Mutex vs Monitor

In this lesson, let's learn what a monitor is and how it is different than a mutex. Monitors are advanced concurrency constructs and specific to language frameworks.

Mutex vs Monitor

Continuing our discussion from the previous section on locking and signaling mechanisms, we'll now pore over an advanced concept, the **monitor**. Note that in the theory of operating systems, a monitor is implemented using condition variables, and a monitor can be associated with more than one condition variable. In Ruby, a monitor can be associated with multiple condition variables. However, in some languages, e.g., C# and Java, there's no type equivalent of a theoretical condition variable. Instead, every object inherently implements a condition variable. The discussion that follows isn't specific to Ruby but about the general concept of a monitor.

Concisely, a monitor is a mutex and then some. Monitors are generally language-level constructs, whereas mutex and semaphore are lower-level or OS-provided constructs.

To understand monitors, let's first see the problem they solve. Usually, in multi-threaded applications, a thread needs to wait for some program predicate to be true before it can proceed forward. Think about a producer/consumer application. If the producer hasn't produced anything, the consumer can't consume anything. So the consumer must ***wait on*** a predicate that lets the consumer know that something has indeed been produced. What could be a crude way of accomplishing this? The consumer could repeatedly check in a loop for the predicate to be set to true. The pattern would resemble the pseudocode below:

```
def busyWaitFunction()
    // acquire mutex

    while predicate is false
      // release mutex
      // acquire mutex
    end
    // do something useful
    // release mutex
end
```

Within the while loop, we'll first release the mutex giving other threads a chance to acquire it and set the loop predicate to true. Before we check the loop predicate again, we make sure we have acquired the mutex. This works but is an example of ***"spin waiting"*** which wastes a lot of CPU cycles. Let's see how condition variables solve the spin-waiting issue.

Condition Variables

Mutex provides mutual exclusion. However, at times mutual exclusion is not enough. We want to test for a predicate with a mutually exclusive lock so that no other thread can change the predicate when we test for it, but if we find the predicate to be false, we'd want to wait on a condition variable till the predicate's value is changed. This is the solution to spin waiting.

Conceptually, each condition variable exposes two methods `wait()` and `signal()`. The `wait()` method, when called on the condition variable, will cause the associated mutex to be atomically released and the calling thread would be placed in a **wait queue**. There could already be other threads in the **wait queue** that previously invoked `wait()` on the condition variable. Since the mutex is now released, it gives other threads a chance to change the predicate that will eventually let the thread that was just placed in the wait queue to make progress. As an example, say we have a consumer thread that checks for the size of the buffer, finds it empty and invokes `wait()` on a condition variable. The predicate in this example would be ***the size of the buffer***.

Now imagine a producer places an item in the buffer. The predicate, the size of the buffer, just changed and the producer wants to let the consumer threads know that there is an item to be consumed. This

producer thread would then invoke `signal()` on the condition variable.

The `signal()` method, when called on a condition variable, causes one of the threads that has been placed in the **wait queue** to get ready for execution. Note that we didn't say the woken up thread starts executing. It just gets ready - and that could mean being placed in the ready queue. It is only **after the producer thread, which calls the `signal()` method has released the associated mutex that the thread in the ready queue starts executing.** The thread in the ready queue must wait to acquire the mutex associated with the condition variable before it can start executing.

Let's see how this translates into code.

```
        def efficientWaitingFunction
            mutex.acquire()
            while predicate == false
              condVar.wait()
            end
            // Do something useful
            mutex.release()
        end

        def changePredicate()
            mutex.acquire()
            set predicate = true
            condVar.signal()
            mutex.release()
        end
```

Let's dry run the above code. Say **thread A** executes `efficientWaitingFunction()` first and finds the loop predicate is false and enters the loop. Next, **thread A** executes the statement `condVar.wait()` and is placed in a wait queue. At the same time **thread A** gives up the mutex. Now **thread B** comes along and executes `changePredicate()` method. Since the mutex was given up by **thread A**, **thread B** is able to acquire it and set the predicate to true. Next, it signals the condition variable `condVar.signal()`. This step places **thread A** into the ready

queue but **thread A** doesn't start executing until **thread B** has released the mutex.

Note that the order of signaling the condition variable and releasing the mutex can be interchanged, but generally, the preference is to signal first and then release the mutex. However, the ordering might have ramifications on thread scheduling, depending on the threading implementation.

Why the *while* Loop

The wary reader would have noticed us using a while loop to test for the predicate. After all, the pseudocode could have been written as follows:

```
    def efficientWaitingFunction()
        mutex.acquire()
        if predicate == false
          condVar.wait()
        end
        // Do something useful
        mutex.release()
    end
```

If the snippet is re-written in the above manner using an `if` clause instead of a `while` then, we need a guarantee that once the variable `condVar` is signaled, the predicate can't be changed by any other thread and that the signaled thread becomes the owner of the monitor. This may not be true. For one, a different thread could get scheduled and change the predicate back to false before the signaled thread gets a chance to execute, therefore the signaled thread must again check the predicate once it acquires the monitor. Secondly, the use of the loop is necessitated by design choices of monitors that we'll explore in the next section. Last but not the least, on POSIX systems, **spurious or fake wakeups** are possible (also discussed in later chapters) even though the condition variable has not been signaled and the predicate hasn't changed. **The idiomatic and correct usage of a monitor dictates that *the predicate always be tested for in a while loop*.**

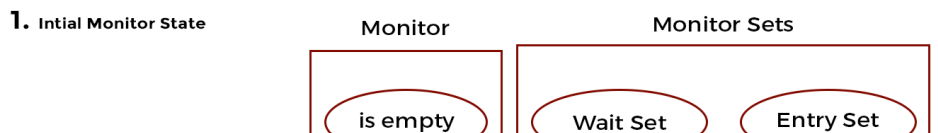
Monitor Explained

After the above discussion, we now realize that **a monitor is made up of a mutex and one or more condition variables**. A single monitor can have multiple condition variables but not vice versa. Theoretically, another way to think about a monitor is to consider it as an entity having two queues or sets where threads can be placed. One is the **entry set** and the other is the **wait set**. When a thread A ***enters*** a monitor, it is placed into the **entry set**. If no other thread ***owns*** the monitor, which is the equivalent of saying no thread is actively executing within the monitor section, then thread A will ***acquire*** the monitor and is said to own it too. Thread A will continue to execute within the monitor section till it ***exits*** the monitor or calls `wait()` on an associated condition variable and be placed into the wait set. While thread A **owns** the monitor, no other thread will be able to execute any of the critical sections protected by the monitor. New threads requesting ownership of the monitor get placed into the **entry set**.

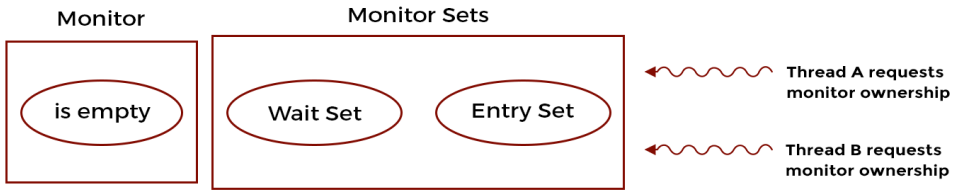
Continuing with our hypothetical example, say another thread B comes along and gets placed in the **entry set** while thread A sits in the **wait set**. Since no other thread owns the monitor, thread B successfully acquires the monitor and continues execution. If thread B exits the monitor section without calling `notify()` on the condition variable, then thread A will remain waiting in the **wait set**. Thread B can also invoke `wait()` and be placed in the **wait set** along with thread A. This then would require a third thread to come along and call `notify()` on the condition variable on which both threads A and B are waiting. Note that only a single thread will be able to **own** the monitor at any given point and will have exclusive access to data structures or critical sections protected by the monitor.

You can think of a monitor as a ***mutex with a wait set***. Monitors allow threads to exercise **mutual exclusion** as well as **cooperation** by allowing them to wait and signal on conditions.

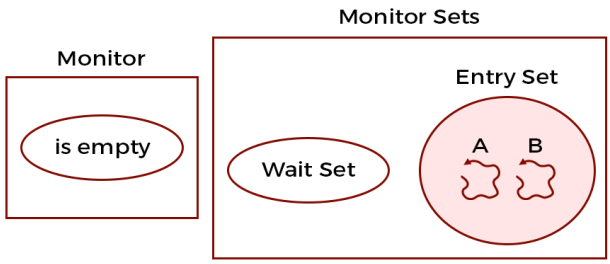
A pictorial representation of the working of a monitor is shown below:



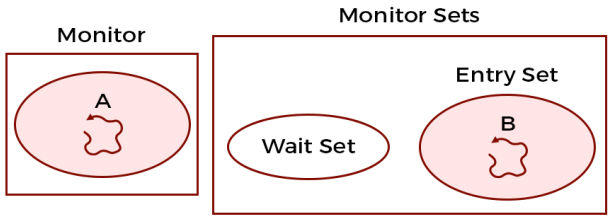
2. Two threads come along to enter the monitor



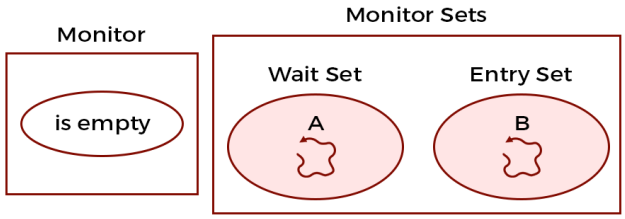
3. Thread A and B get placed in the entry set



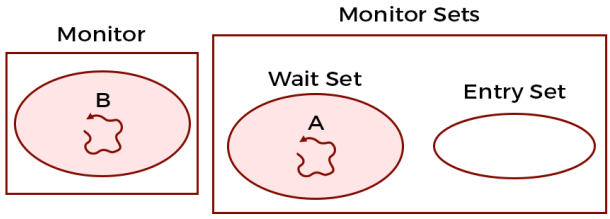
4. Thread A enters the monitor and starts execution



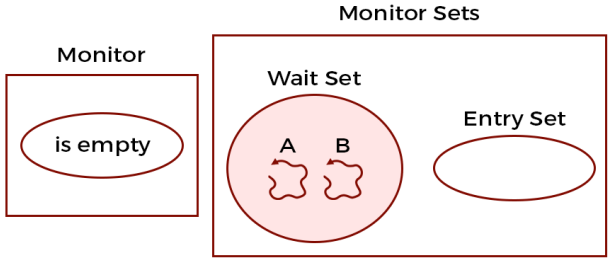
5. Thread A execution Wait() and gets placed in wait set



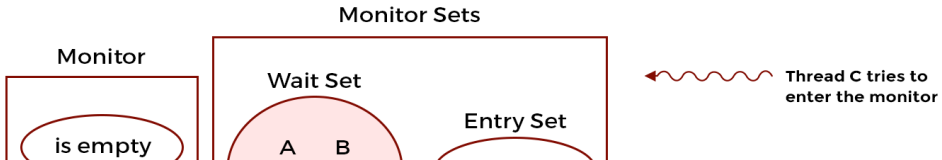
6. Thread B now able to enter the monitor



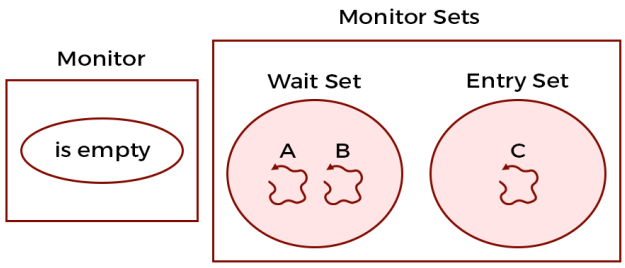
7. Thread B also invokes wait() and gets placed in the wait set



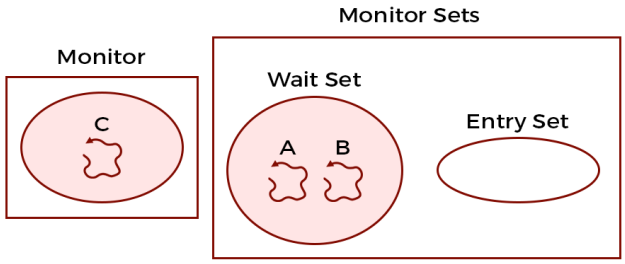
8. Thread C comes along to enter the monitor



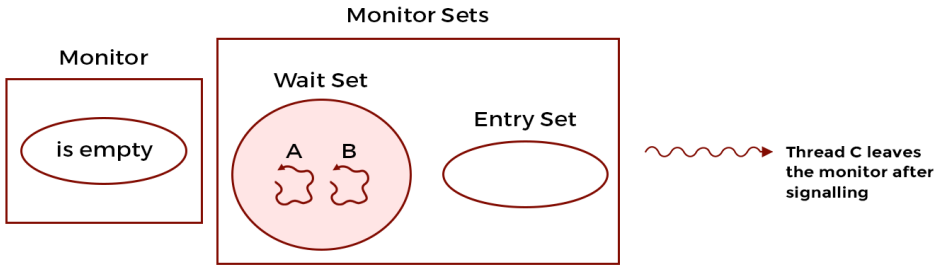
9. Thread C is placed in the entry set



10. Thread C enters the monitor



11. Thread C exits monitor after signalling



12. Thread A resumes ownership of the monitor