# Thread Concurrency Basics:

# Difference between lock and monitor – Java Concurrency

Short answer, **locks provide necessary support for implementing monitors**. Long answer read below.

Monitor : Its for certain set of code [should](http://www.permies.com/t/36936/tnk/) be executed by only one thread at a time.   
Which contains an shared object(resource). Monitors are set only on objects.   
  
Locks : Locks is for each object(resource). whenever one thread gets resource it occupies locks on that object.   
& at exit time tht thread just release lock of that object.

## Locks

**A lock is kind of data which is logically part of an object’s header on the heap memory.** Each object in a JVM has this lock (or mutex) that any program can use to coordinate multi-threaded access to the object. If any thread want to access instance variables of that object; then thread must “own” the object’s lock (set some flag in lock memory area). All other threads that attempt to access the object’s variables have to wait until the owning thread releases the object’s lock (unset the flag).

Once a thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads. If a thread requests a lock three times, for example, that thread will continue to own the lock until it has “released” it three times.

Please note that lock is acquired by a thread, when it explicitly ask for it. In Java, this is done with the synchronized keyword, or with wait and notify.

## Monitors

**Monitor is a synchronization construct that allows threads to have both mutual exclusion (using locks) and cooperation** i.e. the ability to make threads wait for certain condition to be true (using **wait-set**).

In other words, along with data that implements a lock, every Java object is logically associated with data that implements a wait-set. Whereas locks help threads to work independently on shared data without interfering with one another, wait-sets help threads to cooperate with one another to work together towards a common goal e.g. all waiting threads will be moved to this wait-set and all will be notified once lock is released. **This wait-set helps in building monitors with additional help of lock (mutex).**

#### Mutual exclusion

Putting in very simple words, a monitor is like a building that contains one special room (object instance) that can be occupied by only one thread at a time. The room usually contains some data which needs to be protected from concurrent access. From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room. Entering the monitor building is called “entering the monitor.” Entering the special room inside the building is called “acquiring the monitor.” Occupying the room is called “owning the monitor,” and leaving the room is called “releasing the monitor.” Leaving the entire building is called “exiting the monitor.”

When a thread arrives to access protected data (enter the special room), it is first put in queue in building reception (entry-set). If no other thread is waiting (own the monitor), the thread acquires the lock and continues executing the protected code. When the thread finishes execution, it release the lock and exits the building (exiting the monitor).

If when a thread arrives and another thread is already owning the monitor, it must wait in reception queue (entry-set). When the current owner exits the monitor, the newly arrived thread must compete with any other threads also waiting in the entry-set. Only one thread will win the competition and own the lock.

**There is no role of wait-set feature.**

#### Cooperation

In general, mutual exclusion is important only when multiple threads are sharing data or some other resource. If two threads are not working with any common data or resource, they usually can’t interfere with each other and needn’t execute in a mutually exclusive way. Whereas mutual exclusion helps keep threads from interfering with one another while sharing data, cooperation helps threads to work together towards some common goal.

**Cooperation is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state e.g. producer/consumer problem** where read thread needs the buffer to be in a “not empty” state before it can read any data out of the buffer. If the read thread discovers that the buffer is empty, it must wait. The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading. It is also sometimes called a “**Wait and Notify**” OR “**Signal and Continue**” monitor because it retains ownership of the monitor and continues executing the monitor region (the continue) if needed. At some later time, the notifying thread releases the monitor and a waiting thread is resurrected to own the lock.

**This cooperation requires both i.e. entry-set and wait-set.** Below given diagram will help you in understand this cooperation.



Above figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor’s owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set.

Important Notes :

1. Wait()/notify()/notifyAll() will only be called if the current thread is lock on this object, otherwise it will throw IllegalMonitorStateException.

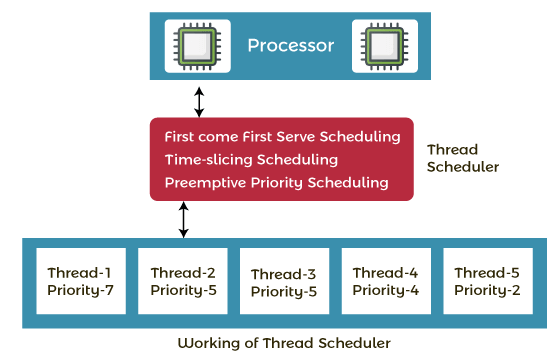
As the thread calls the synchronized method it is placed in the Entry Set.

* If no thread currently owns the monitor and no other threads are waiting in the entry set, the thread becomes the owner of the monitor and starts executing the code. This is called the active thread.
* Else, if there is another thread that is owning the monitor, the current thread is placed in the Entry set and will have to wait for its turn along with other already existing threads in Entry Set, if any.

While the current thread is waiting in the Entry Set,

* The active thread will release the monitor when it is done with executing the code in the synchronized block. (other way of releasing is through wait() method that we will ignore for our discussion)
* Subsequently, as the monitor is free now, the threads in the entry set will compete to acquire the monitor and one of them will get a chance.

**NOTE :** As indicated above, for the above discussion, we have assumed that there are no threads in the Wait Set for simplicity to keep discussion to the point. Wait Set comes into the picture with wait() and notify() calls.



There is a component in Java that basically decides which thread should execute or get a resource in the operating system.

Scheduling of [threads](https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/) involves two boundary scheduling.

1. Scheduling of user-level threads (ULT) to kernel-level threads (KLT) via lightweight process (LWP) by the application developer.
2. Scheduling of kernel-level threads by the system scheduler to perform different unique OS functions.

## **Contention Scope**

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library.

Depending upon the extent of contention it is classified as-

* **Process Contention Scope (PCS) :**  
  The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
* **System Contention Scope (SCS) :**  
  The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.

In LINUX and UNIX operating systems, the POSIX Pthread library provides a function Pthread\_attr\_setscope to define the type of contention scope for a thread during its creation.

Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

The thread scheduler selects a thread for execution from runnable state. But there is no guarantee that which thread from runnable pool will be selected next to run by the thread scheduler.

Java runtime system mainly uses one of the following two strategies:

1. **Preemptive scheduling**
2. **Time-sliced scheduling**

## Preemptive Scheduling

This scheduling is based on priority. Therefore, this scheduling is known as priority-based scheduling. In the priority-based scheduling algorithm, Thread scheduler uses the priority to decide which thread should be run.

If a thread with a higher priority exists in Runnable state (ready state), it will be scheduled to run immediately.

In case more than two threads have the same priority then CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

## Time-Sliced Scheduling

The process of allocating time to threads is known as **time slicing in Java**. Time-slicing is based on non-priority scheduling. Under this scheduling, every running thread is executed for a fixed time period.

A currently running thread goes to the Runnable state when its time slice is elapsed and another thread gets time slots by CPU for execution.

With time-slicing, threads having lower priorities or higher priorities gets the same time slots for execution.