This chapters examines **deadlock**, a problem arising in systems with multiple concurrently active asynchronous processes. There are three basic approaches to deal with deadlock: **prevention**, **avoidance**, and **detection**.

The Deadlock

- A deadlock occurs when two or more processes are waiting *indefinitely* for an event that can be caused only by one of the waiting processes. This occurs when there are a set of "blocked" processes in that each process holding a resource while waiting to acquire a resource held by another process in the set.
- If there is a deadlock, the following **four necessary** conditions must hold simultaneously: *mutual exclusion, hold and wait, no preemption,* and *circular wait*.
- **Resource-Allocation Graph** describes the current resource allocation of all processes. This, along with the future resource requests from all processes and the available resources that a system possess, represents the **resource allocation state** in the system. Deadlocks can be modeled with resource- allocation graphs, where a cycle potentially indicates deadlock. If each type of resource only has one instance, the Resource Allocation graph can be reduced to **Wait-For graph**.
- In general, we do not know the **sufficient** conditions for a deadlock to occur except in the special case where all resources have only *a single instance*, in which the circular wait (cycle in a *Resource-Allocation Graph* or in a *Wait-For graph*) is both a sufficient and necessary condition of deadlock.

Deadlock Prevention

- Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made. Such restrictions ensure that at least one of the necessary conditions cannot occur. However, in general this is either not feasible (for example mutual exclusion must hold for non-sharable resources, and preemption cannot be applied to resources such as locks and semaphores), or it can result in extremely low resource utilization (for instance, in order to avoid hold and wait, this may require each process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process is not holding any resource).

Deadlock Avoidance

- This requires additional prior knowledge on how each process will utilize system resources. The simple and most common method requires that each process declares the **maximum** number of resources of each type (i.e., the maximum demand) that the process may need throughout its entire execution.
- A **resource-allocation state** is defined by **three** pieces of information, the number of available resources (a vector of **m** types), the number of allocated resources (**NxM** matrix), and the maximum demands of all processes (**NxM** matrix). This allows us to derive the number of resources that processes need (might request) given the current allocation, i.e., max demand – allocation = need (**NxM** matrix)
- The deadlock-avoidance algorithm dynamically examines the resource-allocation

state upon each resource request by any process to ensure that there is no *circular-wait* in a system, so that a deadlock would never occur.

The Banker's Algorithm

- Deadlock can be avoided by using Banker's algorithm, which does not grant resources if such allocations would lead the system into an ***unsafe state*** where deadlock becomes possible.
- For a given resource allocation, a sequence <P_1, P_2, …, P_n> is said to be **safe** if for each P_i the resources that P_i requires (needs) can be satisfied by currently available resources in the system **plus** the resources held by all P_j, where j<i. This implies that the currently available resources can satisfy the need for P_1, which thus is impossible to be involved in any circular wait or a deadlock; P_2 need can be satisfied by the currently available resources and the resources held by P_1, and so on. The **safe sequence** in general is not unique.
- System is in **safe state** if there exists a **safe sequence** consisting of all processes.
- The Banker's algorithm verifies three conditions: (1) request <= need, otherwise the request violates the declared maximum; (2) request <= available, the system has sufficient resources to satisfy the request; (3) assuming (pretending) the request is granted, whether the resulting system is in a safe state or not.

Deadlock Detection

- A deadlock-detection algorithm evaluates processes and resources on a running system to determine if a set of processes is possibly in a deadlocked state.
- An algorithm also uses Banker's algorithm to periodically detect whether the current state is safe or not. The difference is to use ***request vector*** instead of ***need vector*** (i.e., need + allocation = maximum in the original Banker's algorithm).
- The primary question is when and how often the detection algorithm should be invoked, which has to take into account the possible frequency of deadlock occurrences and the number of processes involved in a deadlock.
- Noticing, however, Banker's algorithm (either in deadlock avoidance or deadlock detection algorithm) only verifies *at a particular time instant*, whether the system is safe or not. It does not make any assumption or does not even concern on how resources will be actually utilized and requested in the future. The complexity of using Banker's algorithm is that the system state changes very frequently with each resource request or release by any process.

Recovery from Deadlock

- **Process termination**: abort all deadlocked processes (great expenses) or abort one process at a time until the deadlock cycle is eliminated (which one to select).
- **Resource preemption**: how to *select a victim*, determine the *rollback* (return to some safe state and restart the process), and consider possible *starvation*.