MIT-ADT UNIVERSITY
PUNE, INDIA

Third Year Engineering

21BTCS504-Operating System

Class - T.Y. (SEM-I)

Unit - III

- PROCESS SYNCHRONIZATION AND DEADLOCKS

AY 2023-2024 **SEM-I**

Unit-III Syllabus

Background, The Critical-Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores, Classic Problems of Synchronization, Monitors, Synchronization Example, Atomic Transactions, Deadlocks, System Model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock

Background

- Process synchronization in operating systems refers to the coordination and orderly execution of multiple processes or threads in a way that prevents undesirable conflicts, race conditions, and inconsistencies when they access shared resources, such as memory, files, or devices.
- In a multitasking or multi-threading environment, where multiple processes or threads run concurrently, there are situations where processes need to interact with each other or share resources.
- If these interactions are not properly managed, various issues can arise, such as data corruption, deadlock, and inconsistent behavior. Process synchronization mechanisms provide ways to handle these interactions effectively and avoid these problems.

Some common scenarios that require process synchronization include:

Mutual Exclusion

Deadlock Avoidance

Producer-Consumer Problem

Readers-Writers Problem

Semaphore

The Critical-Section Problem

- It refers to a situation where multiple processes or threads in a program share a common resource, and each process has a section of code called the "critical section" in which it accesses and modifies the shared resource.
- The critical section problem aims to find a way to ensure that these processes can access the shared resource safely and efficiently, without conflicts or inconsistencies.

This critical section problem is defined by several requirements:

- **Mutual Exclusion:** At any given time, only one process should be allowed to execute its critical section. This ensures that no two processes access the shared resource simultaneously, preventing data corruption and ensuring consistency.
- **Progress**: If no process is currently in its critical section and some processes want to enter, then a process that wants to enter its critical section must be allowed to do so without waiting indefinitely. This ensures that processes continue to make progress and don't get stuck.

Peterson's Solution

- Peterson's Solution is a classic synchronization algorithm used in operating systems to address the critical section problem.
- The critical section problem occurs when multiple processes or threads in a system share a common resource, such as a data structure or a section of code, and they need to ensure that only one process can access the critical section at any given time to prevent data corruption or conflicts.
- In Peterson's Solution t assumes a scenario with two processes, P0 and P1, and provides a turn-based approach

Peterson's Solution works:

1.Shared variables:

- int turn: A variable that indicates whose turn it is to enter the critical section. It can take on the values 0 or 1, where 0 corresponds to P0's turn and 1 corresponds to P1's turn.
- bool flag[2]: An array of Boolean flags, one for each process. It is used to indicate whether a process is ready to enter the critical section. Initially, both flags are set to false.

2. Process code:

- Each process must follow these steps before entering the critical section:
- a. Set its flag to true, indicating its intention to enter the critical section.
- b. Set turn to the other process's identifier (0 or 1), indicating that it's the other process's turn.

C. Check if the other process's flag is true and if it's the other process's turn. If both conditions are met, the process must wait until it's its turn and the other process has released its flag.

3. Exit from the critical section:

- After a process completes its work in the critical section, it sets its flag to false, indicating that it's done with the critical section.

- The key idea behind Peterson's Solution is that the processes take turns trying to enter the critical section. If one process tries to enter while the other is inside the critical section (flag[other] == true), it will be forced to wait until it's the other process's turn (turn == other). This ensures that only one process can be in the critical section at a time, preventing race conditions and ensuring mutual exclusion.

Synchronization Hardware

- synchronization hardware refers to the hardware-level mechanisms and features provided by modern computer architectures to facilitate synchronization and coordination between multiple threads or processes.
- These hardware mechanisms are essential for ensuring that concurrent operations are executed safely and correctly, particularly when multiple threads or processes are accessing shared resources, such as memory locations or I/O devices.
- key aspects of synchronization hardware:
- **Atomic Operations:** Many modern processors support atomic operations, which are hardware-level operations that are guaranteed to be executed without interruption from other threads

- **Memory Barriers**: Memory barriers (also known as memory fences) are hardware instructions that enforce ordering constraints on memory accesses by different threads or processes.
- Memory barriers help prevent race conditions and ensure that memory operations are visible in the expected order to all threads.
- **Interrupts and Interrupt Controllers:** Hardware interrupts and interrupt controllers play a role in synchronization by allowing the operating system to handle asynchronous events and preemptively switch between threads or processes.
- **Hardware Transactional Memory (HTM):** In some modern processors, HTM support allows programmers to specify regions of code as transactions that should execute atomically. If the hardware detects conflicts between transactions, it can automatically roll back

Semaphore

1. Synchronization tool that provides more sophisticated ways for process to synchronize their activities.
2. Semaphore ***S*** – integer variable
3. Can only be accessed via two indivisible (atomic) operations 1. wait() and 2. signal() Originally called P() and V().

- Definition of the **`wait() operation:`**
- Wait() operation is used by a thread or process to request access to a shared resource protected by a semaphore. When a thread invokes the Wait() operation on a semaphore, the following happens:

❑ If the semaphore's value is greater than zero, it is decremented by one, and the thread is allowed to proceed (i.e., it gains access to the resource).

❑ If the semaphore's value is already zero (indicating that the resource is currently in use by other threads or processes), the requesting thread is blocked or put into a wait state until the semaphore's value becomes greater than zero.

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

2. Signal() operation

- This operation is used by a thread or process to release a previously acquired resource protected by a semaphore.
- When a thread invokes the Signal() operation on a semaphore, the following happens:
  The semaphore's value is incremented by one, indicating that the resource is now available for other threads or processes to use.

Definition of the **signal() operation**

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

Counting semaphore – integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1
Same as a **mutex lock**
Can solve various synchronization problems
Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Can implement a counting semaphore ***S*** as a binary semaphore

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- It involves multiple processes or threads that need to share a finite-size buffer or queue for data exchange, while ensuring that they do not access the buffer in a way that leads to data corruption or conflicts.
- There is a shared buffer (or queue) of finite size, which can hold a maximum of N items.
- The main objectives in solving the Bounded-Buffer Problem are:

1. **Mutual Exclusion**: Ensuring that only one process can access the buffer at a time to prevent data corruption.
2. **Full Buffer Handling**: Producers should wait if the buffer is full, and consumers should wait if the buffer

```
# Shared variables
buffer = []  # Shared buffer
mutex = Semaphore(1)      # Mutual exclusion semaphore
empty = Semaphore(N)     # Semaphore to track empty slots in the buffer
full = Semaphore(0)           # Semaphore to track filled slots in the buffer

# Producer process
def producer():
    item = produce_item()
    empty.wait()  # Wait if buffer is full
    mutex.wait()  # Acquire the buffer lock
    buffer.append(item)
    mutex.signal()  # Release the buffer lock
    full.signal()   # Signal that a slot is filled

# Consumer process
def consumer():
    full.wait()    # Wait if buffer is empty
    mutex.wait()   # Acquire the buffer lock
    item = buffer.pop(0)
    mutex.signal()  # Release the buffer lock
    empty.signal()  # Signal that a slot is empty
    consume_item(item)

# Initialize producer and consumer processes
# Start them concurrently
```

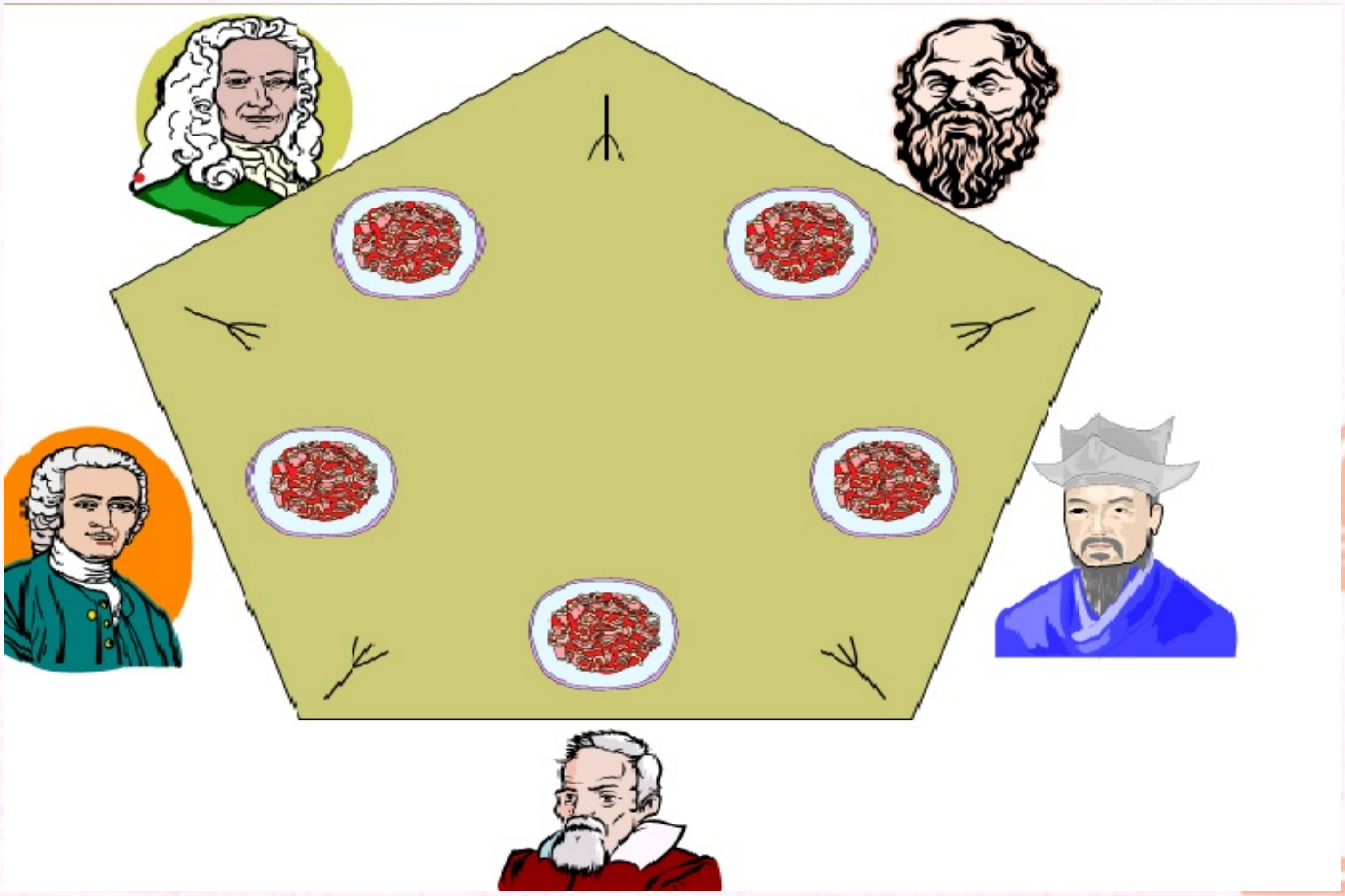
Readers and Writers Problem

- The Readers and Writers Problem is another classic synchronization problem in concurrent programming.
- It involves multiple processes (readers and writers) trying to access a shared resource (usually a data structure or database) in a coordinated manner.
- The problem is to ensure that multiple readers can access the resource concurrently for reading purposes, but only one writer can access it at a time for writing purposes.

- This prevents data corruption that could occur if a writer writes while a reader is reading.
- Here are the basic rules for the Readers and Writers Problem:

1. Multiple readers can read the shared resource simultaneously without any issues.
2. Only one writer can write to the resource at a time, and when a writer is writing, no other readers or writers should have access to the resource.

Dining a Philosopher Problem



1. The Dining Philosophers Problem is a classic synchronization and concurrency problem in computer science and operating systems.
2. **Problem Statement:-** Five philosophers sit at a round dining table, and each philosopher thinks and eats. There are five forks placed between them. To eat, a philosopher needs both the fork on their left and the fork on their right. Once a philosopher has both forks, they can eat for a while and then put the forks back on the table.
3. To solve this problem, various synchronization mechanisms can be employed, such as semaphores, mutexes, or other concurrency primitives.

The structure of Philosopher *i*:

```
do  {
     wait ( chopstick[i] );
     wait ( chopStick[ (i + 1) % 5] );
          //  eat
     signal ( chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );
          //  think
} while (TRUE);
```

Monitors in Process Synchronization

- Monitors are a synchronization tool used in process synchronization to manage access to shared resources and coordinate the actions of numerous threads or processes.
- Monitors provide an abstraction that simplifies the design of concurrent programs while ensuring data integrity and avoiding race conditions.
- Monitors provide a structured way to manage concurrent access to resources and ensure data

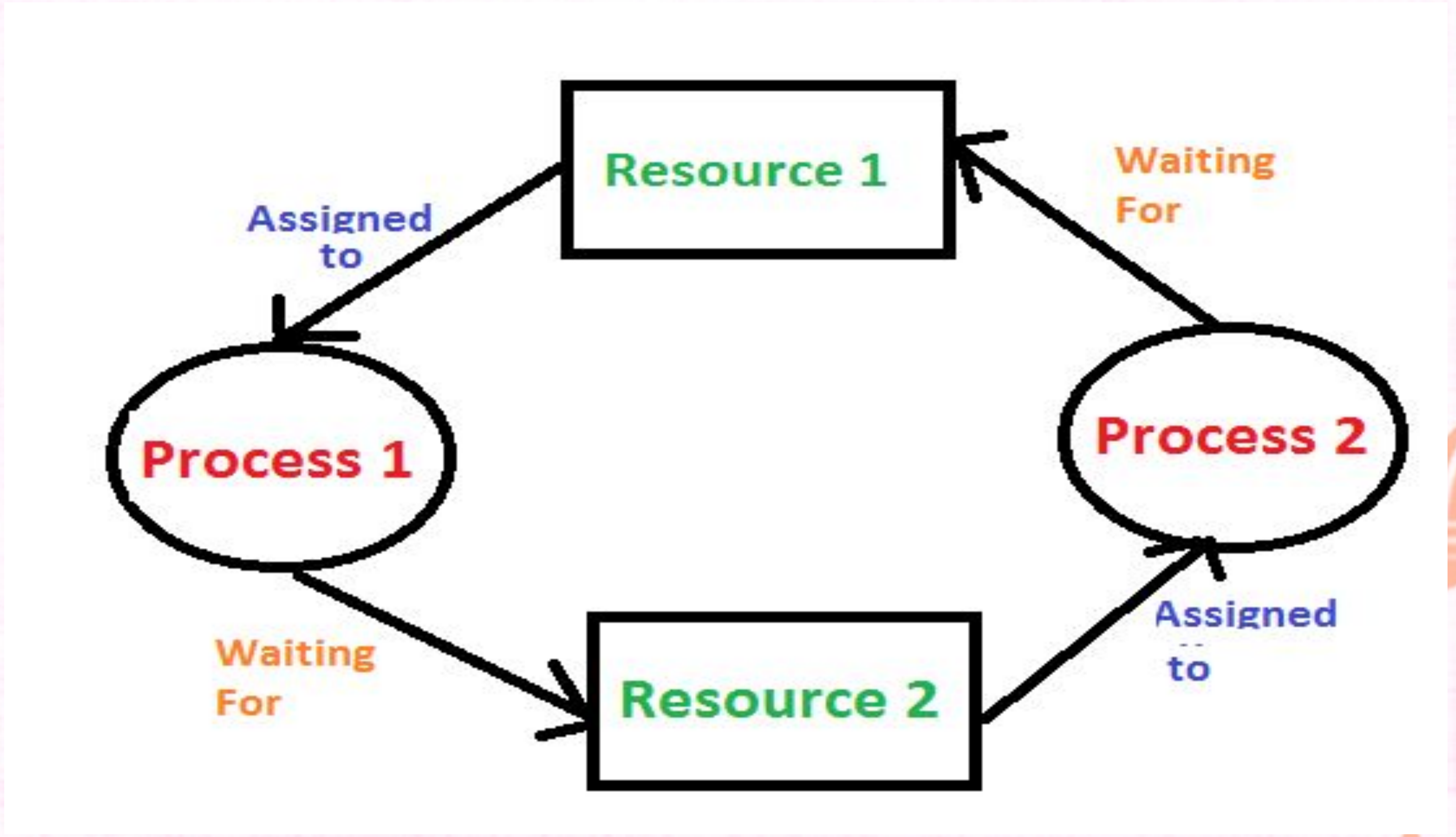
- Monitors provide a structured way to manage concurrent access to resources and ensure data integrity.
- An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

Syntax of Monitor:

Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
      Initialization code ( ….) { … }
                …
        }
}
```

DEADLOCKS


Resource 1
Assigned to
Waiting For
Process 1
Process 2
Waiting For
Resource 2
Assigned to

- A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
- E.g You can't get a job without experience; you can't get experience without a job.

Deadlock Characterization

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. MUTUAL EXCLUSION

- At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. HOLD AND WAIT.

- A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. NO PREEMPTION.

- Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. CIRCULAR WAIT.

- A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by Pi,……., P(i-1) is waiting for a resource held by Pn.

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
Hold and Wait – must guarantee that whenever a thread requests a resource, it does not hold any other resources

- Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
- Low resource utilization; starvation possible

No Preemption:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the thread is waiting
- Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait:

Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration

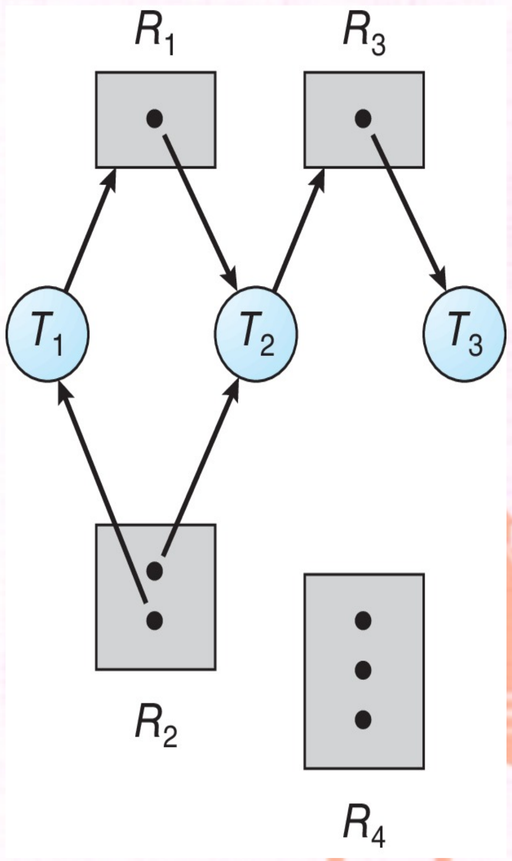
Resource-Allocation Graph

- A Resource-Allocation Graph (RAG) is a graphical representation used in deadlock detection algorithms to track resource allocation and resource requests among processes in a computer system.

1. **Processes**: Each process in the system is represented by a node in the graph.
2. **Resources**: Resources, such as CPU, memory, I/O devices, or any other shared resource, are also represented by nodes in the graph
3. **Edges**: The edges in the graph indicate relationships between processes and resources. There are two types of edges in a Resource-Allocation Graph:

    Request Edge (R-edge): An R-edge from a process node to a resource node represents a request by the process for that resource
    Assignment Edge (A-edge): An A-edge from a resource node to a process node indicates that the resource has been allocated to the

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3



The Banker's algorithm:

- The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in computer science and operating systems to ensure that processes (or tasks) can request and release resources in a way that avoids deadlock.
- The Banker's algorithm works by keeping track of the available resources and the maximum demand of each process.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [MAX] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [ALLOCATED] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [AVAILABLE] resource.

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available:** It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max**: It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation**: It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.

1. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i][j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work. Nedd[i][j] = Max[i][j] - Allocation[i][j].
2. **Finish**: It is the vector of the order m. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task

Safe State

- A safe state is a state in which the system can allocate resources to processes in such a way that no deadlock will occur, meaning all processes can eventually complete their execution and release their allocated resources.

- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <T1, T2, …, Tn> of ALL the threads in the systems such that for each Ti, the resources that Ti can still request can be satisfied by currently available resources + resources held by all the Tj, with j < I
- That is:

If Ti resource needs are not immediately available, then Ti can wait until all Tj have finished
When Tj is finished, Ti can obtain needed resources, execute, return allocated resources, and terminate
When Ti terminates, Ti +1 can obtain its needed resources, and so on

If a system is in safe state ⇒ no deadlocks

If a system is in unsafe state ⇒ possibility of deadlock

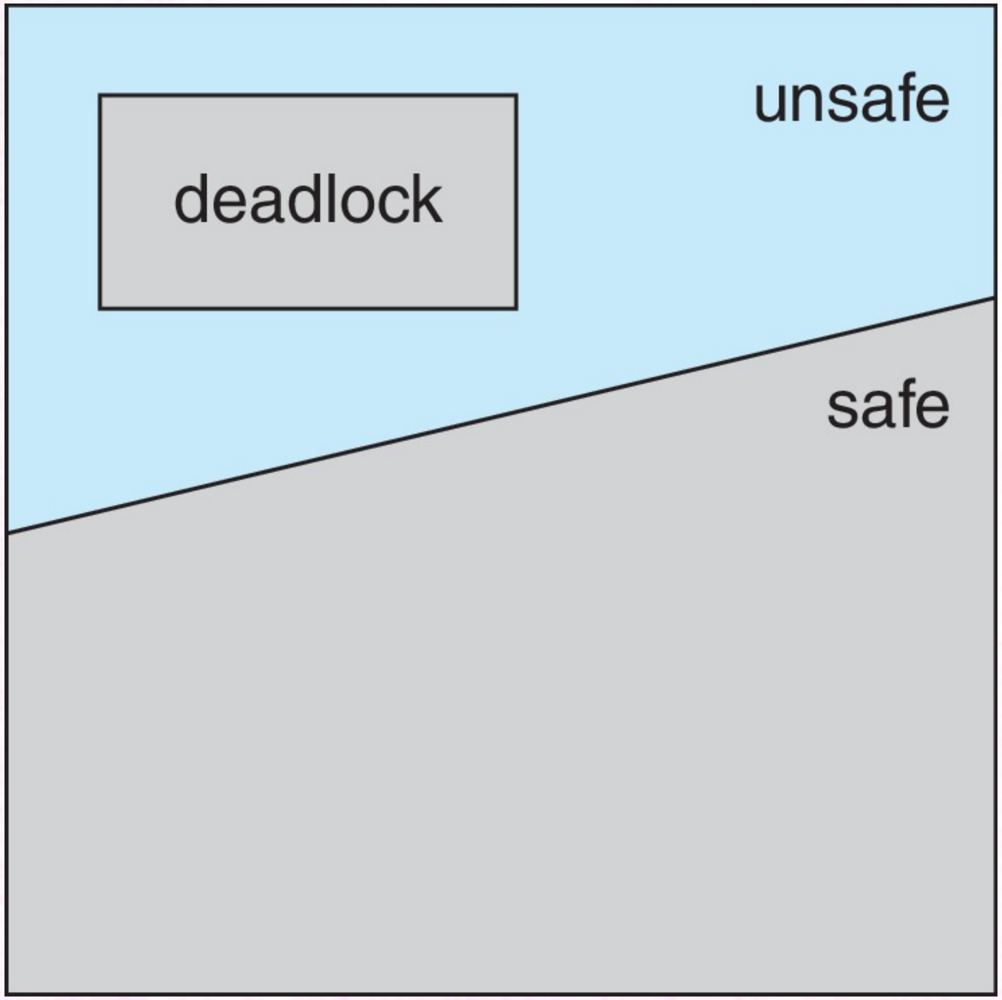
Avoidance ⇒ ensure that a system will never enter an unsafe state.

If a system is in safe state ⇒ no deadlocks

If a system is in unsafe state ⇒ possibility of deadlock

Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Deadlock Detection

- If a system has no deadlock prevention and no deadlock avoidance scheme, then it needs a deadlock detection scheme with recovery from deadlock capability.

Deadlock Detection Algorithm

Available [m]

Allocation [n,m] as in Banker's Algorithm Request [n,m]

indicates the current requests of each process.

Let Work and Finish be vectors of length m and n, as in the safety algorithm.

The algorithm is as follows:

1. Initialize Work = Available For i = 1 to n do If Allocation(i) = 0 then Finish[i] = true else Finish[i] = false
2. Search an i such that Finish[i] = false and Request(i) ≤ Work If no such i can be found, go to step 4.
3. For that i found in step 2 do: Work = Work + Allocation(i) Finish[i] = true Go to step 2.
4. If Finish[i] ≠ true for a some i then the system is in deadlock state else the system is safe

Recovery from Deadlock

- Abort all deadlocked threads
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

1. Priority of the thread
2. How long has the thread computed, and how much longer to completion
3. Resources that the thread has used
4. Resources that the thread needs to complete
5. How many threads will need to be terminated
6. Is the thread interactive or batch?

Questions

Q.1 What is deadlock? Explain necessary conditions for deadlock.

Q.2 Explain deadlock prevention.

Q.3 Explain Bankers algorithm with suitable example.

Q.4 Explain the concept of resource allocation graph.

Q.5 Explain deadlock avoidance.

Q.6 Explain the concept of safe state with suitable example.

Q.7 What is critical section problems and what are the possible solutions to the problems.

Q.8 Explain the readers and writers problem with the help of semaphore.

Q.9 Explain software approaches to solve synchronization problem.

Q.10 Explain hardware approaches to solve synchronization problem.

Questions

Q.11 Define process synchronization and explain petersons solution algorithm.

Q.12 What is semaphore. Differentiate between mutex and semaphore.

Q.13 Explain the producer and consumer problem with the help of semaphore.

Q.14 Explain deadlock detection and recovery.

Q.15 What is monitor? Explain with any suitable example.