Deadlocks

- In a multiprogramming environment, several processes may compete for a infinite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.
- A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.
- CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances.

- If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.
- 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.

- 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.
- Consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW 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.

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

- 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 nonsharable 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 {P0 , P1 , ..., Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1 , P1 is waiting for a resource held by P2 , ..., Pn−1 is waiting for a resource held by Pn , and Pn is waiting for a resource held by P0 .

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→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 → Pi ; it signifies that an instance of resource type Rj has been allocated to process Pi .
- A directed edge Pi → Rj is called a **request edge**; a directed edge Rj → Pi is called an **assignment edge**.

- 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.
- 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.

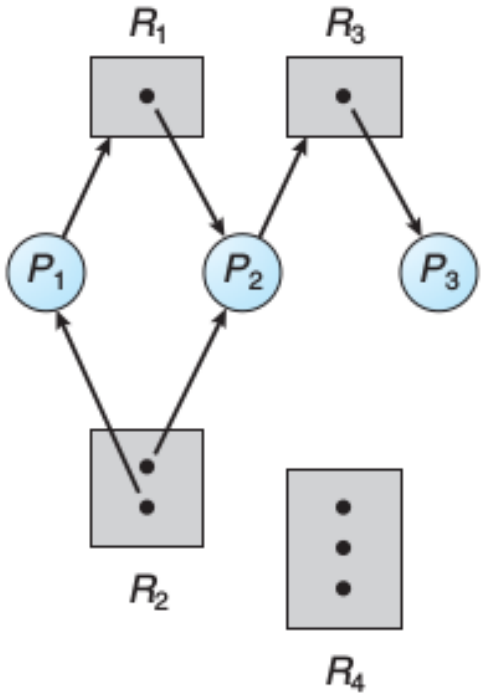


Figure 7.1 Resource-allocation graph.

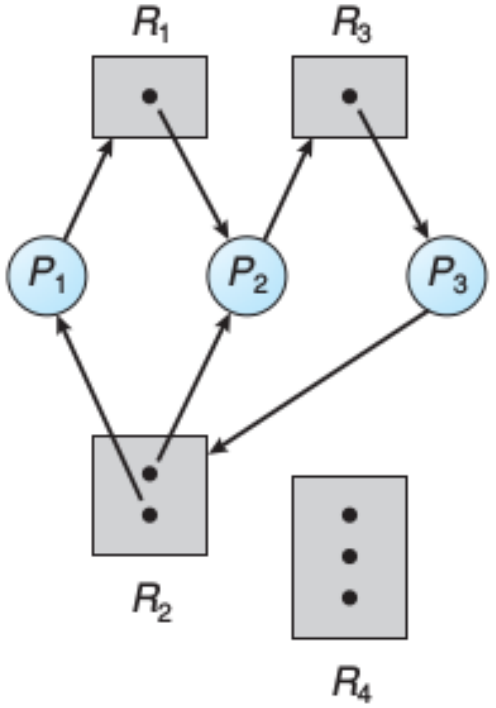
- P = {P1,P2,P3}
- R = {R1,R2,R3,R4}
- E = {P1 -> R1, P2 -> R3, R1 -> P2, R3 -> P3, R2 -> P1, R2 -> P2}
- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

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 .

- **If the RAG 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 case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
- Suppose that process P3 requests an instance of resource type R2 . Since no resource instance is currently available, we add a request edge P3 → R2 to the graph



Resource-allocation graph with a deadlock.

At this point, two minimal cycles exist in the system:

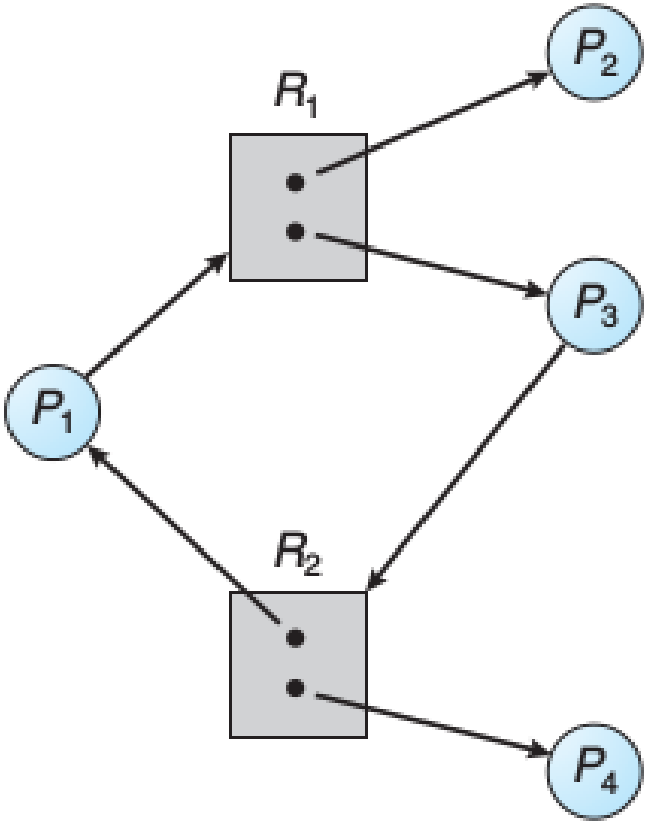
- P1 → R1 → P2 → R3 → P3 → R2 → P1
- P2 → R3 → P3 → R2 → P2

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 .
- In the following example, we also have a cycle:

  P1 → R1 → P3 → R2 → P1

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



Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

- 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.
- Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

- 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.