4.2 Wait and Notify

Just as each object has a lock that can be obtained and released, each object also provides a mechanism that allows it to be a waiting area. And just like the lock mechanism, the main reason for this mechanism is to aid communication between threads.[1] The idea behind the mechanism is actually simple: one thread needs a certain condition to exist and assumes that another thread will create that condition. When this other thread creates the condition, it notifies the first thread that has been waiting for the condition. This is accomplished with the following methods:

[1] Under Solaris or POSIX threads, these are often referred to as *condition variables* ; on Windows 95/NT, they are referred to as *event variables*.

void wait()

Waits for a condition to occur. This is a method of the Object class and must be called from within a synchronized method or block.

void notify()

Notifies a thread that is waiting for a condition that the condition has occurred. This is a method of the Object class and must be called from within a synchronized method or block.

wait(), notify(), and the Object Class

Interestingly enough, just like the synchronized method, the *wait and notify* mechanism is available from every object in the Java system. However, this mechanism is accomplished by method invocations, whereas the synchronized mechanism is done by adding a keyword.

The `wait()/notify()` mechanism works because these are methods of the Object class. Since every object in the Java system inherits directly or indirectly from the Object class, it is also an Object and hence has support for this mechanism.

What is the purpose of the wait and notify mechanism, and how does it work? The wait and notify mechanism is also a synchronization mechanism; however, it is more of a communication mechanism: it allows one thread to communicate to another thread that a particular condition has occurred. The wait and notify mechanism does not specify what the specific condition is.

Can wait and notify be used to replace the synchronized method? Actually, the answer is no. Wait and notify does not solve the race condition problem that the synchronized mechanism solves. As a matter of fact, wait and notify must be used in conjunction with the synchronized lock to prevent a race condition in the wait and notify mechanism itself.

Wait and Notify and Synchronization

As noted, the wait and notify mechanism has a race condition that needs to be solved with the synchronization lock. Unfortunately, it is not possible to solve the race condition without integrating the lock into the wait and notify mechanism. This is why it is mandatory for the `wait()` and `notify()` methods to hold the locks for the objects for which they are waiting or notifying.

The `wait()` method releases the lock prior to waiting, and reacquires the lock prior to returning from the `wait()` method. This is done so that no race condition exists. If you recall, there is no concept of releasing and reacquiring a lock in the Java API. The `wait()` method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the `wait()` method purely in Java: it is a native method.

This integration of the wait and notify and the synchronized method is actually standard. In other systems, such as Solaris or POSIX threads, condition variables also require that a mutex lock be held for the mechanism to work.

What happens when notify() is called and there is no thread waiting? This is a valid situation. Even with our BusyFlag class, it is perfectly valid to free the `busyflag` when there is no other thread waiting to get the `busyflag`. Since the wait and notify mechanism does not know the condition about which it is sending notification, it assumes that a notification for which there is no thread waiting is a notification that goes unheard. In other words, if `notify()` is called without another thread waiting, then `notify()` simply returns.

What are the details of the race condition that exists in wait and notify? In general, a thread that uses the `wait()` method confirms that a condition does not exist (typically by checking a variable) and then calls the `wait()` method. When another thread sets the condition (typically by setting that same variable), it then calls the `notify()` method. A race condition occurs when:

1. The first thread tests the condition and confirms that it must wait.
2. The second thread sets the condition.
3. The second thread calls the `notify()` method; this goes unheard, since the first thread is not yet waiting.
4. The first thread calls the `wait()` method.

How does this potential race condition get resolved? This race condition is resolved by the synchronization lock discussed earlier. In order to call `wait()` or `notify()`, we must have obtained the lock for the object on which we're calling the `wait()` or `notify()` method. This is mandatory: the methods will not work properly and will generate an exception condition if the lock is not held. Furthermore, the `wait()` method also releases the lock prior to waiting and reacquires the lock prior to returning from the `wait()` method. The developer must use this lock to ensure that checking the condition and setting the condition is atomic, which typically means that the condition is held in an instance variable within the locked object.

Is there a race condition during the period that the wait() method releases and reacquires the lock? The `wait()` method is tightly integrated with the lock mechanism. The object lock is not actually freed until the waiting thread is already in a state in which it can receive notifications. This would have been difficult, if not impossible, to accomplish if we had needed to implement the `wait()` and `notify()` methods ourselves. For our purposes, this is an implementation detail. It works, and works correctly. The system prevents any race conditions from occurring in this mechanism.

4.3 wait(), notify(), and notifyAll()

What happens when there is more than one thread waiting for the notification? Which thread actually gets the notification when notify() is called? The answer is that it depends: the Java specification doesn't define which thread gets notified. Which thread actually receives the notification varies based on several factors, including the implementation of the Java virtual machine and scheduling and timing issues during the execution of the program. There is no way to determine, even on a single platform, which of multiple threads receives the notification.

There is another method of the Object class that assists us when multiple threads are waiting for a condition:

void notifyAll()

Notifies all the threads waiting on the object that the condition has occurred. This is a method of the Object class and must be called from within a synchronized method or block.

The Object class also provides the notifyAll() method, which helps us in those cases where the program cannot be designed to allow any arbitrary thread to receive the notification. This method is similar to the notify() method, except that all of the threads that are waiting on the object will be notified instead of a single arbitrary thread. Just like the notify() method, the notifyAll() method does not let us decide which threads get notification: they all get notified. By having all the threads receive notification, it is now possible for us to work out a mechanism for the threads to choose among themselves which thread should continue and which thread(s) should call the wait() method again.

Does notifyAll() Really Wake Up All the Threads?

Yes and no. All the waiting threads will wake up, but they still have to reacquire the object lock. So the threads will not run in parallel: they must each wait for the object lock to be freed. Thus only one thread can run at a time, and only after the thread that called the notifyAll() method releases its lock.

Why would you want to wake up all of the threads? There are a few possible reasons, one of which is if there is more than one condition to wait for. Since we cannot control which thread gets the notification, it is entirely possible that a notification wakes up a thread that is waiting for an entirely different condition. By waking up all the waiting threads, we can design the program so that the threads decide among themselves which should execute next.

Another reason is the case where the notification can satisfy multiple waiting threads. Let's examine a case where we need such control:

```
public class ResourceThrottle {

    private int resourcecount = 0;
    private int resourcemax = 1;

    public ResourceThrottle (int max) {
        resourcecount = 0;
        resourcemax = max;
    }

    public synchronized void getResource (int numberof) {
        while (true) {
            if ((resourcecount + numberof) <= resourcemax) {
                resourcecount += numberof;
                break;
            }
            try {
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized void freeResource (int numberof) {
        resourcecount -= numberof;
        notifyAll();
    }
}
```

We are defining a new class called the ResourceThrottle class. This class provides two methods, getResource() and freeResource(). Both of these methods take a single parameter that specifies how many resources to grab or release. The maximum number of resources available is defined by the constructor of the ResourceThrottle class. This class is similar to our BusyFlag class, in that our getResource() method would have to wait if the number of requested resources is not available. The freeResource() method also has to call the notify() method so that the waiting threads can get notification when more resources are available.

The difference in this case is that we are calling the `notifyAll()` method instead of the `notify()` method. There are two reasons for this:

- It is entirely possible for the system to wake up a thread that needs more resources than are available, even with the resources that have just been freed. If we had used the `notify()` method, another thread that could be satisfied with the current amount of resources would not get the chance to grab those resources because the system picked the wrong thread to wake up.

- It is possible to satisfy more than one thread with the number of resources we have just freed. As an example, if we free ten resources, we can then let four other threads grab three, four, one, and two resources, respectively. There is not a one-to-one ratio between the number of threads freeing resources and the number of threads grabbing resources.

By notifying all the threads, we solve these two problems with little work. However, all we have accomplished is to simulate a targeted notification scheme. We are not really controlling which threads wake up; instead, we are controlling which thread takes control after they all get notification. This can be very inefficient if there are many threads waiting to get notification, because many wake up only to see that the condition is still unsatisfied, and they must wait again.

If we really need to control which thread gets the notification, we could also implement an array of objects whose sole purpose is to act as a waiting point for threads and who are targets of notification of conditions. This means that each thread waits on a different object in the array. By having the thread that calls the `notify()` method decide which thread should receive notification, we remove the overhead of many threads waking up only to go back to a wait state moments later. The disadvantage of using an array of objects is, of course, that we will lock on different objects. This acquisition of many locks could lead to confusion or, even worse, deadlock. It is also more complicated to accomplish; we may even have to write a new class just to help with notification targeting:

```
public class TargetNotify {

    private Object Targets[] = null;

    public TargetNotify (int numberOfTargets) {
        Targets = new Object[numberOfTargets];

        for (int i = 0; i < numberOfTargets; i++) {
            Targets[i] = new Object();
        }
    }

    public void wait (int targetNumber) {
        synchronized (Targets[targetNumber]) {
            try {
                Targets[targetNumber].wait();
            } catch (Exception e) {}
        }
    }

    public void notify (int targetNumber) {
        synchronized (Targets[targetNumber]) {
            Targets[targetNumber].notify();
        }
    }
}
```

The concept is simple: in our TargetNotify class, we are using an array of objects for the sole purpose of using the wait and notify mechanism. Instead of having all the threads wait on the `this` object, we choose an object to wait on. (This is potentially confusing: we are not overriding the `wait()` method of the Object class here since we've provided a unique signature.) Later, when we decide which threads should wake up, we can target the notification since the threads are waiting on different objects.

Whether the efficiency of a targeted notification scheme outweighs the extra complexity is the decision of the program designer. In other words, both techniques have their drawbacks, and we leave it up to the implementors to decide which mechanism is best.