Thread Synchronisation

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.

Critical section :
Critical section refers to the parts of the program where the shared resource is accessed. Concurrent accesses to shared resource can lead to race condition.

Race condition :
A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

threading module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System.

A semaphore is a synchronization object that controls access by multiple processes/ threads to a common resource in a parallel programming environment.
It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change.
Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values.
Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Lock class provides following methods:
• acquire([blocking]) : To acquire a lock. A lock can be blocking or non-blocking.
• When invoked with the blocking argument set to True (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return True.
• When invoked with the blocking argument set to False, thread execution is not blocked. If lock is unlocked, then set it to locked and return True else return False immediately.
• release() : To release a lock.
• When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
• If lock is already unlocked, a ThreadError is raised.

Consider below application which demonstrates use of Lock

```
import threading

amount = 0

def Counter(fun,lock):
    fun(lock)

def Credit(lock):
    value = input("Enter amount to Creadit")
    lock.acquire()
    global amount
    amount += value
    print("Balance is",amount)
    lock.release()

def Debit(lock):
    value = input("Enter amount to widraw")
    lock.acquire()
    global amount
    if amount < value:
        print("Unable to withdraw")
    else:
        amount += value
        print("Amount withdrawn ",value)
        print("Balance is",amount)
    lock.release()

def main():
    lock = threading.Lock()
    t1 = threading.Thread(target = Counter, args = (Credit,lock,))
    t2 = threading.Thread(target = Counter, args = (Debit,lock,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

if __name__ == "__main__":
        main()
```

In above application at a time only one thread gets executed. If any of the above thread gets scheduled it acquires lock and at that time other thread has to wait.

Output of above application

```
Today — -bash — 51×20
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
ThreadSync.py
Enter amount to Creadit3000
Enter amount to widraw('Balance is', 3000)
200
('Amount withdrawn ', 200)
('Balance is', 3200)
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```