


OBJECTIVES/TOPICS

- The Critical Section Problem.
- Semaphores.
- Classical Problems of Synchronization.
- Deadlocks.
- System Model.
- Deadlocks Characterization.
- Methods for Handling Deadlocks.
- Deadlock Prevention, Avoidance, Detection, & Recovery.

INTRODUCTION

- A system typically consists of several (perhaps hundreds or even thousands) of threads running either concurrently or in parallel
- Threads often share user data.
- Meanwhile, the OS continuously updates various data structures to support multiple threads.
- A race condition exists when access to shared data is not controlled, possibly resulting in corrupt data values.

INTRODUCTION

- A race condition occurs when multiple threads read and write the same variable i.e. they have access to some shared data & they try to change it at the same time
- Process synchronization involves using tools that control access to ~~shared data~~ to avoid race conditions.

- These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock

INTRODUCTION

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (i.e, both code & data) or be allowed to share data only through shared memory or message passing

INTRODUCTION

- Concurrent access to shared data may result in data inconsistency, however.
- We will discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained

BACKGROUND

- Processes can execute concurrently or in parallel
- The CPU scheduler switches rapidly between processes to provide concurrent execution
 - This means that one process may only partially complete execution before another process is scheduled.
 - In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process

BACKGROUND

- In parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores
- In this topic we will explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

THE CRITICAL SECTION PROBLEM

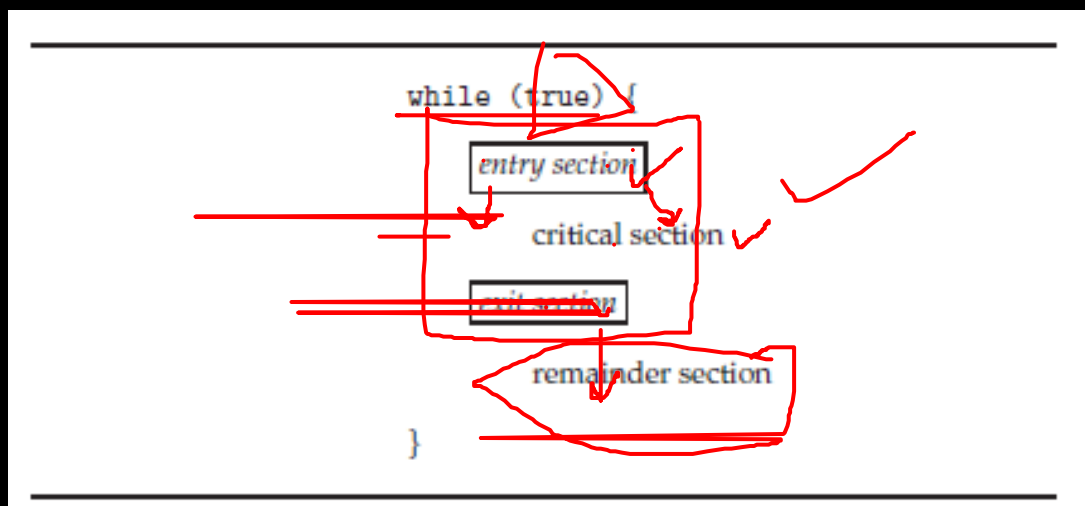
- Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$.
- Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section

THE CRITICAL SECTION PROBLEM

- That is, no two processes are executing in their critical sections at the same time.
- The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data
- Each process must request permission to enter its critical section.
 - The section of code implementing this request is the entry section

THE CRITICAL SECTION PROBLEM

- The critical section may be followed by an exit section.
- The remaining code is the remainder section
- The general structure of a typical process is shown below.



THE CRITICAL SECTION PROBLEM

- The entry section and exit section are enclosed in boxes to highlight these important segments of code.
- A solution to the critical-section problem must satisfy the following three requirements:
 - 1. **Mutual exclusion.** ✓
 - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

THE CRITICAL SECTION PROBLEM

- **2. Progress**
 - If no process is executing in its critical section & some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely

THE CRITICAL SECTION PROBLEM

- **3. Bounded waiting**

- There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that request is granted.

- We assume that each process is executing at a nonzero speed.

THE CRITICAL SECTION PROBLEM

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non-preemptive kernels**
 - A preemptive kernel allows a process to be preempted while it is running in kernel mode
 - A non-preemptive kernel does not allow a process running in kernel mode to be preempted;
 - a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

THE CRITICAL SECTION PROBLEM

- Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.
- We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions

SEMAPHORES

- Semaphores are robust tools which provide more sophisticated ways for processes to synchronize their activities.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

SEMAPHORES

- The definition of `wait()` is as follows:

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- The definition of `signal()` is as follows:

```
signal(S) {  
    S++;  
}
```

SEMAPHORES

- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically.
 - That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value
- In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification ($S--$), must be executed without interruption

SEMAPHORES: Semaphore usage

- Operating systems often distinguish between counting and binary semaphores.
 - The value of a counting semaphore can range over an unrestricted domain.
 - The value of a binary semaphore can range only between 0 and 1
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances

SEMAPHORES: Semaphore usage

- The semaphore is initialized to the number of resources available $\zeta = 2$
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count)
 - When the count for the semaphore goes to 0, all resources are being used

SEMAPHORES: Semaphore usage

- After that, processes that wish to use a resource will block until the count becomes greater than 0.
- We can also use semaphores to solve various synchronization problems.
 - For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 .
 - Suppose we require that S_2 be executed only after S_1 has completed.

SEMAPHORES: Semaphore usage

- We can implement this scheme readily by letting P_1 & P_1 share a common semaphore synch initialized to 0.
- In process P_1 , we insert the statements

```
S1;  
signal(synch);
```

- In process P_2 , we insert the statements

```
wait(synch);  
S2;
```

SEMAPHORES: Semaphore usage

- Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after statement S_1 has been executed.

DEADLOCKS

- In a multiprogramming environment, several threads may compete for a finite number of resources.
- A thread requests resources; if the resources are not available at that time, the thread enters a waiting state.
- Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads.
- This situation is called a deadlock

SYSTEM MODEL

- A system consists of a finite number of resources to be distributed among a number of competing threads
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances
 - CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) are examples of resource types.

SYSTEM MODEL

- If a system has four CPUs, then the resource type CPU has four instances.
- Similarly, the resource type network may have two instances.
- If a thread requests an instance of a resource type, the allocation of any instance of the type should satisfy the request.
- If it does not, then the instances are not identical, and the resource type classes have not been defined properly

SYSTEM MODEL

- A thread must request a resource before using it and must release the resource after using it.
- A thread may request as many resources as it requires to carry out its designated task.
 - Obviously, the number of resources requested may not exceed the total number of resources available in the system.
- In other words, a thread cannot request two network interfaces if the system has only one.

SYSTEM MODEL

- Under the normal mode of operation, a thread may utilize a resource in only the following sequence:
 - **1. Request.**
 - The thread requests the resource.
 - If the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource.
 - **2. Use.**
 - The thread can operate on the resource
 - **3. Release.**
 - The thread releases the resource.

SYSTEM MODEL

- The request and release of resources may be system calls
- Examples are the request() and release() of a device, open() and close() of a file, and allocate() and free() memory system calls
- Similarly, request and release can be accomplished through the wait() and signal() operations on semaphores and through acquire() and release() of a mutex lock.

SYSTEM MODEL

- For each use of a kernel-managed resource by a thread, the operating system checks to make sure that the thread has requested and has been allocated the resource.
- A system table records whether each resource is free or allocated.
- For each resource that is allocated, the table also records the thread to which it is allocated.

SYSTEM MODEL

- If a thread requests a resource that is currently allocated to another thread, it can be added to a queue of threads waiting for this resource.
- A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set.
- The events with which we are mainly concerned here are resource acquisition and release.

DEADLOCK CHARACTERIZATION

Necessary Conditions

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
 - **1. Mutual exclusion.**
 - At least 1 resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource.
 - If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

DEADLOCK CHARACTERIZATION

Necessary Conditions

- **2. Hold and wait.**
 - A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
- **3. No preemption.**
 - Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.

DEADLOCK CHARACTERIZATION

Necessary Conditions

- 4. Circular wait.
- A set $\{T_0, T_1, \dots, T_n\}$ of waiting threads must exist such that T_0 is waiting for a resource held by T_1 , T_1 is waiting for a resource held by T_2 , ..., T_{n-1} is waiting for a resource held by T_n , and T_n is waiting for a resource held by T_0 .
- We emphasize that all four conditions must hold for a deadlock to occur.

DEADLOCK CHARACTERIZATION

Necessary Conditions

- The circular-wait condition implies the hold-and-wait condition, so the ~~four~~ conditions are not completely independent.

METHODS FOR HANDLING DEADLOCKS

- Generally speaking, we can deal with the deadlock problem in one of three ways:
 - 1. We can ignore the problem altogether and pretend that deadlocks never occur in the system.
 - 2. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
 - 3. We can allow the system to enter a deadlocked state, detect it, and recover

METHODS FOR HANDLING DEADLOCKS

- The first solution is the one used by most operating systems, including Linux and Windows.
- It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution.
- Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

METHODS FOR HANDLING DEADLOCKS

- To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme.
- Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions does not occur/hold.
- Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime.

METHODS FOR HANDLING DEADLOCKS

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise.
- In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

DEADLOCK PREVENTION

- for a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.
- We elaborate on this approach by examining each of the four necessary conditions separately.

DEADLOCK PREVENTION

Mutual Exclusion

- The mutual-exclusion condition must hold.
 - That is, at least one resource must be non-sharable.
- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock
 - Read-only files are a good example of a sharable resource.

DEADLOCK PREVENTION

Mutual Exclusion

- If several threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A thread never needs to wait for a sharable resource.

DEADLOCK PREVENTION

Hold and wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources.
- One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution
 - This is of course impractical for most applications due to the dynamic nature of requesting resources.

DEADLOCK PREVENTION

Hold and wait

- An alternative protocol allows a thread to request resources only when it has none.
 - A thread may request some resources and use them.
 - Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Both alternatives have issues of starvation and resource underutilization

DEADLOCK PREVENTION

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, we can use the following protocol
 - If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted.

DEADLOCK PREVENTION

No Preemption

- Alternatively, if a thread requests some resources, we first check whether they are available.
 - If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources.
 - If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread.
 - If the resources are neither available nor held by a waiting thread, the requesting thread must wait.

DEADLOCK PREVENTION

Circular Wait

- The three options presented thus far for deadlock prevention are generally impractical in most situations.
- However, the fourth and final condition for deadlocks — the circular-wait condition — presents an opportunity for a practical solution by invalidating one of the necessary conditions.

DEADLOCK PREVENTION

Circular Wait

- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.
- To illustrate, we let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types.

DEADLOCK PREVENTION

Circular Wait

- We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering
- Each thread can request resources only in an increasing order of enumeration.
 - That is, a thread can initially request an instance of a resource—say, R_i .
 - After that, the thread can request an instance of resource R_j if and only if $j > i$.

DEADLOCK AVOIDANCE

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested
- For example, in a system with resources R_1 and R_2 , the system might need to know that thread P will request first R_1 and then R_2 before releasing both resources, whereas thread Q will request R_2 and then R_1 .

DEADLOCK AVOIDANCE

- With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

DEADLOCK AVOIDANCE

- The simplest and most useful model requires that each thread declare the maximum number of resources of each type that it may need.
- A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist
- The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the threads

DEADLOCK AVOIDANCE

Safe State

- A state is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock
- More formally, a system is in a safe state only if there exists a safe sequence
 - A sequence of threads $\langle T_1, T_2, \dots, T_n \rangle$ is a safe sequence for the current allocation state if, for each T_i , the resource requests that T_i can still make can be satisfied by the currently available resources plus the resources held by all T_j , with $j < i$.

DEADLOCK AVOIDANCE

Safe State

- In this situation, if the resources that T_i needs are not immediately available, then T_i can wait until all T_j have finished
- When they have finished, T_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate

DEADLOCK AVOIDANCE

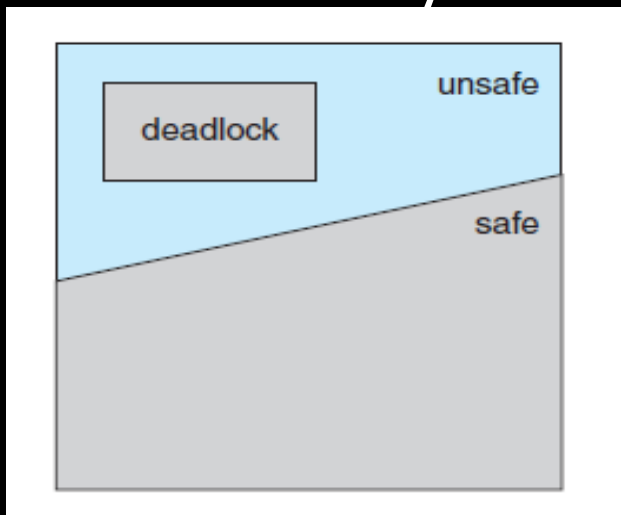
Safe State

- When T_i terminates, T_{i+1} can obtain its needed resources, and so on.
- If no such sequence exists, then the system state is said to be unsafe

DEADLOCK AVOIDANCE

Safe State

- A safe state is not a deadlocked state.
- Conversely, a deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks, however
- An unsafe state may lead to a deadlock



DEADLOCK AVOIDANCE

Safe State

- To illustrate, consider a system with twelve resources and three threads: T_0 , T_1 , and T_2 .
 - Thread T_0 requires ten resources,
 - thread T_1 may need as many as four,
 - thread T_2 may need up to nine resources.
- Suppose that, at time t_0 ,
 - thread T_0 is holding five resources,
 - thread T_1 is holding two resources,
 - thread T_2 is holding two resources.

DEADLOCK AVOIDANCE

Safe State

- Thus, there are three free resources

	<u>Maximum Needs</u>	<u>Current Needs</u>
T_0	10	5
T_1	4	2
T_2	9	2 \geq

- At time t_0 , the system is in a safe state
 - The sequence $\langle T_1, T_0, T_2 \rangle$ satisfies the safety condition.

DEADLOCK AVOIDANCE

Safe State

- Thread T_1 can immediately be allocated all its resources & then return them (the system will then have five available resources);
- Then thread T_0 can get all its resources and return them (the system will then have ten available resources);
- finally thread T_2 can get all its resources and return them (the system will then have all twelve resources available).

DEADLOCK AVOIDANCE

Safe State

- A system can go from a safe state to an unsafe state.
 - Suppose that, at time t_1 , thread T_2 requests and is allocated one more resource.
 - The system is no longer in a safe state.
 - At this point, only thread T_1 can be allocated all its resources.
 - When it returns them, the system will have only four available resources

DEADLOCK AVOIDANCE

Safe State

- Since thread T_0 is allocated five resources but has a maximum of ten, it may request five more resources.
 - If it does so, it will have to wait, because they are unavailable
- Similarly, thread T_2 may request six additional resources and have to wait, resulting in a deadlock.
 - Our mistake was in granting the request from thread T_2 for one more resource.

DEADLOCK AVOIDANCE

Safe State

- If we had made T_2 wait until either of the other threads had finished and released its resources, then we could have avoided the deadlock.

DEADLOCK DETECTION

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
 - An algorithm that examines the state of the system to determine whether a deadlock has occurred
 - An algorithm to recover from the deadlock

DEADLOCK DETECTION

- When should we invoke the detection algorithm? The answer depends on two factors:
 - 1. How often is a deadlock likely to occur?
 - 2. How many threads will be affected by deadlock when it happens?

RECOVERY FROM DEADLOCKS

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
 - One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
 - Another possibility is to let the system recover from the deadlock automatically.

RECOVERY FROM DEADLOCKS

- Deadlock Recovery Algorithms include
 - 1. Process and Thread Termination
 - 2. Resource Preemption

READ ON THESE.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

- It is commonly used to illustrate the power of synchronization primitives.
- Here, we present a general structure of this scheme without committing ourselves to any particular implementation.
- In our problem, the producer and consumer processes share the following data structures:

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

```
int n;  
semaphore mutex = 1;  
semaphore empty = n;  
semaphore full = 0
```

- A Producer is a process which is able to produce data/item.
- A Consumer is a Process that is able to consume the data/item produced by the Producer.
- Both Producer and Consumer share a common memory buffer

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

- We assume that the pool consists of n buffers, each capable of holding one item.
- The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1
 - Mutual exclusion is a program object that blocks multiple users from accessing the same shared variable or data at the same time

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

- The empty and full semaphores count the number of empty and full buffers.
- The code for the producer process is shown below

```
while (true) {  
    . . .  
    /* produce an item in next_produced */  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    /* add next_produced to the buffer */  
    . . .  
    signal(mutex);  
    signal(full);  
}
```

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

- The code for the consumer process is shown below

```
while (true) {  
    wait(full);  
    wait(mutex);  
  
    . . .  
    /* remove an item from buffer to next_consumed */  
    . . .  
    signal(mutex);  
    signal(empty);  
  
    . . .  
    /* consume the item in next_consumed */  
    . . .  
}
```

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

1. The Bounded-Buffer Problem

- Note the symmetry between the producer and the consumer.
- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

2. The Readers–Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas other may want to update (that is, read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

2. The Readers–Writers Problem

- Obviously, if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
 - This is referred to as the readers–writers problem.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

2. The Readers–Writers Problem

- The readers–writers problem has several variations, all involving priorities.
- The simplest one, referred to as the first readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object
 - In other words, no reader should wait for other readers to finish simply because a writer is waiting

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

2. The Readers–Writers Problem

- The second readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible.
 - In other words, if a writer is waiting to access the object, no new readers may start reading
- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

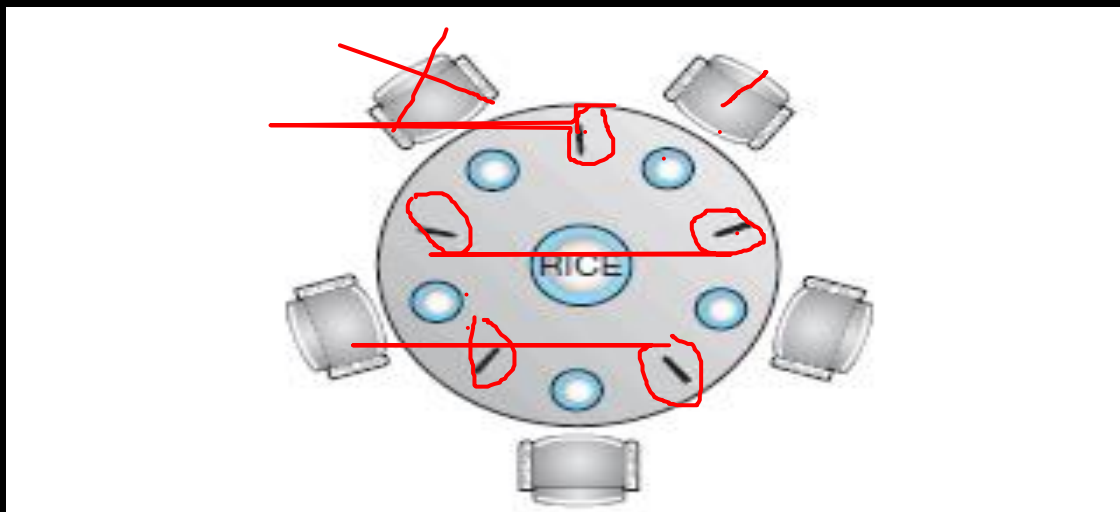
CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.'
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks
- As shown in the next diagram

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem



- When a philosopher thinks, she does not interact with her colleagues

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- From time to time, a philosopher gets hungry & tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time.
 - Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks
- When she is finished eating, she puts down both chopsticks and starts thinking again.
- The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- 1 solution is to represent each chopstick with semaphore.
 - A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
 - She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
 - Thus, the shared data are

```
semaphore chopstick[5];
```

- where all the elements of chopstick are initialized to 1.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- The structure of philosopher i is shown below:

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);

    . . .
    /* eat for a while */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* think for awhile */
    . . .
}
```

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick
- All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab her right chopstick, she will be delayed forever.

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- Several possible remedies to the deadlock problem are the following:
 - i. Allow at most four philosophers to be sitting simultaneously at the table.
 - li. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

3. The Dining-Philosophers Problem

- Several possible remedies to the deadlock problem are the following:
 - iii. Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

THE END