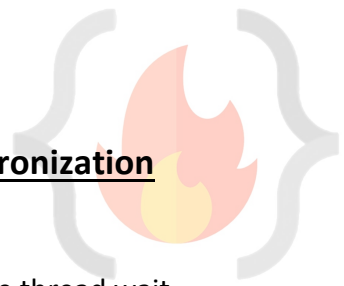


## LEC-16: Critical Section Problem and How to address it



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.

## LEC-17: Conditional Variable and Semaphores for Threads synchronization



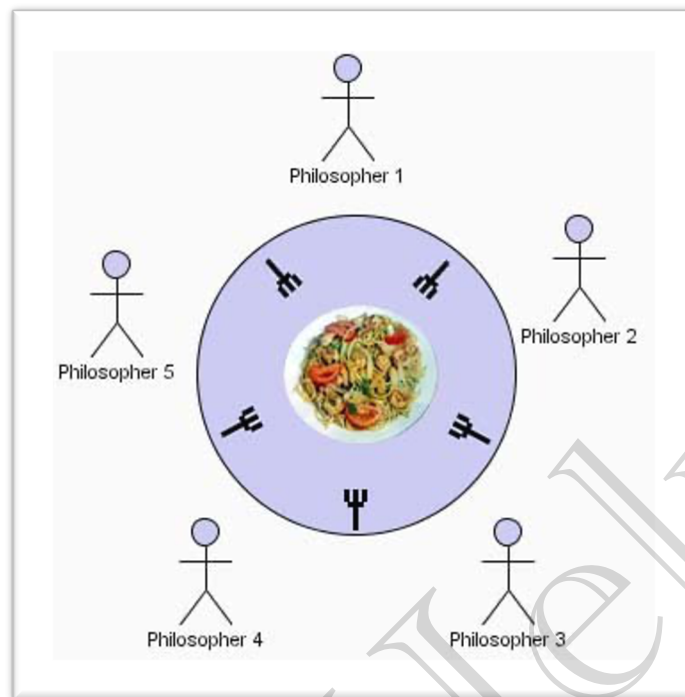
### 1. Conditional variable

- a. The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs.
- b. Works with a lock
- c. Thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock and wait until another thread notifies that the event has occurred. Once the waiting thread enters the running state, it again acquires the lock immediately and starts executing.
- d. Why to use conditional variable?
  - i. To avoid busy waiting.
- e. **Contention** is not here.

### 2. Semaphores

- a. Synchronization method.
- b. An integer that is equal to number of resources
- c. Multiple threads can go and execute C.S concurrently.
- d. Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time.
- e. Binary semaphore: value can be 0 or 1.
  - i. Aka, mutex locks
- f. Counting semaphore
  - i. Can range over an unrestricted domain.
  - ii. Can be used to control access to a given resource consisting of a finite number of instances.
- g. To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block- operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the Waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
- h. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

## Lec-20: The Dining Philosophers problem



1. We have **5 philosophers**.
2. They spend their life just being in **two states**:
  - a. Thinking
  - b. Eating
3. They sit on a circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodles, and the table is laid with 5 single forks.
4. **Thinking state**: When a ph. Thinks, he doesn't interact with others.
5. **Eating state**: When a ph. Gets hungry, he tries to pick up the 2 forks adjacent to him (Left and Right). He can pick one fork at a time.
6. One can't pick up a fork if it is already taken.
7. When ph. Has both forks at the same time, he eats without releasing forks.
8. Solution can be given using semaphores.
  - a. Each fork is a binary semaphore.
  - b. A ph. Calls wait() operation to acquire a fork.
  - c. Release fork by calling signal().
  - d. **Semaphore fork[5]{1};**
9. Although the semaphore solution makes sure that no two neighbors are eating simultaneously but it could still create **Deadlock**.
10. Suppose that all 5 ph. Become hungry at the same time and each picks up their left fork, then All fork semaphores would be 0.
11. When each ph. Tries to grab his right fork, he will be waiting for ever (Deadlock)
12. We must use **some methods to avoid Deadlock and make the solution work**
  - a. Allow at most 4 ph. To be sitting simultaneously.
  - b. Allow a ph. To pick up his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically).

c. **Odd-even rule.**

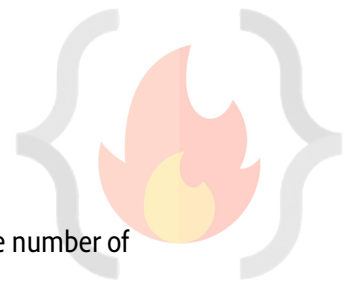
an odd ph. Picks up first his left fork and then his right fork, whereas an even ph. Picks up his right fork then his left fork.

13. Hence, only semaphores are not enough to solve this problem.

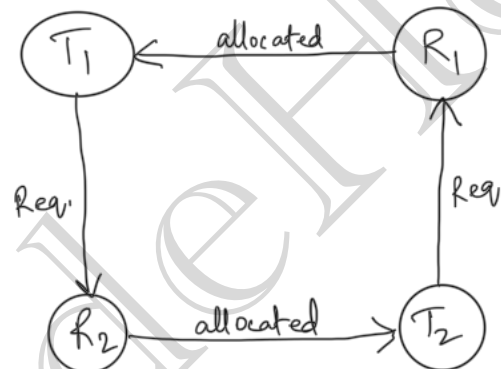
We must add some enhancement rules to make deadlock free solution.

CodeHelp

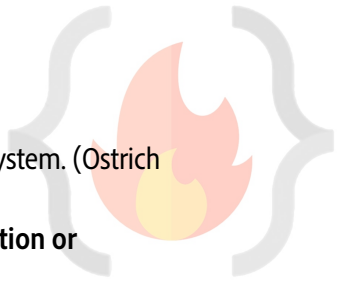
## LEC-21: Deadlock Part-1



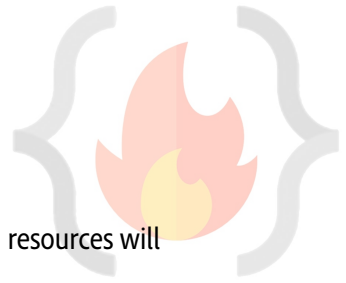
1. In Multi-programming environment, we have several processes competing for finite number of resources
2. Process requests a **resource (R)**, if R is not available (taken by other process), process enters in a waiting state. Sometimes that waiting process is never able to change its state because the 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.

## LEC-22: Deadlock Part-2



1. **Deadlock Avoidance:** Idea is, the kernel be given in advance info concerning which resources will use in its lifetime.

By this, system can decide for each request whether the process should wait.

To decide whether the current request can be satisfied or delayed, the system must consider the resources currently available, resources currently allocated to each process in the system and the future requests and releases of each process.

- a. Schedule process and its resources allocation in such a way that the DL never occur.
- b. Safe state: A state is safe if the system can allocate resources to each process (up to its max.) in some order and still avoid DL.  
A system is in safe state only if there exists a safe sequence.
- c. In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.
- d. The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the resulting state is a safe state.
- e. In a case, if the system is unable to fulfill the request of all processes, then the state of the system is called unsafe.
- f. Scheduling algorithm using which DL can be avoided by finding safe state. (Banker Algorithm)

### 2. Banker Algorithm

- a. When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state. If yes, then the resources may be allocated to the process. If not, then the process must wait till other processes release enough resources.

### 3. Deadlock Detection: Systems haven't implemented deadlock-prevention or a deadlock avoidance technique, then they may employ DL detection then, recovery technique.

- a. Single Instance of Each resource type (**wait-for graph method**)
  - i. A deadlock exists in the system if and only if there is a cycle in the wait-for graph. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the wait-for graph.
- b. Multiple instances for each resource type
  - i. Banker Algorithm

### 4. Recovery from Deadlock

- a. Process termination
  - i. Abort all DL processes
  - ii. Abort one process at a time until DL cycle is eliminated.
- b. Resource preemption
  - i. To eliminate DL, we successively preempt some resources from processes and give these resources to other processes until DL cycle is broken.