Low level Synchronization Tool

LOCK :-

→ Critical Section Problem could be more simply solved in a uniprocessor, if we could disallow interrupts to occur while a shared variable is being modified.

→ Must ensure that current sequence of instⁿ. would be allowed to execute in order without preemption.

→ But the above is not possible in multiprocessor environment. Disabling interrupts on multiprocessor is time consuming due to message passing to all Processors.

→ Hence many m/c there fore provide special H/w instⁿ. for CS Problem.

→ A LOCK is a object that provides the following 2 operations :-

(A) acquire ( ) → wait to enter CS

(B) release ( ) → Allow another to enter to CS.

```
acquire_lock ( )
    Critical Section
Release_lock ( )
    Reminder Section
```

Ex:

```
int withdraw(account, amount)
{
    acquire (lock);
    balance = get_balance(account);
    balance = balance - amount;
    Put_balance(account, balance);
    release (lock);
} Return balance ;
```

Lock implementation

```
Struct lock
  {
    int held = 0;
  }
void acquire (lock)
  {
    while (lock->held);
    lock->held = 1;
  }
void release (lock)
  {
    lock->held = 0;
  }
```

→ if inst. preempted after while then mutual exclusion violated. // To prevent it disable interrupts.
↳ one solution.
→ Busy waits
→ Context switch here.
→ Hence this sequence must be atomic. //

Solution — **Test & Set :-**

```
test_set (*lock)
  {
    b = *lock;
    *lock = 1;      } → Atomic
    return b;
  }
```

```
repeat
  while (Test_Set (&lock)) do
    no-op
  CS
  lock = false;
  RS
until false;
```