncritical();
                    try {sem.acquire();}
                     catch (InterruptedException e) {}
                    critical();
                    sem.release();
              }
      }
public static void main(String[] args) {
     Semaphore sem = new Semaphore(1, true);
     for (int i = 0; i < 4; i++)
            new Process2(i, sem).start();
     } }
```

Fairness: threads selected in FIFO order

Synchronization: Semaphore class

- **Example 1:** protecting critical sections

```
import java.util.concurre
class Process2 extends Th
      private int id;
      private Semaphore s

      public Process2(int i, S
          id = i; sem = s;}

      private void busy() {
        try {
          sleep(new
java.util.Random().nextInt(500
        } catch (InterruptedExc
      }
      private void msg(String s)
            System.out.println("Thre

      private void noncritical() {
            msg(" is NON critical");
            busy();
      }
```

```
C:\CSD>java Process2
Thread 0 is NON critical
Thread 3 is NON critical
Thread 1 is NON critical
Thread 2 is NON critical
Thread 2 entering CS
Thread 2 leaving CS
Thread 2 is NON critical
Thread 1 entering CS
Thread 1 leaving CS
Thread 1 is NON critical
Thread 3 entering CS
Thread 3 leaving CS
Thread 3 is NON critical
Thread 0 entering CS
Thread 0 leaving CS
Thread 0 is NON critical
Thread 2 entering CS
Thread 2 leaving CS
Thread 3 entering CS
Thread 3 leaving CS
Thread 0 entering CS
Thread 0 leaving CS
Thread 1 entering CS
Thread 1 leaving CS
```

```
e void critical() {
nsg(" entering CS");
usy();
nsg(" leaving CS");}

 void run() {
r (int i = 0; i < 2; ++i) {
            noncritical();
            try {sem.acquire();}
             catch (InterruptedException e) {}
            critical();
            sem.release();
      }
}
 static void main(String[] args) {
emaphore sem = new Semaphore(1, true);
r (int i = 0; i < 4; i++)
    new Process2(i, sem).start();
} }
```

Synchronization: Semaphore class

- **Example 2**: useful for synchronization

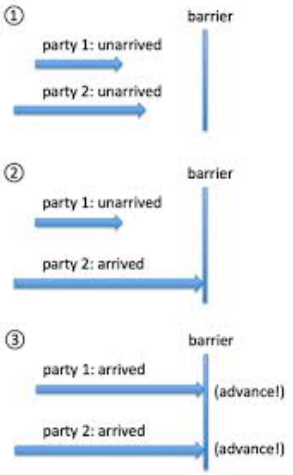
```
import java.util.concurrent.Semaphore;
class ProdCons extends Thread {
      static final int N=6; // buffer size
      static int head=0, tail=0, elems=0;
      static int[] data= new int[N];
      static Semaphore item= new Semaphore(0,true);
      static Semaphore slot= new Semaphore(N,true);
      static Semaphore mutex= new
Semaphore(1,true);

      public static void main(String[] args) {
            new Thread(new Runnable() { // producer
                public void run() {
                    for (int i=0; i<10; i++) {
                          busy();
                          put(i);
             }}}).start();
             new Thread(new Runnable() {// consumer
                public void run() {
                    for (int i=0; i<10; i++) {
                         busy();
                          System.out.println(get());
            }}}).start();
      }
```

```
private static void busy() {
      try {sleep(new java.util.Random().nextInt(500));}
      catch (InterruptedException e) {}
}

public static int get() {
    try {item.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    int x=data[head]; head= (head+1)%N;  elems--;
    mutex.release();
    slot.release();
    return x;
}

public static void put(int x) {
    try {slot.acquire();} catch (InterruptedException e) {}
    try {mutex.acquire();} catch (InterruptedException e) {}
    data[tail]=x; tail= (tail+1)%N;  elems++;
    mutex.release();
    item.release();
  }
}
```

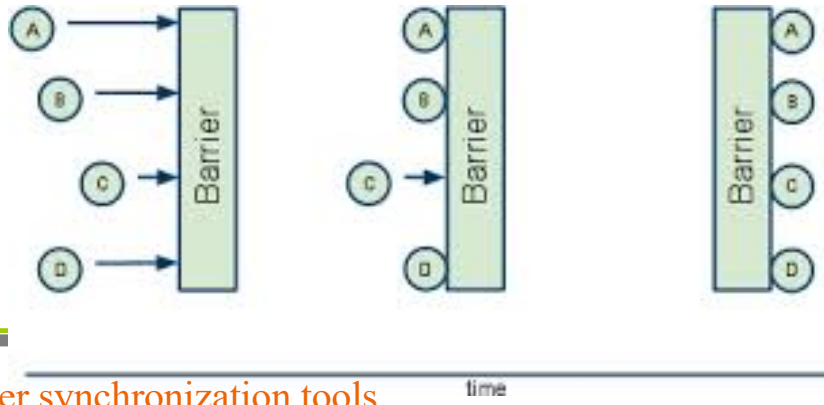
Barriers

- They allow synchronizing multiple executing threads.
- In *java.util.concurrent* there are two types of barriers:
  - CyclicBarrier
  - CountdownLatch

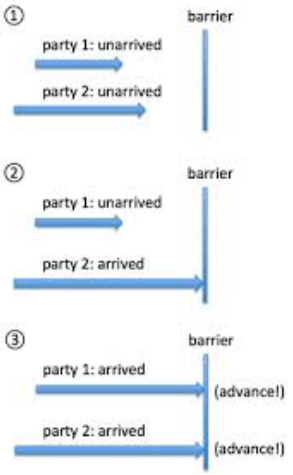
Barriers: CyclicBarrier



- **CyclicBarrier:**
  - They allow a group of threads to wait for each other
    - In a common point: the barrier
  - The barrier gets open when a certain number of threads reach the barrier
    - ***await***() method of the barrier: suspends a thread
    - When the last thread invokes ***await****()*, then barrier gets open → all waiting threads are awaken
  - The number of threads to be suspended is specified in the barrier constructor



Barriers: CyclicBarrier



- **CyclicBarrier:**
  - It is reusable (this is why it is named *cyclic*)
    - When the threads are released (i.e. allowed to cross the barrier), the initial conditions are restored
    - It allows threads to wait again at the barrier point
  - Very useful for iterative applications
    - E.g.- in each iteration a Worker thread processes a row of the matrix, and a coordinator thread (Solver) combines the results when all workers have finished.

Barriers: CyclicBarrier → Example

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
      while (!done()) {
          processRow(myRow);
          try {  barrier.await();
          } catch (InterruptedException e)
              {return;}
        catch (BrokenBarrierException e)
              {return;}
   } } }
```

Main method: **await**()
→threads wait at the barrier

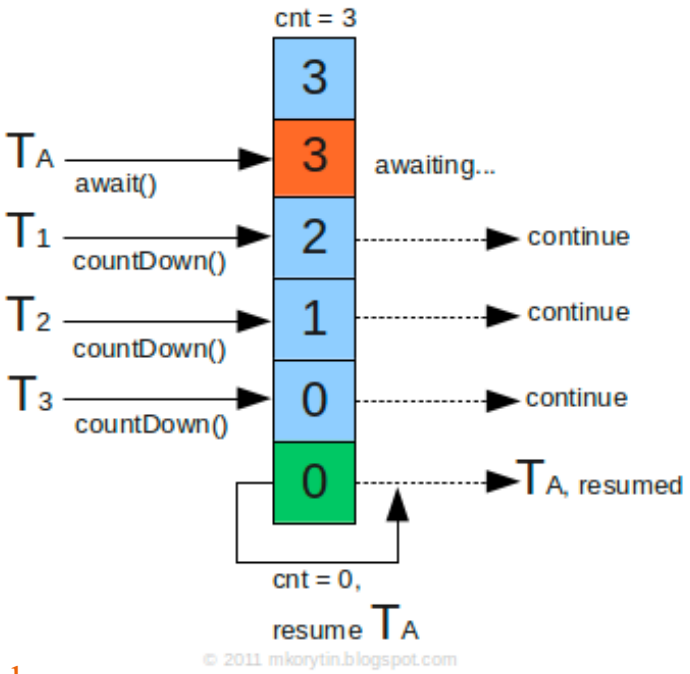
```
public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
      new Runnable() {
        public void run() {
            mergeRows(...);
        }
     });

   for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();
      waitUntilDone();
  }
}
```

Maximum Number of threads that will wait at the barrier

Optional: Task to be done when barrier gets open (before reactivating threads)

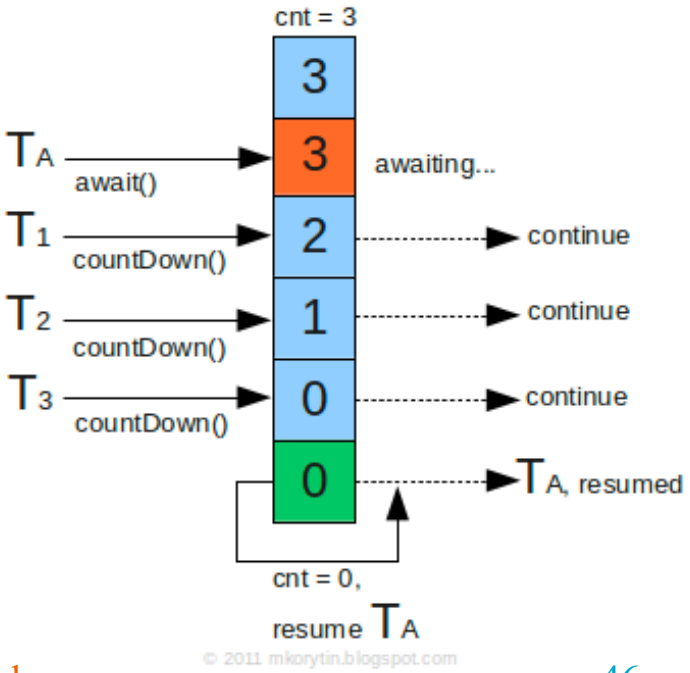
Barriers: CountDownLatch

- **CountDownLatch:**
  - Allows suspending a group of threads, which remain waiting until the occurrence of some event that must be generated by another thread.
  - There is a counter, which is initialized in the barrier constructor.
  - The barrier is created initially closed.



Barriers: CountDownLatch

- **CountDownLatch:**
  - Methods that offers:
    - **await**(): threads get blocked in the barrier while it is closed (i.e. its counter is higher than zero).
    - **countDown**(): decrements in one unit the value of the counter (if it was higher than zero).
      - If the counter reaches 0, then the barrier opens, releasing all waiting threads.
  - One-shot phenomenon:
    - The counter cannot be reset
    - Once it reaches 0, it keeps like that.
    - For a reusable barrier, we must employ *CyclicBarrier*



Barriers: CountDownLatch → Example

```
class Driver { // …
   void main() throws InterruptedException {
     CountDownLatch startSignal = new CountDownLatch(1);
     CountDownLatch doneSignal = new CountDownLatch(N);
     for (int i = 0; i < N; ++i)
       new Thread(new Worker(startSignal, doneSignal)).start();
     doSomethingElse();
     startSignal.countDown();
     doSomethingElse();
     doneSignal.await();
   }
 }
```

Why do they have these values?

startSignal: : for allowing Worker threads to begin all at once

```
class Worker implements Runnable {
   private final CountDownLatch startSignal;
   private final CountDownLatch doneSignal;
   Worker(CountDownLatch start, CountDownLatch done) {
      startSignal = start;
      doneSignal = done;
   }
   public void run() {
         try {  startSignal.await();
                doWork();
                doneSignal.countDown();
          } catch (InterruptedException ex) {return;}
   }
   void doWork() { … }
 }
```

Barriers: CountDownLatch → Example

```
class Driver { // …
   void main() throws InterruptedException {
     CountDownLatch startSignal = new CountDownLatch(1);
     CountDownLatch doneSignal = new CountDownLatch(N);
     for (int i = 0; i < N; ++i)
       new Thread(new Worker(startSignal, doneSignal)).start();
     doSomethingElse();
     startSignal.countDown();
     doSomethingElse();
     doneSignal.await();
   }
 }
```

Why do they have these values?

doneSignal: where Driver thread waits till all Workers finish their work

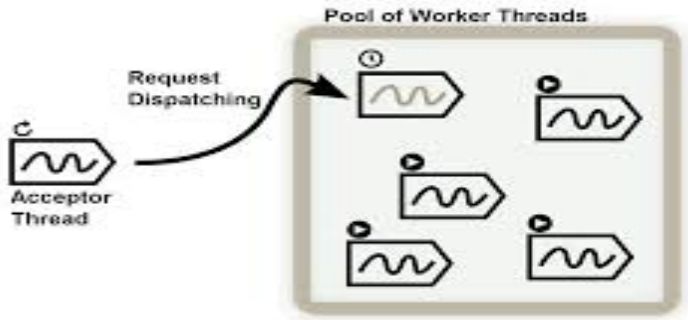
```
class Worker implements Runnable {
   private final CountDownLatch startSignal;
   private final CountDownLatch doneSignal;
   Worker(CountDownLatch start, CountDownLatch done) {
      startSignal = start;
      doneSignal = done;
   }
   public void run() {
      try {  startSignal.await();
             doWork();
             doneSignal.countDown();
      } catch (InterruptedException ex) {return;}
   }
   void doWork() { … }
 }
```

Components of *java.util.concurrent*

- This library includes several useful components
  - Locks
  - Condition Variables
  - Concurrent Collections
  - Atomic Variables
  - Synchronization: Semaphores and Barriers
  - Environment for thread execution (*Executors*)

Environment for thread execution (Executors)

- **Creating threads** is an expensive, resource-intensive and potentially time-consuming operation.
- ***Executor*** **Interface**: offers and environment for creation and management of threads in Java
  - It allows its invocation, planning, execution and control of execution policies.



- Advantages:
  - Prevents excessive consumption of resources
  - A library (**Executor**) is already implemented to create tasks in a very flexible way
  - This type of environment is widely used

Environment for thread execution (Executors)

- Executor interface offers the ExecutorService subinterface, that allows creating differerent types of **Executors** :
  - A single thread (***SingleThreadExecutor***): uses a single working thread that operates in an unlimited queue.
    - It ensures that tasks are executed sequentially, and that no more than one task is active at any given time.

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
```

Creates a single thread in which tasks are queued and executed sequentially

  - A *thread-pool* (e.g. for a server)
    - It allows to maintain a set of already created threads, recycling them to execute new tasks.

```
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
```

Creates a pool with 10 threads

Environment for thread execution (Executors)

- Simple example of using an ExecutorService:

```
ExecutorService executorService =
                    Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

Environment for thread execution (Executors)

- There are different ways to delegate tasks for the execution of an ExecutorService:
  - execute(Runnable), submit(Runnable), submit(Callable), invokeAny(…), invokeAll(…)

- We will focus here on the use of **Runnable**:

```
Runnable command = new Runnable() {
    public void run() {
        doLongWork();
    }
};

ExecutorServe executor = Executors.newSingleThreadExecutor();
executor.execute(command);
```

Environment for thread execution (Executors)

```
package executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {

   public static void main(String[] args) {

      ExecutorService executor = Executors.newFixedThreadPool(10);
      executor.execute(() -> doLongWork("hi 1"));
      executor.execute(() -> doLongWork("hi 2"));
      executor.execute(() -> doLongWork("hi 3"));
   }

   private static void doLongWork(String hello) {
      System.out.println("Running " + hello);
      try {
         Thread.sleep(1000l);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Note: we have used here **Lambda** notation (but we will not see this in detail)

Environment for thread execution (Executors)

- By default, **ExecutorService** remains "listening" to new tasks
  - It remains active to receive new tasks and assign them to its pool
  - Even if the main thread ends, ExecutorService is still active.

- There are two ways to stop **ExecutorService**:
  - shutdown() → allows you to stop the Executor's threads.
    - It does not stop immediately, but will no longer accept new tasks
    - When all the threads have finished with their current tasks, then it will stop
  - shutdownNow() → tries to stop all running tasks
    - Tasks submitted but not processed are ignored
    - Attempts to stop running tasks (although some may run to completion)
    - Returns the list of tasks that were not completed

Environment for thread execution (Executors)

- Important: even if the mian thread of the application ends, ExecutorService will continue to run (and therefore the application) as long as it is active.

- We can use awaitTermination() to wait for ExecutorService to stop.
  - Example:

```
executorService.shutdown();
executorService.awaitTermination();
```

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Identify the drawbacks of the basic Java primitives.
  - Describe the tools of the *java.util.concurrent* package that enable the development of concurrent applications:
    - Illustrate the usage of locks and conditions. Compare them with using monitors.
    - Understand the use of *thread-safe* concurrent collections (example: *BlockingQueue*)
    - Describe the behaviour of atomic classes. Example: compare *AtomicInteger* with using monitors.
    - Illustrate the usage of Semaphores for mutual exclusion and synchronization.
    - Illustrate the behavior of barriers, comparing *CyclicBarrier* and *CountDownLatch*.
    - Know the environment for thread execution (*Executors*)