# How JVM works internally while one Thread work on A synchronized method and other waits for it till lock release?

Threads and shared data

One of the strengths of the Java programming language is its support for multithreading at the language level. Much of this support centers on coordinating access to data shared among multiple threads.

The JVM organizes the data of a running Java application into several runtime data areas: one or more Java stacks, a heap, and a method area. For a backgrounder on these memory areas, see the first Under the Hood article: “The lean, mean virtual machine.”

Inside the Java virtual machine, each thread is awarded a Java stack, which contains data no other thread can access, including the local variables, parameters, and return values of each method the thread has invoked. The data on the stack is limited to primitive types and object references. In the JVM, it is not possible to place the image of an actual object on the stack. All objects reside on the heap.

There is only one heap inside the JVM, and all threads share it. The heap contains nothing but objects. There is no way to place a solitary primitive type or object reference on the heap — these things must be part of an object. Arrays reside on the heap, including arrays of primitive types, but in Java, arrays are objects too.

Besides the Java stack and the heap, the other place data may reside in the JVM is the method area, which contains all the class (or static) variables used by the program. The method area is similar to the stack in that it contains only primitive types and object references. Unlike the stack, however, the class variables in the method area are shared by all threads.

Object and class locks

As described above, two memory areas in the Java virtual machine contain data shared by all threads. These are:

The heap, which contains all objects  
The method area, which contains all class variables

If multiple threads need to use the same objects or class variables concurrently, their access to the data must be properly managed. Otherwise, the program will have unpredictable behavior.

To coordinate shared data access among multiple threads, the Java virtual machine associates a lock with each object and class. A lock is like a privilege that only one thread can “possess” at any one time. If a thread wants to lock a particular object or class, it asks the JVM. At some point after the thread asks the JVM for a lock — maybe very soon, maybe later, possibly never — the JVM gives the lock to the thread. When the thread no longer needs the lock, it returns it to the JVM. If another thread has requested the same lock, the JVM passes the lock to that thread.

Class locks are actually implemented as object locks. When the JVM loads a class file, it creates an instance of class java.lang.Class. When you lock a class, you are actually locking that class’s Class object.

Threads need not obtain a lock to access instance or class variables. If a thread does obtain a lock, however, no other thread can access the locked data until the thread that owns the lock releases it.

Monitors

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian in that it watches over a sequence of code, making sure only one thread at a time executes the code.

Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code that is under the watchful eye of a monitor, the thread must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code.

When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.