**Deadlocks vs Race Conditions**

**Race Conditions**

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

**Deadlocks**

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.

**Preventing Race Conditions**

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a synchronized block of Java code. Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger.

**Critical Section Throughput**

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

Here is a very simplified Java code example to show what I mean:

Public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

public void add(int val1, int val2){

synchronized(this){

this.sum1 += val1;

this.sum2 += val2;

}

}

}

Notice how the add() method adds values to two different sum member variables. To prevent race conditions the summing is executed inside a Java synchronized block. With this implementation only a single thread can ever execute the summing at the same time.

However, since the two sum variables are independent of each other, you could split their summing up into two separate synchronized blocks, like this:

Public class TwoSums {

Private int sum1 = 0;

Private int sum2 = 0;

Public void add (int val1, int val2){

Synchronized (this){

this.sum1 += val1;

}

Synchronized (this){

this.sum2 += val2;

}

}

}

Now two threads can execute the add() method at the same time. One thread inside the first synchronized block and another thread inside the second synchronized block. This way threads will have to wait less for each other to execute the add() method.

This example is very simple, of course. In a real life shared resource the breaking down of critical sections may be a whole lot more complicated, and require more analysis of execution order possibilities.