Summary on Monitor Implementation techniques

Note: in this document we use process and thread interchangeably.

Monitor is neither a process (thread) nor an active entity. It is just an abstract data type (or class) whose code can be executed by only one process (or thread) at a time. A monitor object is an object intended to be used to coordinate multiple threads. Methods in the monitor can be either invoked static or non-static ways (creates monitor objects).

The methods defined in the monitor are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

Need of Condition Variables in the Monitor

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition *P* holds true. A busy waiting (or spin lock) is not an efficient solution for implementing wait() operation associated with a condition variable. Because by the definition of the monitor, only one thread is allowed in the monitor, the thread with busy waiting will prevent any other thread from entering the monitor to make the condition true.

A condition variable defined in the monitor is associated with a queue on which a thread may wait for some condition to become true. Thus each condition variable *c* is associated with an assertion P_c. While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable *c* to indicate that assertion P_c is true in the current state.

Thus there are two main operations required on condition variables:

- **wait** c is called by a thread that needs to wait until the assertion P_c is true before proceeding. While the thread is waiting, it does not occupy the monitor.
- **signal** c (sometimes termed as **notify** c) is called by a thread to indicate that the assertion P_c is true.

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. To ensure only one thread is active in the monitor at each time, a choice must be made.

Two possibilities as the text book explains:

- *Blocking condition variables* (or *Signal and Wait*) give priority to a signaled thread.
- *Non-blocking condition variables* (or *Signal and Continue*) give priority to the signaling thread.

1. Blocking condition variables (Signal and Wait)

The original proposals by two famous computer scientists C. A. R. Hoare and Per Brinch Hansen were for *blocking condition variables*. With a blocking condition variable, the signaling thread must wait until the

signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable.

Let us first see the monitor implementation (with blocking condition variables) by using semaphores explained in the textbook.

- Only one thread can be active in a monitor at any instance.
    - When a thread calls a monitor procedure (method), the procedure will first check to see if any other thread is currently active within the monitor.
    - If so, the calling thread will be suspended until the other thread has left the monitor.
    - If no other thread is using the monitor, the calling thread may enter.
- So, by turning all the critical sections (of a problem) into monitor procedures, no two threads (or processes) will ever execute their critical sections at the same time.
- For each condition variable x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0.
- Here, we implement the signal and wait in such a way that a signaling thread must wait until the resumed thread either leaves or wait. To do that we need an additional semaphore, next, initialized to 0, on which the signaling threads may suspend themselves.

Our textbook does not explain the blocking condition variable concept clearly. Also the blocking condition variables (signal wait approach) can be implemented in two different ways (signal and urgent wait; signal and wait) with/without an additional queue in the monitor.

To better understand a blocking implementation technique (signal and urgent wait), let us see the Figure 1.

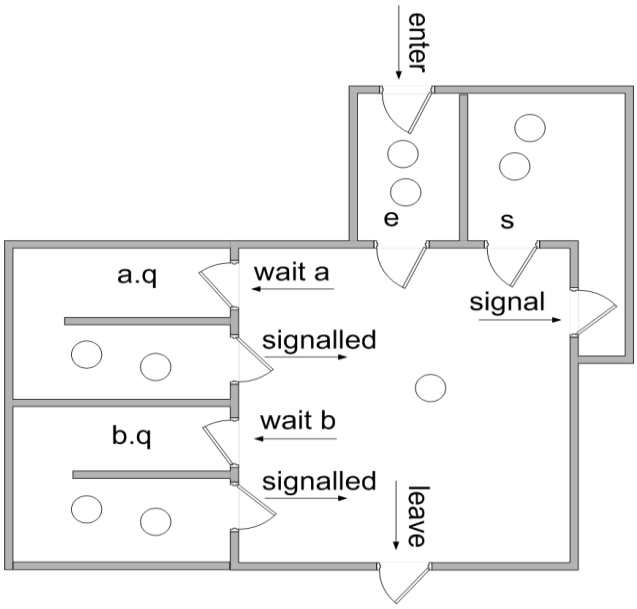

Figure 1: Monitor implementation based on blocking condition variables (signal and urgent wait)
There are two condition variables a b in this monitor. We assume there are two queues of threads associated with each monitor object

- e is the entrance queue
- s is a queue of threads that have signaled.

In addition we assume that for each condition variable *c*, there is a queue

- *c*.q, which is a queue for threads waiting on condition variable *c*

All queues are typically guaranteed to be fair.
Each monitor operation in this implementation technique (signal and urgent wait) is explained below.

```
enter the monitor operation:
   enter the method
   if the monitor is locked
      add this thread to e
      block this thread
   else
      lock the monitor

leave the monitor operation:
   return from the monitor (i.e., from the method invoked)
   schedule //see schedule operation below

 wait c operation (operation on condition variable c):
   add this thread to c.q
   block this thread
   schedule

signal c operation :
   if there is a thread waiting on c.q
      select and remove one such thread t from c.q
      (t is called "the signaled thread")
      add this thread (that have signaled) to s
      block this thread
      resume t
      (so t will occupy the monitor next)

  Schedule operation :
   if there is a thread on s
     select and remove one thread from s and restart it
     (this thread will occupy the monitor next)
   else if there is a thread on e
     select and remove one thread from e and restart it
     (this thread will occupy the monitor next)
   else
     unlock the monitor  (the monitor will become unoccupied)
```

An alternative technique is "signal and wait". There is no s queue in this technique and signaler waits on the e queue instead. In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread on waiting on the condition variable, the signaling

thread hands occupancy over to the signaled thread , so that no other thread can gain occupancy in between.

2. Non-blocking condition variables (signal and continue)

With *non-blocking condition variables* (*signal and continue* condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the e queue. There is no need for the s queue.

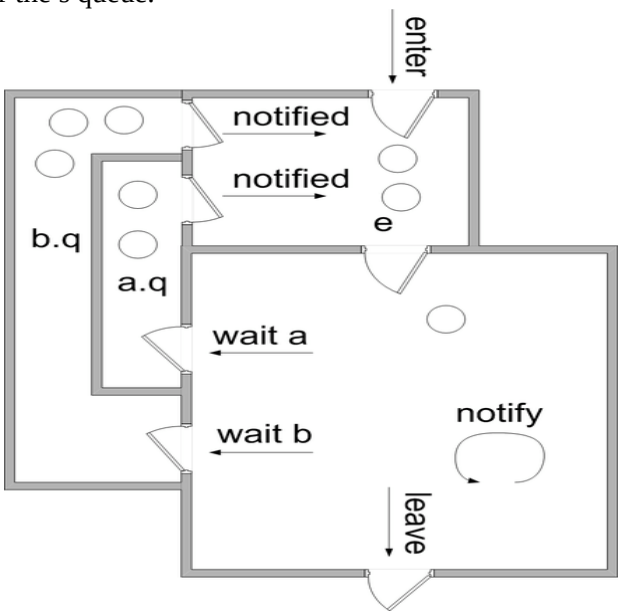

Figure 2: Non-blocking condition variables (signal and continue) implementation

With non-blocking condition variables, the **signal** operation is often called **notify.** It is also common to provide a **notifyAll** operation that moves all threads waiting on a condition variable to the e queue. Each monitor operation in this implementation technique (signal and continue) is explained below.

```
enter the monitor operation:
   enter the method
   if the monitor is locked
     add this thread to e
     block this thread
   else
     lock the monitor

leave the monitor operation:
   return from the method
   schedule
```

wait c operation:

```
    add this thread to c.q
    block this thread
    schedule

 notify c operation :
    if there is a thread waiting on c.q
       select and remove one thread t from c.q
       (t is called "the notified thread")
       move t to e

notify all c  operation:
    move all threads waiting on c.q to e

schedule operation :
    if there is a thread on e
      select and remove one thread from e and restart it
    else
      unlock the monitor
```

As a variation on this scheme, the notified thread may be moved to a queue called w, which has priority over e.

3. Java Monitor Implementation Technique

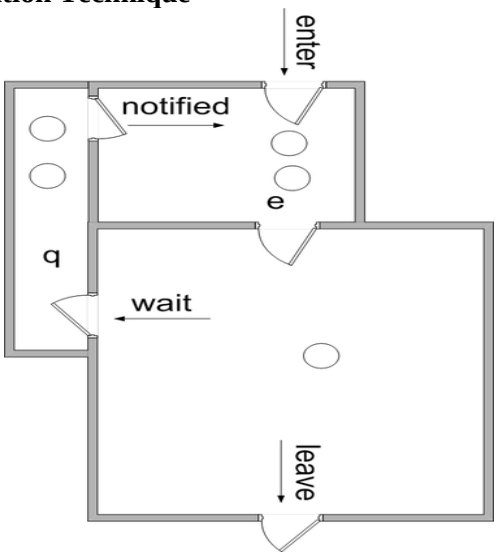

Figure 3: Java monitor implementation

In the Java language, each object may be used as a monitor. (However, methods that require mutual exclusion must be explicitly marked as synchronized.) Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue, in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notify all** operations apply to this queue. This approach has also been adopted in other languages such as C#. In some case notify signaling may be done implicitly (i.e., returning thread will trigger implicit signaling).