This chapters examines **deadlock**, which is a problem arising in a system with multiple active asynchronous processes. There are three basic approaches to deal with deadlock: **prevention**, **avoidance**, and **detection** (although the terms *prevention* and *avoidance* are easy to confuse).

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 can occur when 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 in the system, the following **four necessary** conditions must hold simultaneously: *mutual exclusion, hold and wait, no preemption,* and *circular wait.* Therefore, to prevent deadlocks from happening, we need to ensure that one of the necessary conditions does not hold.
- **Resource-Allocation Graph** describes the current resource allocation to all processes. This, along with the future requests of resources from processes and the available resources that system possesses, represents the current resource allocation and requirements for the current processes.
- In general, we do not know the **sufficient** conditions for deadlocks to occur except in the case when all resources have only *one instance*, in which the circular wait (cycle in the Resource-Allocation Graph), will lead to the deadlock.

Deadlock Avoidance

- This requires that the OS has prior knowledge on how each process will utilize system resources. The simple and most common model requires that each process declare the **maximum** number of resources of each type (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)
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition, such that a deadlock would never occur.

The Banker's Algorithm

- For a given resource allocation graph, the sequence <P1, P2, …, Pn> is said to *be safe* if for each Pi, the resources that Pi requires (maximum need) can be satisfied by currently available resources in the system plus the resources held by all the Pj, where j<i. This implies that the currently available resources can satisfy the need for P1, P2 can be satisfied by the currently available resources and the resources hold by P1, and so on and so forth. The safe sequence is not unique.
- System is in *safe state* if there exists a safe sequence containing all processes.
- The banker's algorithm checks to see if the system is in a safe state.

Deadlock Detection

- An algorithm similar to the banker's algorithm can be used to find out 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 frequency of deadlock occurrence and the number of processes involved in a deadlock.

Recovery from Deadlock

- **Process termination**: abort all processes or abort one process (how 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*.