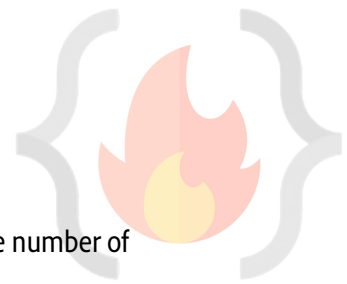
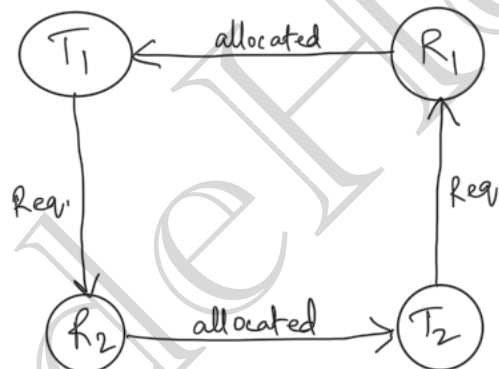


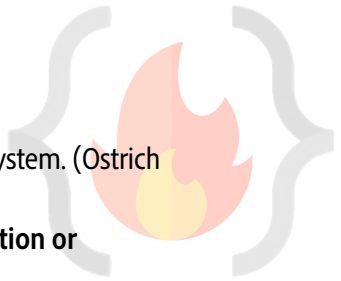
LEC-21: Deadlock Part-1



1. In Multi-programming environment, we have several processes competing for finite number of resources
2. Process requests a **resource (R)**, if R is not available (taken by other process), process enters in a waiting state. Sometimes that waiting process is never able to change its state because the resource, it has requested is busy (forever), called **DEADLOCK (DL)**
3. Two or more processes are waiting on some resource's availability, which will never be available as it is also busy with some other process. The Processes are said to be in **Deadlock**.
4. DL is a bug present in the process/thread synchronization method.
5. In DL, processes never finish executing, and the system resources are tied up, preventing other jobs from starting.
6. **Example of resources:** Memory space, CPU cycles, files, locks, sockets, IO devices etc.
7. Single resource can have multiple instances of that. E.g., CPU is a resource, and a system can have 2 CPUs.
8. How a process/thread utilize a resource?
 - a. Request: Request the R, if R is free Lock it, else wait till it is available.
 - b. Use
 - c. Release: Release resource instance and make it available for other processes



9. **Deadlock Necessary Condition:** 4 Condition should hold simultaneously.
 - a. **Mutual Exclusion**
 - i. Only 1 process at a time can use the resource, if another process requests that resource, the requesting process must wait until the resource has been released.
 - b. **Hold & Wait**
 - i. A process must be holding at least one resource & waiting to acquire additional resources that are currently being held by other processes.
 - c. **No-preemption**
 - i. Resource must be voluntarily released by the process after completion of execution. (No resource preemption)
 - d. **Circular wait**
 - i. A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , and so on.
10. **Methods for handling Deadlocks:**
 - a. Use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlocked state.
 - b. Allow the system to enter a deadlocked state, **detect it, and recover**.

- 
- c. Ignore the problem altogether and pretend that deadlocks never occur in system. (Ostrich algorithm) aka, **Deadlock ignorance**.
 - 11. To ensure that deadlocks never occur, the system can use either a **deadlock prevention or deadlock avoidance scheme**.
 - 12. **Deadlock Prevention**: by ensuring at least one of the necessary conditions cannot hold.
 - a. **Mutual exclusion**
 - i. Use locks (mutual exclusion) only for non-sharable resource.
 - ii. Sharable resources like Read-Only files can be accessed by multiple processes/threads.
 - iii. However, we can't prevent DLs by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.
 - b. **Hold & Wait**
 - i. To ensure H&W condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it doesn't hold any other resource.
 - ii. Protocol (A) can be, each process has to request and be allocated all its resources before its execution.
 - iii. Protocol (B) can be, allow a process to request resources only when it has none. It can request any additional resources after it must have released all the resources that it is currently allocated.
 - c. **No preemption**
 - i. If a process is holding some resources and request another resource that cannot be immediately allocated to it, then all the resources the process is currently holding are preempted. The process will restart only when it can regain its old resources, as well as the new one that it is requesting. (Live Lock may occur).
 - ii. If a process requests some resources, we first check whether they are available. If yes, we allocate them. If not, we check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resource from waiting process and allocate them to the requesting process.
 - d. **Circular wait**
 - i. To ensure that this condition never holds is to impose a proper ordering of resource allocation.
 - ii. P1 and P2 both require R1 and R1, locking on these resources should be like, both try to lock R1 then R2. By this way which ever process first locks R1 will get R2.