floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release. The process releases the resource.

For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example files). However, other types of events may result in deadlocks.

To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CDRW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 5 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur.

6.2. Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

6.2.1. Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0. We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

6.2.2. Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: $P = \{P1, P2, ..., Pn\}$, the set consisting of all the active processes in the system, and $R = \{R1, R2, ..., Rm\}$, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by $Pi \rightarrow Rj$; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by $Rj \rightarrow Pi$; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge $Pi \rightarrow Rj$ is called a **request edge**; a directed edge $Rj \rightarrow Pi$ is called an **assignment edge**.

Pictorially, we represent each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle Rj , whereas an assignment edge must also designate one of the dots in the rectangle.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted. The resource-allocation graph shown in Figure 6.1 depicts the following situation.

- The sets P, R, and E:
  - $P = \{P1, P2, P3\}$
  - $R = \{R1, R2, R3, R4\}$
  - $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$
- Resource instances:

◦ One instance of resource type $R1$

◦ Two instances of resource type $R2$

◦ One instance of resource type $R3$

◦ Three instances of resource type $R4$

• Process states:

◦ Process $P1$ is holding an instance of resource type $R2$ and is waiting for an instance of resource type $R1$.

◦ Process $P2$ is holding an instance of $R1$ and an instance of $R2$ and is waiting for an instance of $R3$.

◦ Process $P3$ is holding an instance of $R3$.

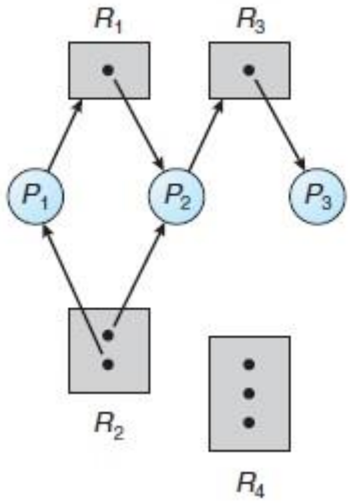

Figure 6-1 Resource-allocation graph

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this