Unit 3. Synchronization Primitives

Monitors

To operate with concurrency there are several tools that are provided either by the Operative System or by programming languages, also some other tools that such as Active-waiting or Test & Set.

In addition to the `locks` seen in the previous unit, there are more tools to implement, for example, **conditional synchronization**.

Let's take profit of the possibilities of OOP, and create classes that will define objects to be shared between threads to orchestrate the synchronization. Those classes will be called **monitors**.

Mutual exclusion

Monitors will ensure that there will be a queue of waiting threads when a thread is currently using the resource. Also, to solve synchronization problems, it will develop conditional synchronization by means of condition variables that will trigger certain events depending on their state.

For instance, in the example of readers-writers, boxes are monitors, and both classes will have to wait when there is someone modifying or reading the data.

```
public synchronized read() {
    while(someLock) {
        wait();
    }

    doSomething();

    notifyAll();
}
```

Monitor Variants

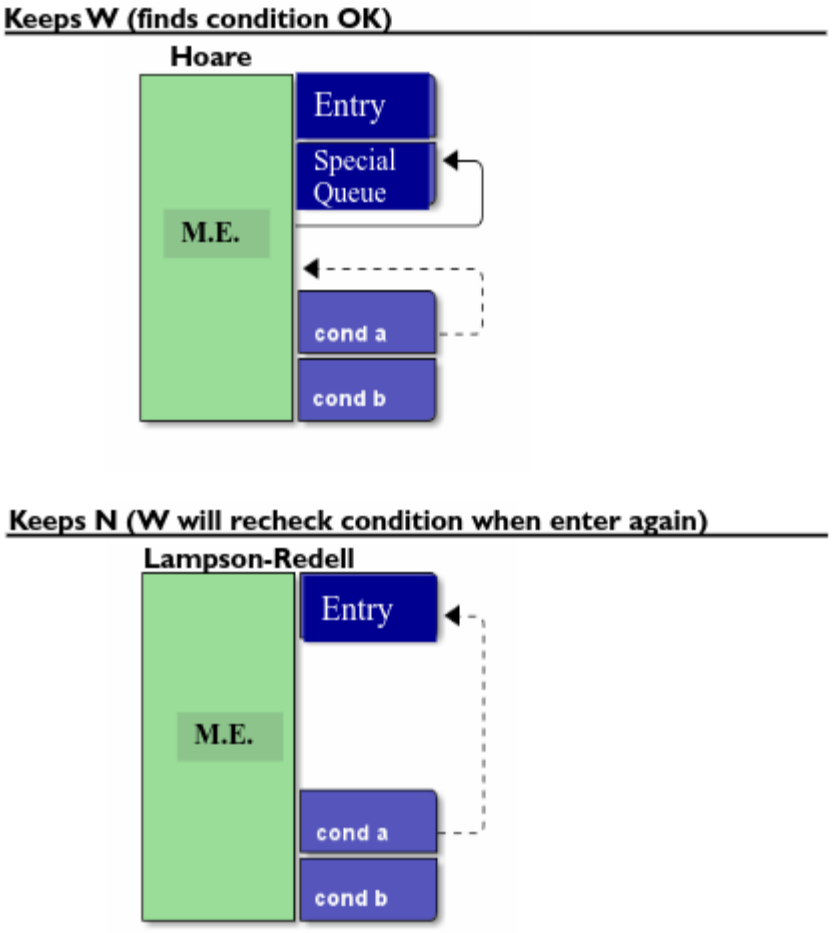
Even though, there are some problems with monitors and that is, where do threads wait? Where are those who are waiting in the entry and those that notify the rest of a change in the state. This is seen under two models:

Hoare model

With W waiting on cond c and N executing notify and reactivating W, process N will go to a **special queue** outside the monitor while W will remain active inside the monitor.

Lampson-Redell model

Similar to Hoare's model, but instead of N waiting outside the monitor, it stays active inside, while W will wait in the outside of the monitor, in the Entry part.



Monitor properties and methods

Attributes and methods inside monitors can be implemented in two ways, each with its upsides and cons. For example, we could use a waiting queue for each element, or a single queue for all elements. The first one is more efficient in terms of waiting, and also it uses a `if` statement instead of a `while`, on the other hand the second creates greater queues.

Monitors in Java

There are two ways to use them, via the `concurrent` package or the implicit way.

Implicit way

Basically, with the keyword `synchronized`,we already ensure **mutual exclusion** with an implicit lock. Also there is a waiting queue that we can operate with by using methods `wait()`, `notify()` and `notifyAll()`. This limits to no more locks or waiting queues.

Properties

- All attributes should be private
- All non-private methods must use `synchronized`
- Use wait, notify and notifyAll for synchronized methods

```
while(condition){
    wait();
}
...
notifyAll(); // It is more recommended to use notifyAll to wa
```

Nested Calls

When nesting synchronized conditionals, the risk of a deadlock is higher

```
// Deadlock example
/** Consider two threads t1 and t2
        Two monitors with the corresponding variables p1 and
*/

class BCell {
```

```
    int value;
    public synchronized void getValue() {
        return value;
    }
    public synchronized void setValue(int i) {
        value=i;
    }
    public synchronized void swap(BCell x) {
        int temp= getValue();
        setValue(x.getValue());
        x.setValue(temp);
    }
}

/** If t1 invokes p1.swap(p2) accesses p1 and t2 calls p2.swa
        t1 will close p1 and wait for p2, as p2 is closed by
        But since t2 is waiting for p1 to be opened, they are
 */
```