as many as four tape drives, and process $P2$ may need up to nine tape drives. Suppose that, at time $t0$, process $P0$ is holding five tape drives, process $P1$ is holding two tape drives, and process $P2$ is holding two tape drives. (Thus, there are three free tape drives.)

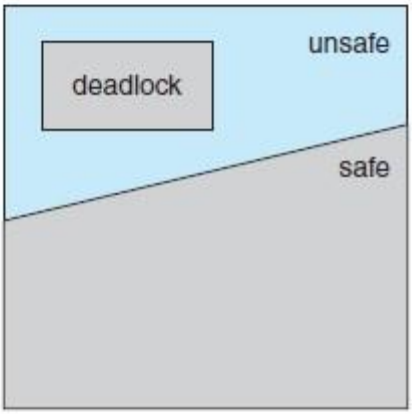

Figure 6-4 Safe, unsafe, and deadlocked state spaces

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time $t0$, the system is in a safe state. The sequence <$P1$, $P0$, $P2$> satisfies the safety condition. Process $P1$ can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process $P0$ can get all its tape drives and return them (the system will then have ten available tape drives); and finally process $P2$ can get all its tape drives and return them (the system will then have all twelve tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time $t1$, process $P2$ requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process $P1$ can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process $P0$ is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process $P2$ may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process $P2$ for one more tape drive. If we had made $P2$ wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state. In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

6.5.2. 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 that has been defined previously 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 $Pi \rightarrow Rj$ 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 $Pi \rightarrow Rj$ is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.

Note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $Pi \rightarrow Rj$ to be added to the graph only if all the edges associated with process Pi are claim edges.

Now suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge $Pi \rightarrow Rj$ to an assignment edge $Rj \rightarrow Pi$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of $n2$ operations, where n is the number of processes in the system.

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. To illustrate this algorithm, we consider the resource-allocation graph of Figure 6.5. 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 (Figure 6.6). A cycle, as mentioned, 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.

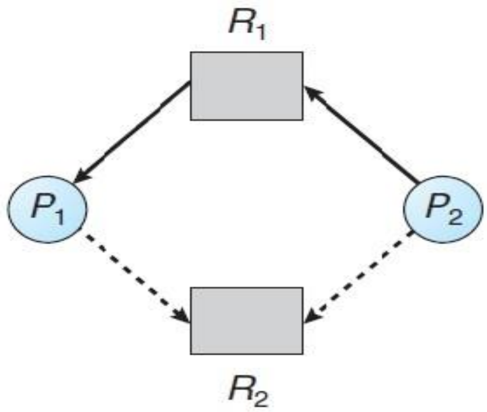

Figure 6-5 Resource-allocation graph for deadlock avoidance

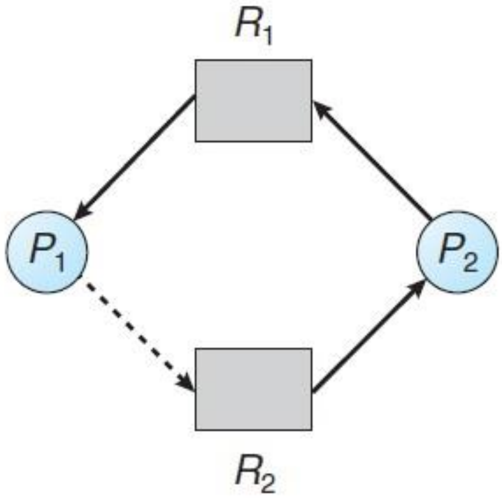

Figure 6-6 An unsafe state in a resource-allocation graph

6.5.3. Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm.** The name 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.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

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

• **Max**. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***[i][j] equals k, then process Pi may request at most k instances of resource type Rj .

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

• **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that ***Need***[i][j] equals***Max***[i][j]

− ***Allocation***[i][j].

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$.