LEC-16: Critical Section Problem and How to address it

1. Process synchronization techniques play a key role in maintaining the consistency of shared data
2. **Critical Section (C.S)**
    a. The critical section refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid-execution.
3. **Major Thread scheduling issue**
    a. Race Condition
        i. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads are "racing" to access/change the data.
4. **Solution to Race Condition**
    a. Atomic operations: Make Critical code section an atomic operation, i.e., Executed in one CPU cycle.
    b. Mutual Exclusion using locks.
    c. Semaphores
5. Can we use a simple flag variable to solve the problem of race condition?
    a. No.
6. **Peterson's solution** can be used to avoid race condition but holds good for only 2 process/threads.
7. **Mutex/Locks**
    a. Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread/process to access critical section.
    b. Disadvantages:
        i. Contention: one thread has acquired the lock, other threads will be busy waiting, what if thread that had acquired the lock dies, then all other threads will be in infinite waiting.
        ii. Deadlocks
        iii. Debugging
        iv. Starvation of high priority threads.