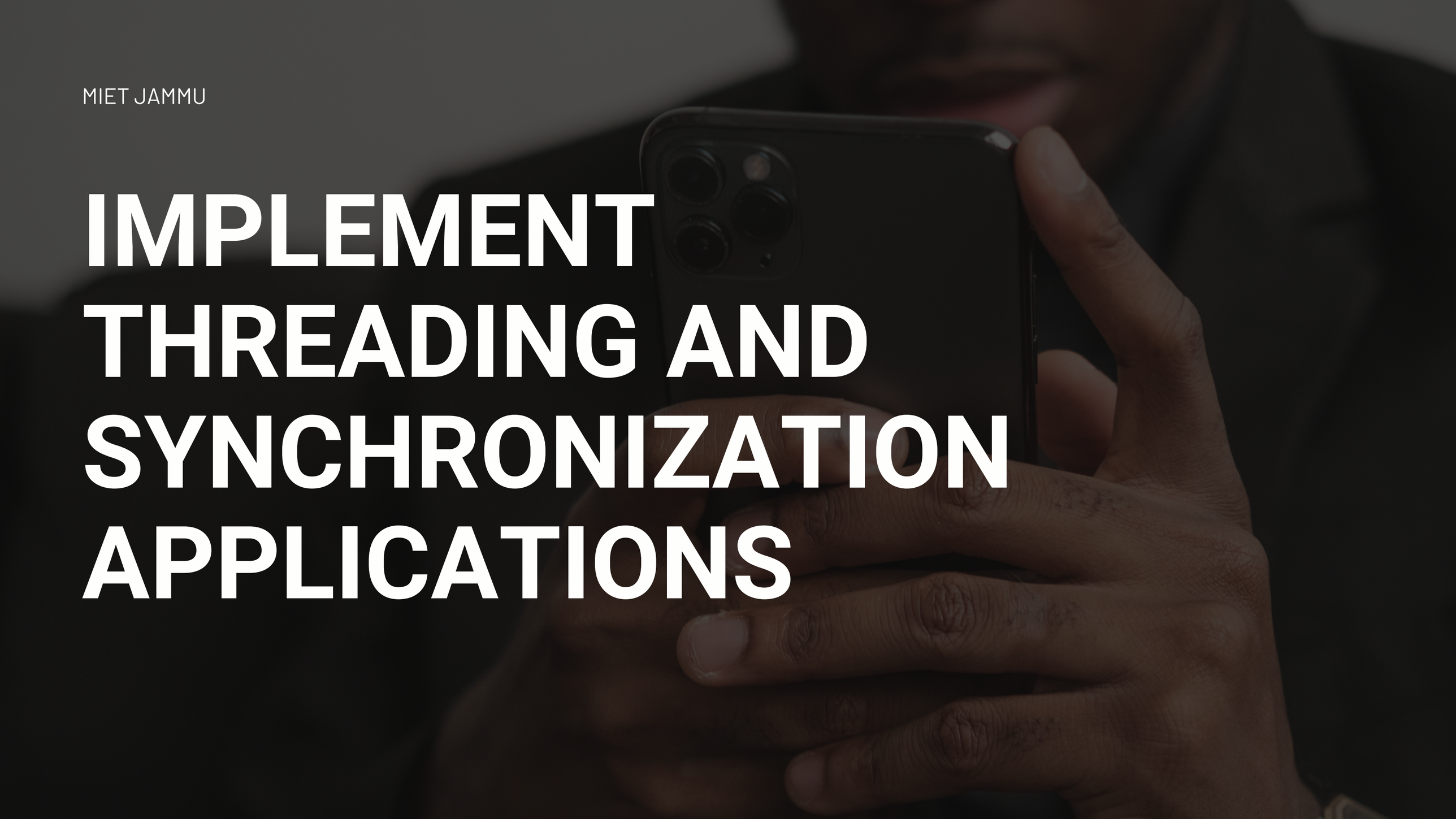
MIET JAMMU

IMPLEMENT THREADING AND SYNCHRONIZATION APPLICATIONS



WHAT IS THREADING?

Thread is a sequential flow of tasks within a process. Threads in OS can be of the same or different types. Threads are used to increase the performance of the applications.

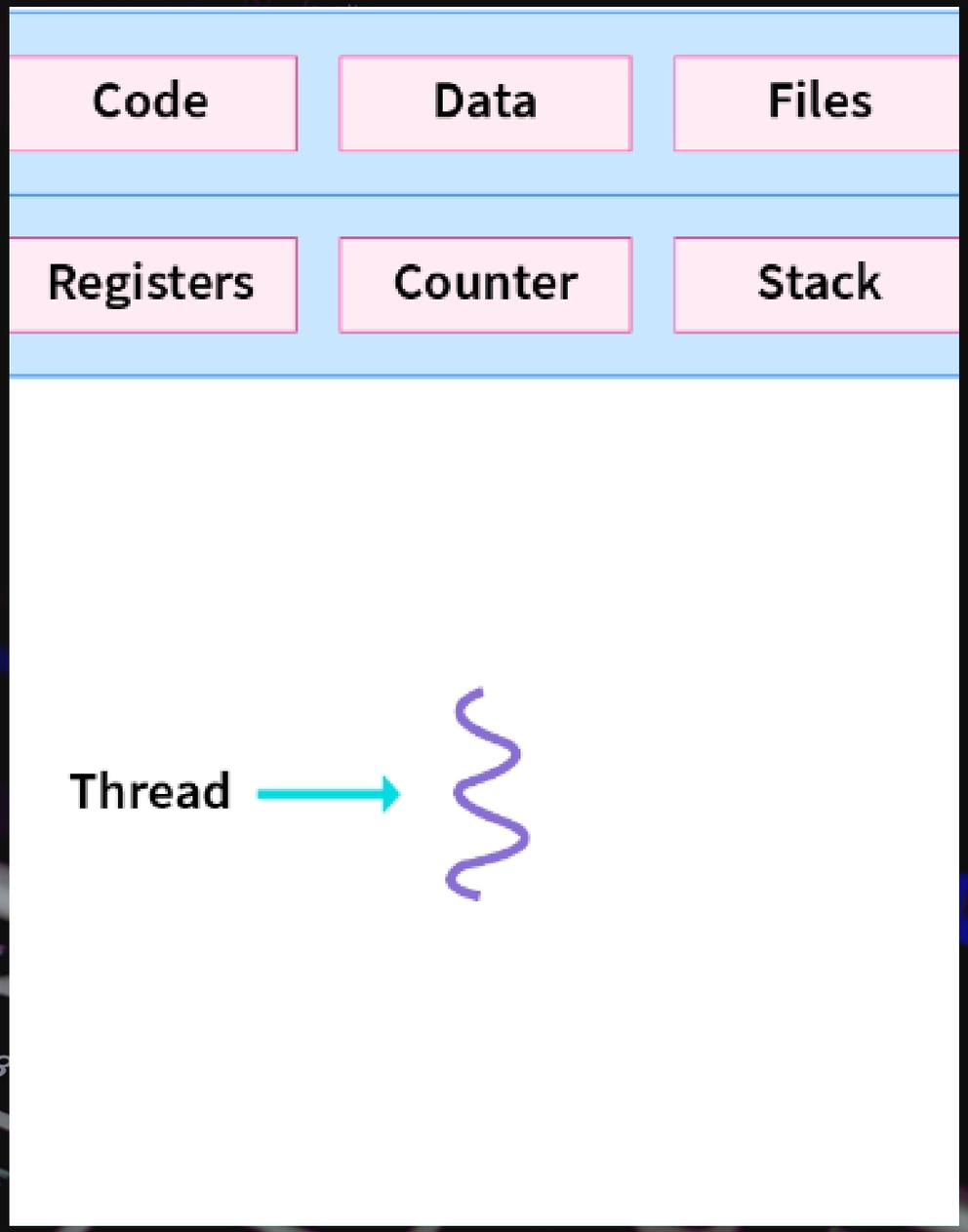
TYPES OF THREADING:

SINGLE THREADED

MULTI-THREADED

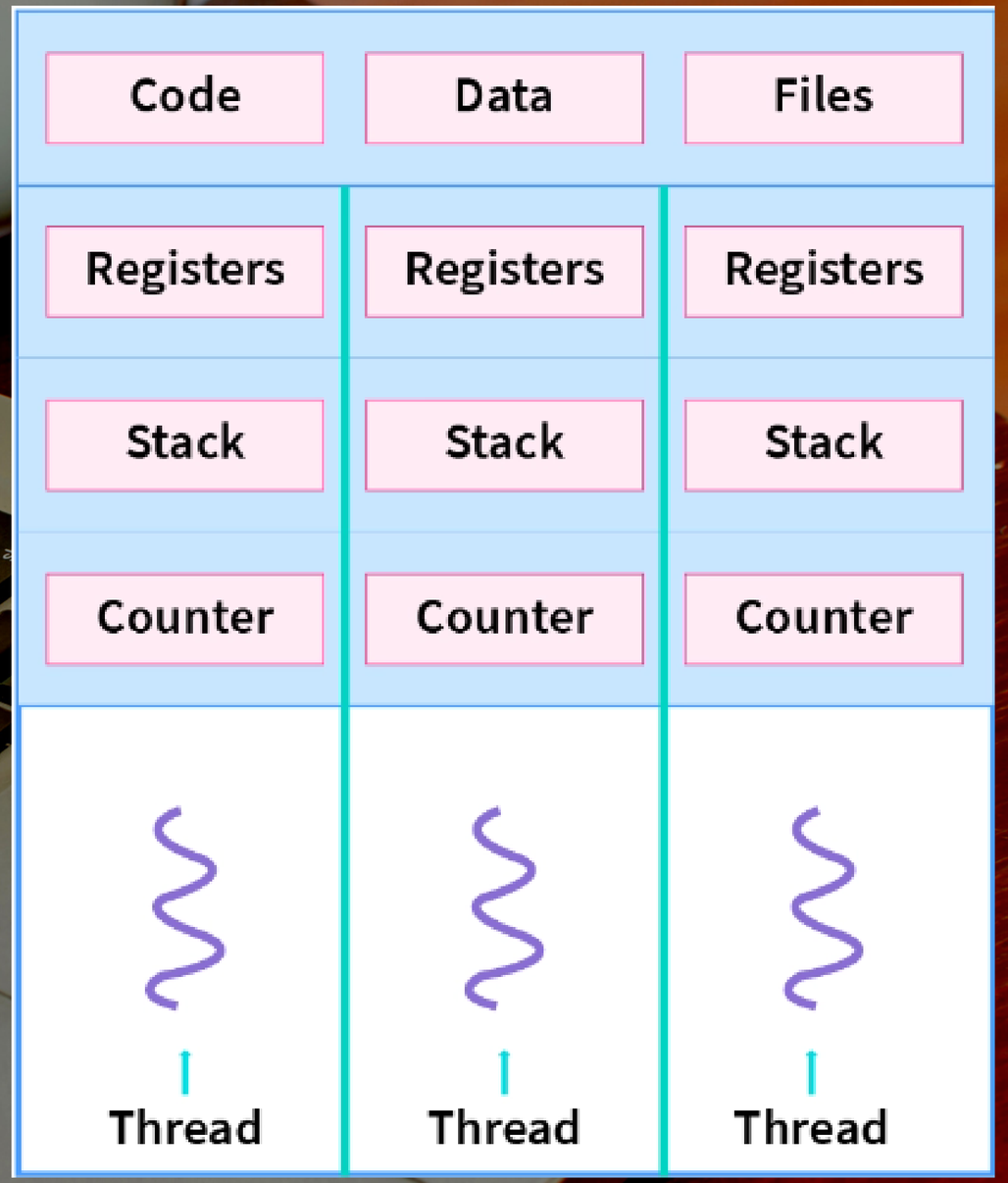
SINGLE THREADED

Single-threaded processes contain the execution of instructions in a single sequence. In other words, one command is processed at a time.



MULTI-THREADED:

Multithreading is a CPU (central processing unit) feature that allows two or more instruction threads to execute independently while sharing the same processor resources.



Why do we need Threads?

Threads in the operating system provide multiple benefits and improve the system's overall performance. Some of the reasons threads are needed in the operating system are:

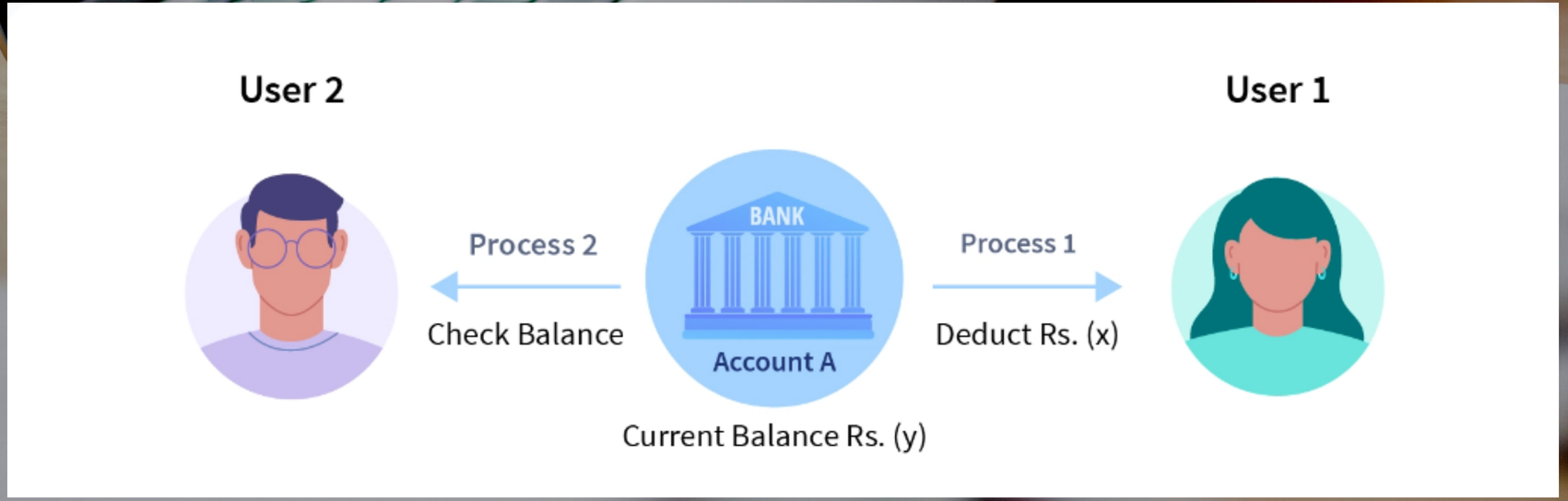
- Since threads use the same data and code, the operational cost between threads is low.
- Creating and terminating a thread is faster compared to creating or terminating a process.
- Context switching is faster in threads compared to processes.

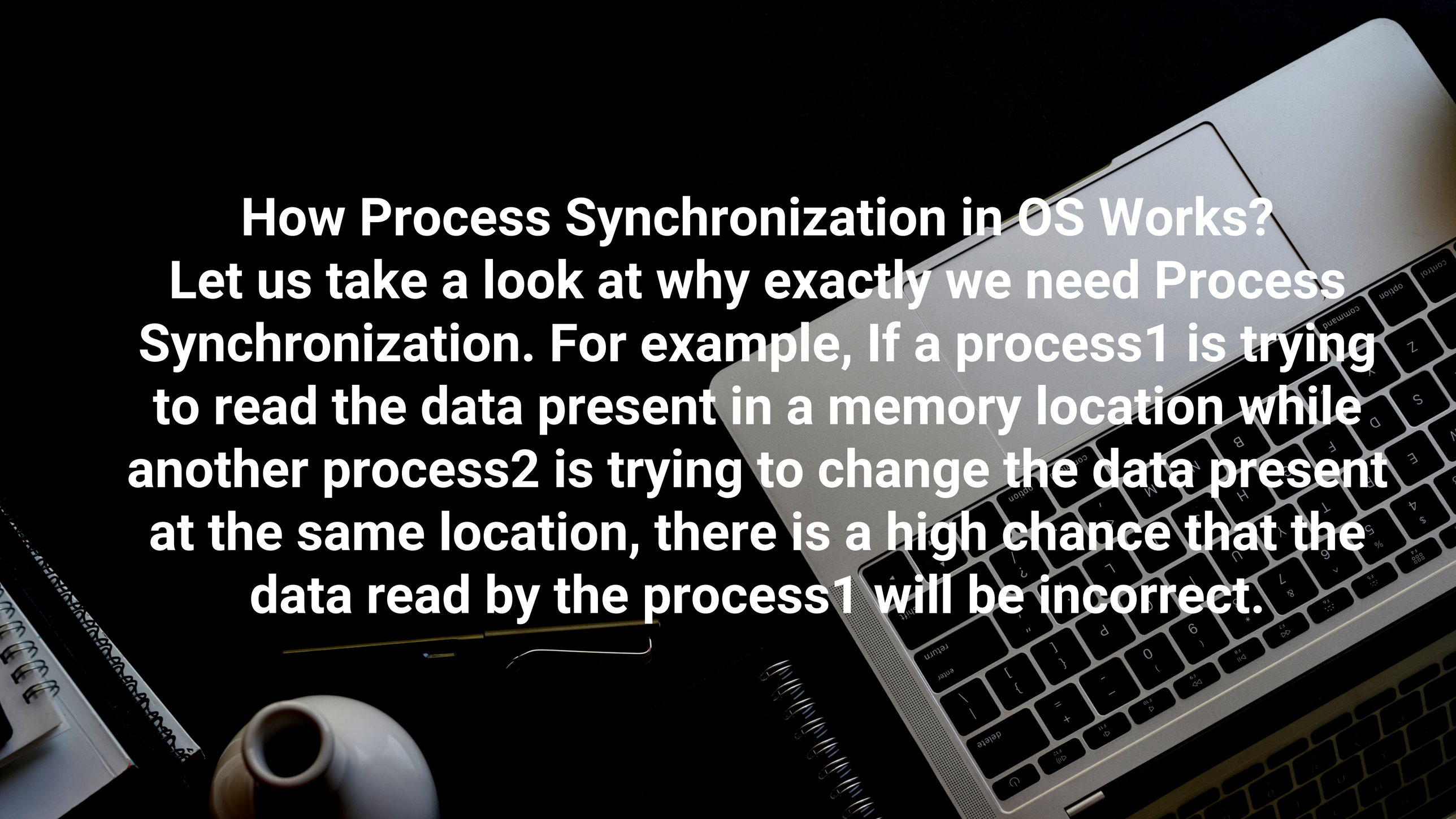
What is Process Synchronization in OS?

An operating system is software that manages all applications on a device and helps in the smooth functioning of our computer.

In the above image, if Process1 and Process2 happen at the same time, user 2 will get the wrong account balance as Y because of Process1 being transacted when the balance is X

Inconsistency of data can occur when various processes share a common resource in a system which is why there is a need for process synchronization in the operating system.



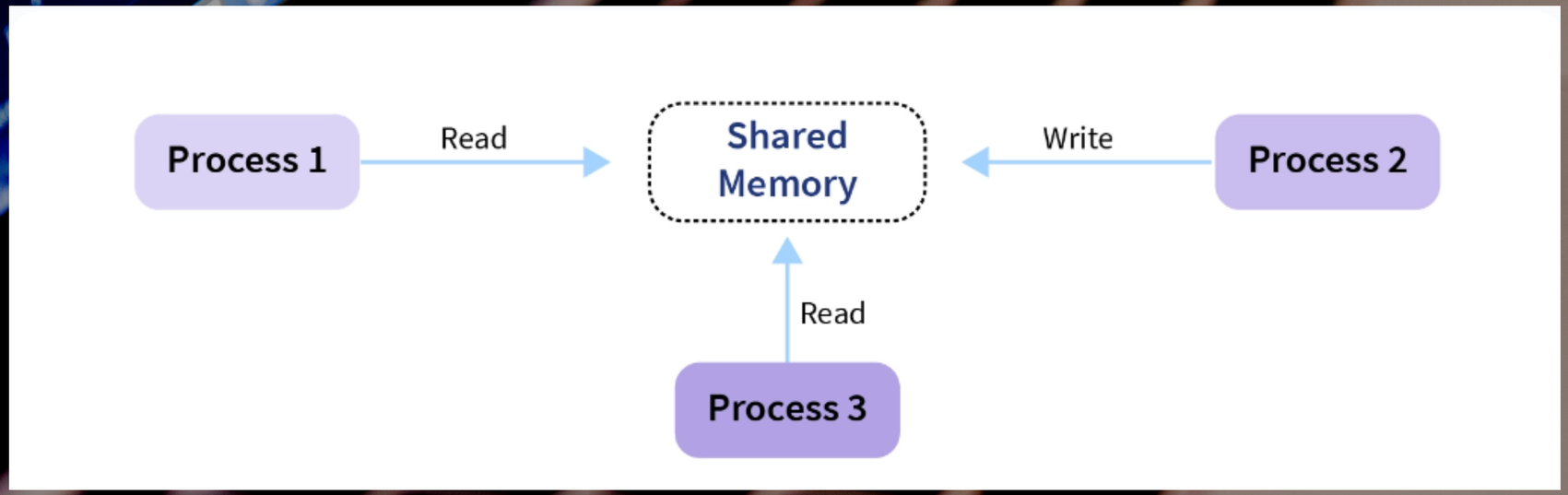


How Process Synchronization in OS Works?

Let us take a look at why exactly we need Process Synchronization. For example, If a process1 is trying to read the data present in a memory location while another process2 is trying to change the data present at the same location, there is a high chance that the data read by the process1 will be incorrect.

Let us look at different elements/sections of a program:

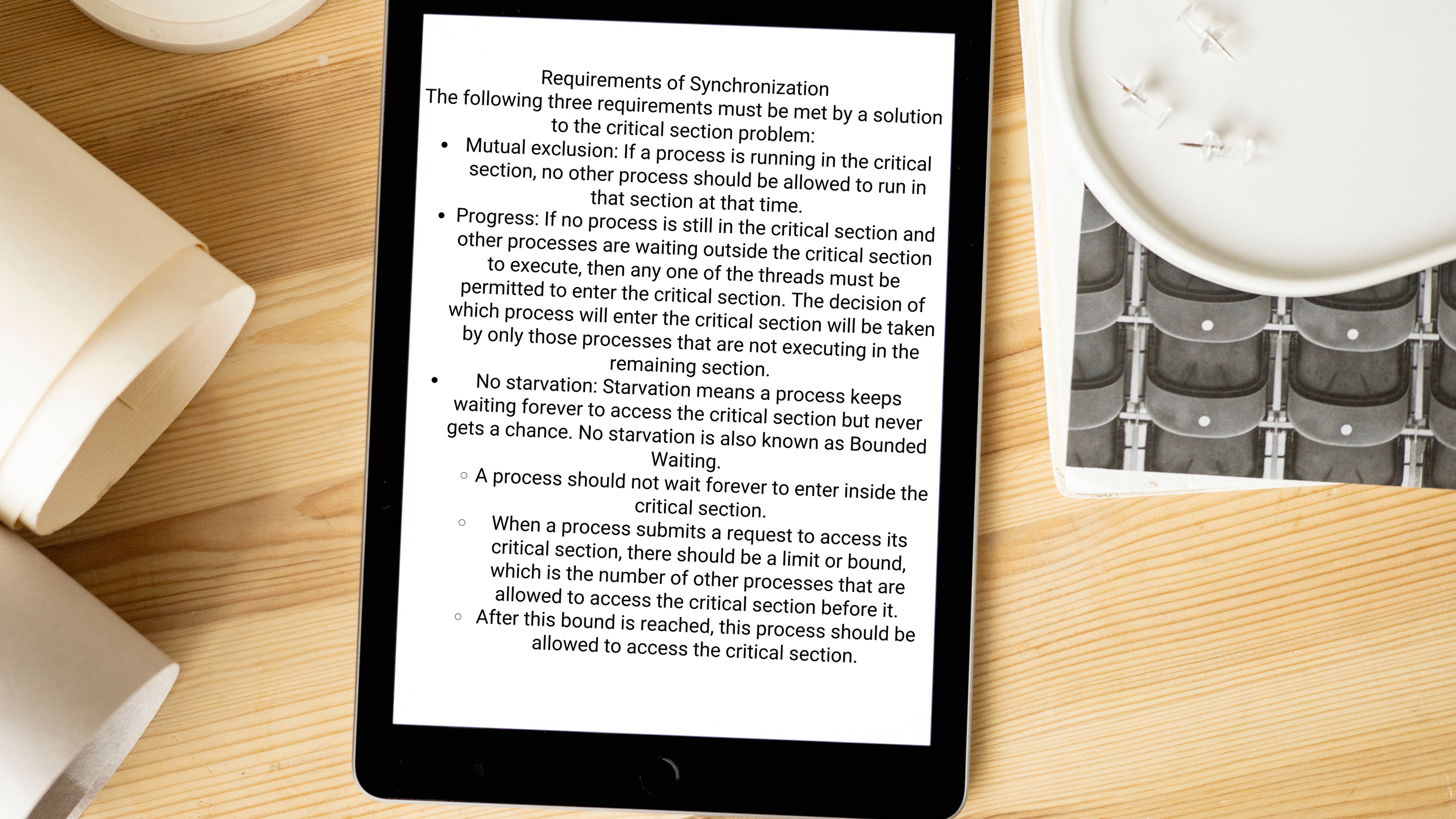
- Entry Section: The entry Section decides the entry of a process.
- Critical Section: Critical section allows and makes sure that only one process is modifying the shared data.
- Exit Section: The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- Remainder Section: The remaining part of the code which is not categorized as above is contained in the Remainder section.

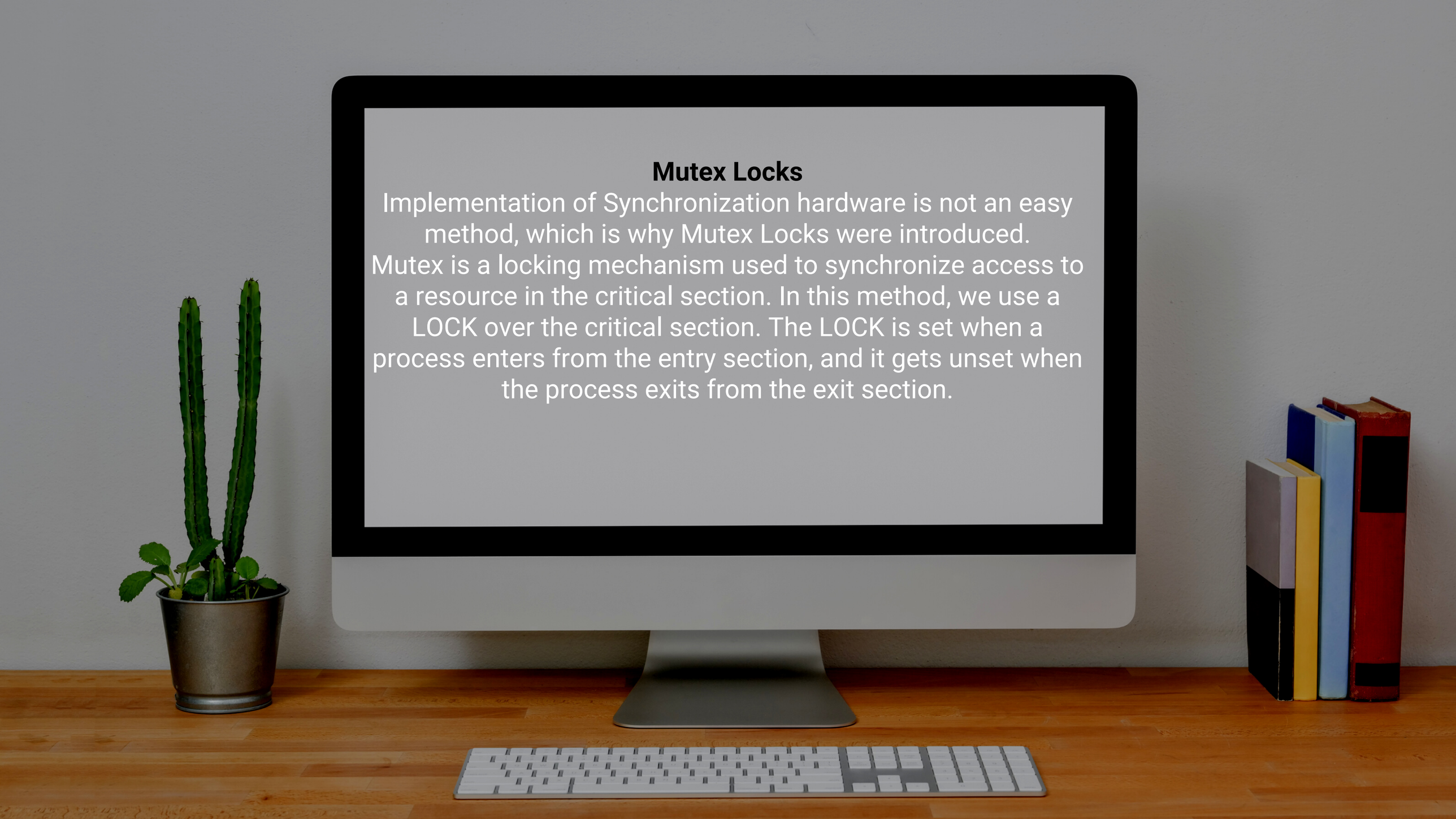


Requirements of Synchronization

The following three requirements must be met by a solution to the critical section problem:

- Mutual exclusion: If a process is running in the critical section, no other process should be allowed to run in that section at that time.
- Progress: If no process is still in the critical section and other processes are waiting outside the critical section to execute, then any one of the threads must be permitted to enter the critical section. The decision of which process will enter the critical section will be taken by only those processes that are not executing in the remaining section.
- No starvation: Starvation means a process keeps waiting forever to access the critical section but never gets a chance. No starvation is also known as Bounded Waiting.
  - A process should not wait forever to enter inside the critical section.
  - When a process submits a request to access its critical section, there should be a limit or bound, which is the number of other processes that are allowed to access the critical section before it.
  - After this bound is reached, this process should be allowed to access the critical section.




Mutex Locks
Implementation of Synchronization hardware is not an easy method, which is why Mutex Locks were introduced.
Mutex is a locking mechanism used to synchronize access to a resource in the critical section. In this method, we use a LOCK over the critical section. The LOCK is set when a process enters from the entry section, and it gets unset when the process exits from the exit section.

Working of a mutex: Suppose one thread has locked a region of code using mutex and is executing that piece of code. Now if the scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked. Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.