Java's Monitor & Hoare vs Mesa Monitors

Continues the discussion of the differences between a mutex and a monitor and also looks at Java's implementation of the monitor.

We discussed the abstract concept of a monitor in the previous section and now let's see the working of a concrete implementation of it in Java.

Java's Monitor

In Java every object is a condition variable and has an associated lock that is hidden from the developer. Each java object exposes `wait()` and `notify()` methods.

Before we execute `wait()` on a java object we need to lock its hidden mutex. That is done implicitly through the **synchronized** keyword. If you attempt to call `wait()` or `notify()` outside of a synchronized block, an `IllegalMonitorStateException` would occur. It's Java reminding the developer that the mutex wasn't acquired before wait on the condition variable was invoked. `wait()` and `notify()` can only be called on an object once the calling thread becomes the ***owner*** of the monitor. The ownership of the monitor can be achieved in the following ways:

- the method the thread is executing has synchronized in its signature
- the thread is executing a block that is synchronized on the object on which wait or notify will be called
- in case of a class, the thread is executing a static method which is synchronized.

Bad Synchronization Example 1

In the below snippet, `wait()` is being called outside of a synchronized block, i.e. without implicitly locking the associated mutex. Running the code results in **IllegalMonitorStateException**

Java — Press + to interact — main.java — Saved

```
class BadSynchronization {

    public static void main(String args[]) throws InterruptedException {
        Object dummyObject = new Object();

        // Attempting to call wait() on the object
        // outside of a synchronized block.
        dummyObject.wait();
    }
}
```

Run

Illegal Monitor Exception

Bad Synchronization Example 2

Bad Synchronization Example 2

Here's another example where we try to call `notify()` on an object in a synchronized block which is synchronized on a different object. Running the code will again result in an **IllegalMonitorStateException**

Java — Press + to interact — main.java — Saved

```
class BadSynchronization {

    public static void main(String args[]) {
        Object dummyObject = new Object();
        Object lock = new Object();

        synchronized (lock) {
            lock.notify();

            // Attempting to call notify() on the object
            // in synchronized block of another object
            dummyObject.notify();
```

Run

Hoare vs Mesa Monitors

So far we have determined that the idiomatic usage of a monitor requires using a while loop as follows. Let's see how the design of monitors affects this recommendation.

```
while( condition == false ) {
    condVar.wait();
}
```

Once the asleep thread is signaled and wakes up, you may ask why does it need to check for the condition being false again, the signaling thread must have just set the condition to true?

In **Mesa monitors** - Mesa being a language developed by Xerox researchers in the 1970s - it is possible that the time gap between thread B calls `notify()` and releases its mutex **and** the instant at which the asleep thread A, wakes up and reacquires the mutex, ***the predicate is changed back to false by another thread different than the signaler and the awoken threads!*** The woken up thread competes with other threads to acquire the mutex once the signaling thread B **empties** the monitor. On signaling, thread B doesn't give up the monitor just yet; rather it continues to own the monitor until it exits the monitor section.

In contrast, **Hoare monitors** - Hoare being one of the original inventor of monitors - the signaling thread B ***yields*** the monitor to the woken up thread A and thread A ***enters*** the monitor, while thread B sits out. This guarantees that the predicate will not have changed and instead of checking for the predicate in a while loop an if-clause would suffice. The woken-up/released thread A immediately starts execution when the signaling thread B signals that the predicate has changed. No other thread gets a chance to change the predicate since no other thread gets to enter the monitor.

Java, in particular, subscribes to Mesa monitor semantics and the developer is always expected to check for condition/predicate in a while loop. Mesa monitors are more efficient than Hoare monitors.

← Back lesson
Mutex vs Monitor

Completed Next →
Semaphore vs Monitor