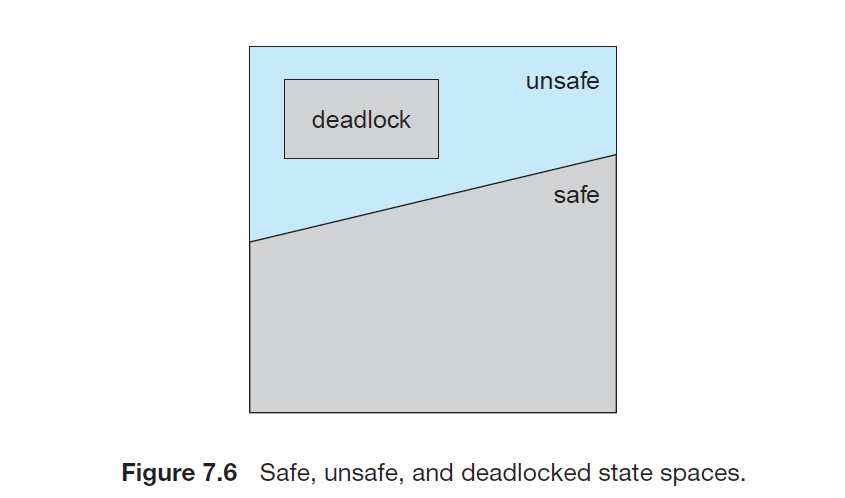
**DEADLOCK AVOIDANCE**

For avoiding deadlocks is to require additional information about how resources are to be requested. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. That means, in deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

**Safe State and Unsafe State:**

A state is safe if the system can allocate resources to each process in some order (up to its maximum requirement) while avoiding a deadlock. Formally, a system is in a safe state only, if there exists a safe sequence. So a safe state is not a deadlocked state and conversely a deadlocked state is an unsafe state.

If the operating system is unable to prevent processes from requesting resources, resulting in Deadlock, then the system is said to be in an Unsafe State. Not all unsafe states are deadlocks, however. Unsafe State may lead to a deadlock.



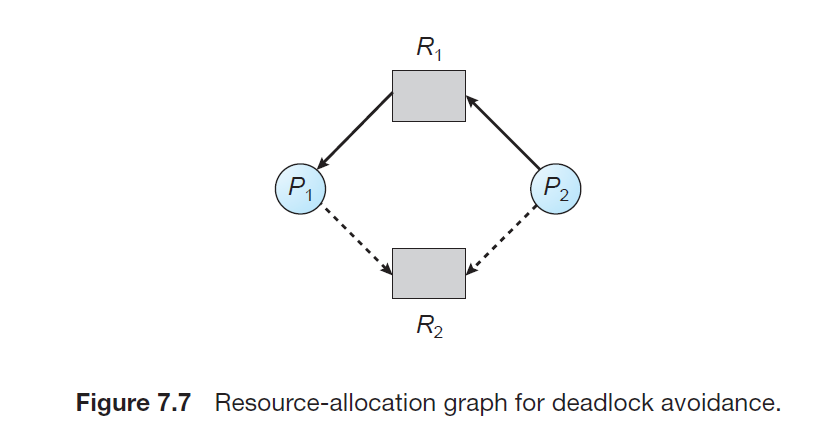
Deadlock Avoidance can be solved by two different algorithms:

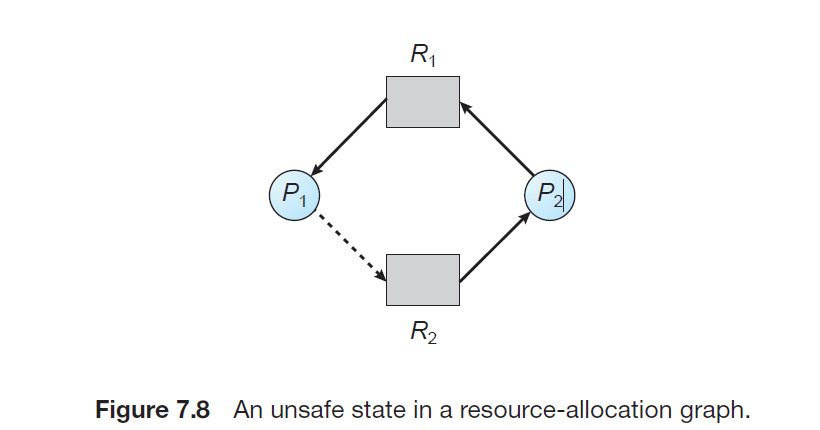
**1)Resource allocation Graph algorithm** 2)**Banker’s Algorithm**

**1)Resource-Allocation-Graph Algorithm:**

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge. A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj.

Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied.





In the above diagram, Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

**2)Banker’s Algorithm:**

The **Bankers’ Algorithm** is a resource allocation and deadlock avoidance algorithm that examines all resource requests made by systems. It checks for the safe state and makes the request if the system remains in the safe state after request approval. If there is no safe state, the request is denied.

###### Inputs required for the Bankers’ Algorithm:

1. Maximum need or resources required by each process. (Need)
2. The resources currently allocated by each process. (Allocation)
3. Free resources available in the system. (Available)

###### Request for the resource will only be granted:

1. If the request made by the process is </= the freely available resource in the system
2. If the request made by the process is </= maximum amount of resources required for the process

**1)Safety Algorithm:**

We can now present the algorithm for finding 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 and Finish[i] = false for i = 0, 1, ..., n − 1.

2. Find an index i such that both

a. Finish[i] == false

b. Needi ≤ Work

If no such i exists, go to step 4.

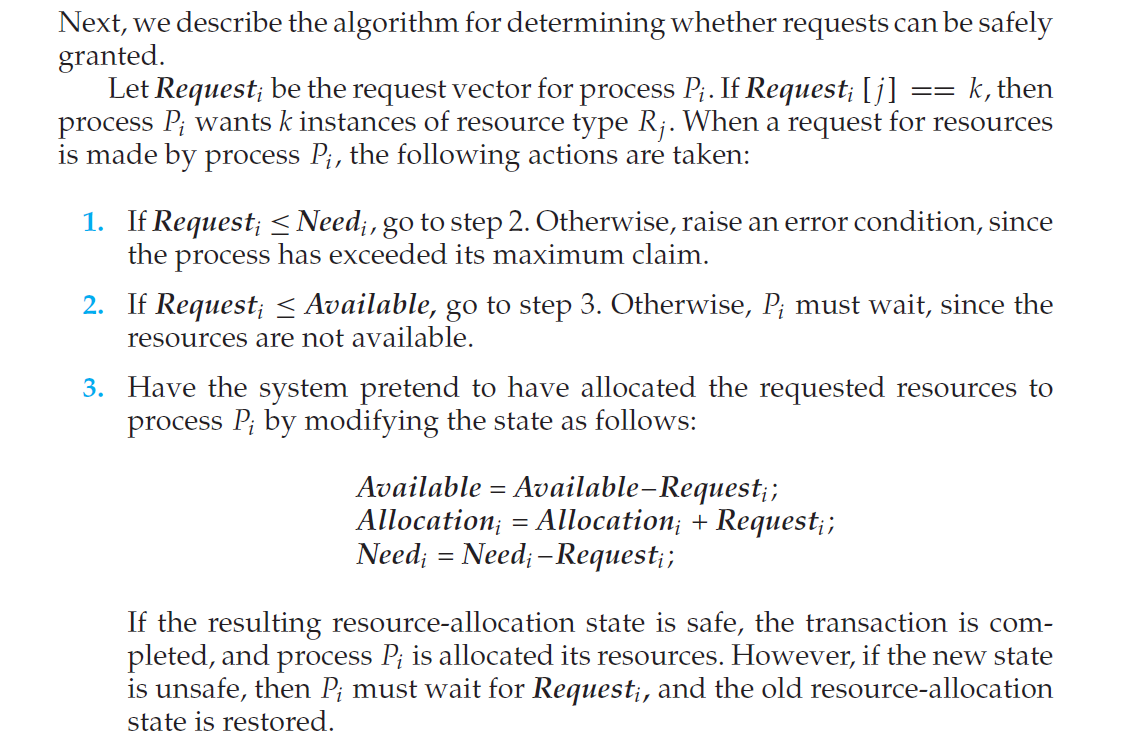
3. Work = Work + Allocationi

Finish[i] = true

Go to step 2.

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

**2)Resource-request algorithm:**

****