case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock. To illustrate this concept, we return to the resource-allocation graph depicted in Figure 6.1. Suppose that process $P3$ requests an instance of resource type $R2$. Since no resource instance is currently available, we add a request edge $P3 \rightarrow R2$ to the graph (Figure 6.2). At this point, two minimal cycles exist in the system:

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

Processes $P1$, $P2$, and $P3$ are deadlocked. Process $P2$ is waiting for the resource $R3$, which is held by process $P3$. Process $P3$ is waiting for either process $P1$ or process $P2$ to release resource $R2$. In addition, process $P1$ is waiting for process $P2$ to release resource $R1$.

Now consider the resource-allocation graph in Figure 6.3. In this example, we also have a cycle:

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

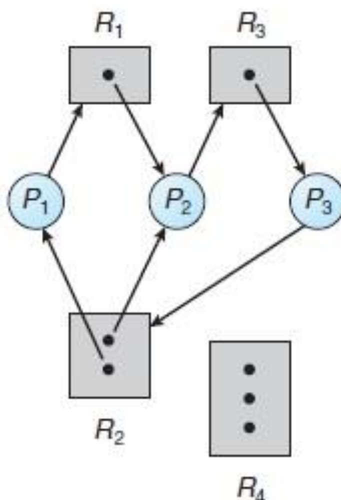


**Figure 6-2 Resource-allocation graph with a deadlock**

However, there is no deadlock. Observe that process $P4$ may release its instance of resource type $R2$. That resource can then be allocated to $P3$, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

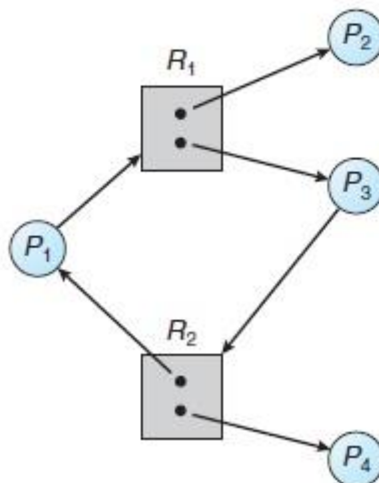


**Figure 6-3 Resource-allocation graph with a cycle but no deadlock**

## 6.3. Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide

for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per year), the extra expense of the other methods may not seem worthwhile. In addition, methods used to recover from other conditions may be put to use to recover from deadlock. In some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

## 6.4. 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. We elaborate on this approach by examining each of the four necessary conditions separately.

### 6.4.1. Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

### 6.4.2. Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and