Assignment OS - 7
Name: Veeransh Shah
Reg Id: 221070063

Aim:
Thread synchronisation using lock mechanism

Theory:

Thread synchronization using lock mechanisms is a crucial aspect of concurrent programming, aimed at preventing race conditions and maintaining data integrity in multi-threaded environments. The fundamental idea behind locks is to ensure that only one thread can access a critical section of code or a shared resource at any given time. When a thread needs to access a critical section, it first acquires the lock associated with that section. If the lock is available, the thread obtains it and proceeds to execute the critical section. If the lock is already held by another thread, the current thread is blocked until the lock becomes available. Once the critical section is executed, the thread releases the lock, allowing other threads to acquire it and access the critical section. This mechanism effectively serializes access to shared resources, preventing concurrent modification and potential inconsistencies.

Locks provide a simple and effective means of ensuring thread safety and synchronization in concurrent programs. However, improper use of locks can lead to issues such as deadlocks and performance bottlenecks. Deadlocks occur when multiple threads are waiting for locks held by each other, resulting in a state where none of the threads can make progress. Performance can also be impacted if locks are held for prolonged periods, causing contention among threads. Therefore, it's essential to carefully design and manage the use of locks to balance synchronization requirements with performance considerations. Additionally, modern programming languages and frameworks often provide higher-level synchronization constructs, such as mutexes and semaphores, built on top of locks, which offer more flexibility and safety in managing concurrency.

Code:

```
import threading


shared_resource = 0


lock = threading.Lock()

def modify_shared_resource():
   global shared_resource

   lock.acquire()
   try:

       shared_resource += 1
       print(f"Shared resource value after modification:
{shared_resource}")
   finally:

       lock.release()


threads = []
for _ in range(5):
   t = threading.Thread(target=modify_shared_resource)
   threads.append(t)


for t in threads:
   t.start()


for t in threads:
   t.join()

print("All threads completed.")
```

```
   # We define a shared resource (shared_resource) that multiple
threads will access.
   # We create a lock using threading.Lock().
   # The modify_shared_resource() function represents a critical
section where the shared resource is being modified. Before accessing
the shared resource, the thread acquires the lock using lock.acquire(),
ensuring exclusive access. After modifying the shared resource, the
lock is released using lock.release().
   # Multiple threads are created to concurrently execute
modify_shared_resource().
   # Each thread executes the critical section atomically, ensuring
that only one thread can access the shared resource at a time.
   # The join() method is called on each thread to wait for all threads
to complete before printing "All threads completed."
```

Output:

```
/usr/bin/python3 /home/veeransh/Desktop/Lab_work/OS/files/assn7.py
veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$ /usr/bin/python3 /home/veeransh/Desktop/Lab_work/OS/files/assn7.py
Shared resource value after modification: 1
Shared resource value after modification: 2
Shared resource value after modification: 3
Shared resource value after modification: 4
Shared resource value after modification: 5
All threads completed.
veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$
```

Conclusion:

In summary, thread synchronization using lock mechanisms ensures safe access to shared resources in concurrent programming. Proper lock management prevents race conditions but requires careful consideration to avoid deadlocks and performance issues. Leveraging higher-level constructs enhances safety and efficiency, making mastery of synchronization techniques essential for robust concurrent application development.