❖ Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

- Ensure that the system will ***never*** enter a deadlock state (**Deadlock prevention** and **Deadlock avoidance**).
- Allow the system to enter a deadlock state and then recover (**Deadlock detection** and **Recovery**).
- Ignore the problem and pretend that deadlocks never occur in the system. Used by most operating systems, including UNIX and Windows.

❖ **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.

❖ **Deadlock Avoidance:** Requires that the system has some additional ***a priori*** information available.

- Requires that each process declares the ***maximum number*** of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition.
- Resource-allocation ***state*** is defined by the number of **available** and **allocated** resources, and the **maximum** demands of the processes.

❖ Safe State

A state is ***safe*** if the system can allocate resources to each process (up to its maximum) and still avoid a deadlock. That is, a system is in a safe state only if there exists a **safe sequence**.

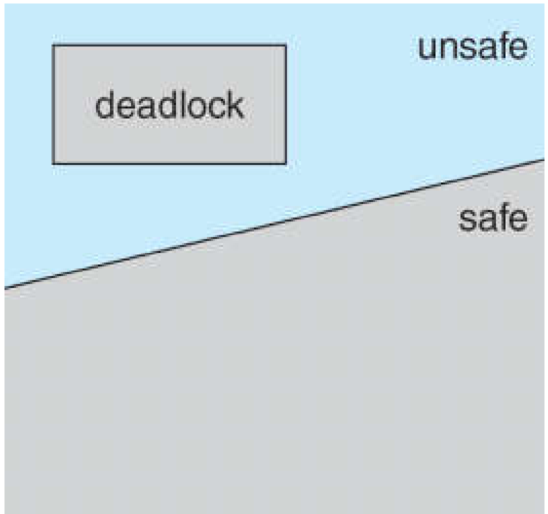
- System is in **safe state** if there exists a sequence $< P_1 , P_2 , \ldots, P_n >$ of ALL the processes in the system such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

❖ **Basic Facts**

- 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, and deadlocked state spaces.

Example: consider a system with 12 magnetic tape drives and 3 processes: *P*0, *P*1, and *P*2.
Suppose that, **at time t_0**

Process	Maximum Needs	Current Needs
*P*0	10	5
*P*1	4	2
*P*2	9	2

- There are 3 free tape drives, the system is in a safe state.
- The sequence <*P*1, *P*0, *P*2> satisfies the safety condition.
- *P*1 can allocate all its tape drives and return them; 5 available tape drives.
- *P*0 can get all its tape drives and return them; 10 available tape drives.
- Finally, *P*2 can get all its tape drives and return them; 12 tape drives available.

Suppose that, **at time t_1**, *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state.

- Only *P*1 can allocate all its drives. After returning them the system will have only 4 available tape drives.
- *P*0 allocates 5 tape drives but has a maximum of 10, it will have to wait.
- *P*2 may request 6 additional tape drives and have to wait, resulting in a deadlock.
- Our mistake was in granting the request from *P*2 for one more tape drive.
- *P*2 should wait until either of the other processes had finished and released its resources, in order to avoided the deadlock.

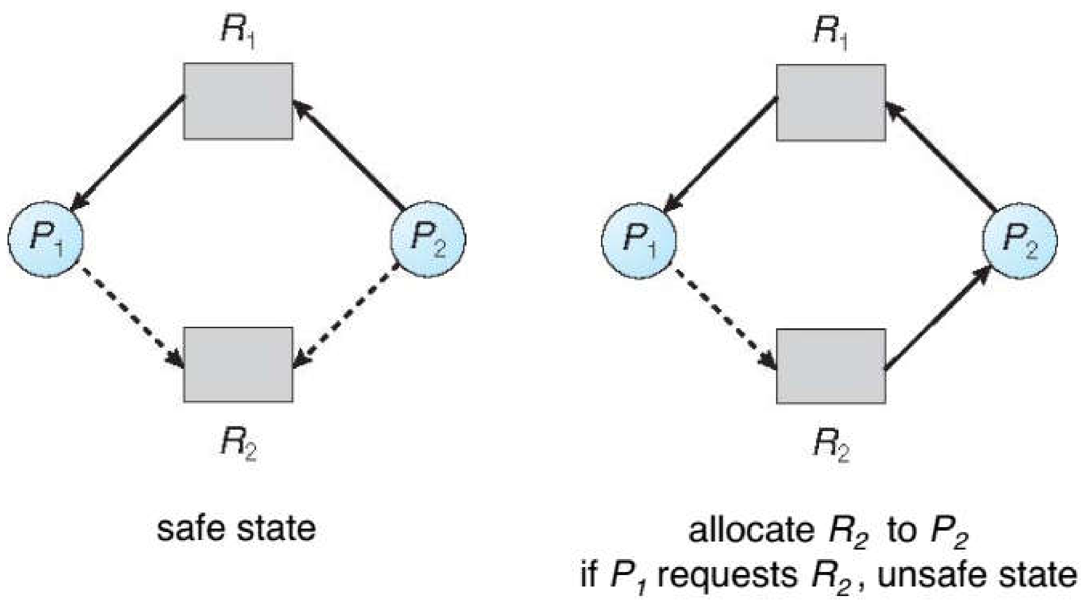
❖ Avoidance Algorithm

- Single instance of a resource type: Use a resource-allocation graph.
- Multiple instances of a resource type: Use the banker's algorithm.

❖ Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicates that P_j ***may*** request R_j; represented by a dashed line.
- **Claim edge** is converted to ***request*** edge when a process requests a resource.
- Request edge is converted to an ***assignment*** edge when a resource is allocated to process.
- When a resource is released by a process, assignment edge is reconverted to a claim edge.
- Resources must be claimed *a* ***priori*** in system (from the start).

Example: Suppose that P_i requests R_j. The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource-allocation graph.



Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

❖ Banker's Algorithm

The name of **banker's algorithm** was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

- It is applicable for a system with ***multiple*** instances of each resource type.

- Each process must *a* ***priori*** claim maximum number of instances of each resource type. This number may not exceed the total number of resources in the system.
- When a process requests a resource, it may have to wait until some other process releases enough resources.
- When a process gets all its resources, it must return them in a finite amount of time.

❖ Data Structures for the Banker's Algorithm

Let n = number of processes and m = number of resource types:

• **Available**. A vector of length m indicates the number of available resources of each type. If ***Available***$[j] = k$, then k instances of resource type R_j are available.

• **Max**. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***$[i,j] = k$, then P_i may request at most k instances of R_j.

• **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***$[i,j]=k$, then P_i is currently allocated k instances of R_j.

• **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***$[i,j]= k$, then P_i may need k more instances of R_j to complete its task.

Note: ***Need***$[i,j]$ = ***Max***$[i,j]$ - ***Allocation***$[i,j]$.

❖ Safety Algorithm

The algorithm is developed to find out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize:

   Work = ***Available***

   Finish$[i]$ = ***false*** for $i = 0, 1, ..., n - 1$

2. Find an i such that both:

   a. ***Finish***$[i]$ == ***false***

   b. ***Need***$_i \leq$ ***Work*** (P_i needs less resources than still available)

   If no such i exists, go to step 4.

3. ***Work*** = ***Work*** + ***Allocation***$_i$ (release resources of P_i back into available)

   Finish$[i]$ = ***true***

   Go to step 2.

4. If ***Finish***$[i]$ == ***true*** for all i, then the system is in a safe state.