Deadlock

4

4.1 Introduction

Deadlock is the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. Unlike other aspects of process management, deadlock is a problem that defies efficient solution in the general case. If deadlock is ignored in the design stage, it must later be detected by some means, and then processes must be terminated and restarted to recover from it. Alternatively, process behavior can be constrained (or systems can be restricted) at the design stage so deadlock does not occur. Resource utilization must suffer under these regimens.

In this chapter we follow the development found in Holt's thesis [Holt, 1971] and a later tutorial [Holt, 1972]. First, we introduce the deadlock problem. Second, we introduce some definitions and results from graph theory that allow a model of deadlock to be constructed in Section 4.4. This model is useful because the only elements are processes and resources. Processes compete for resources or communicate by producing and consuming resources. The abstract nature of the model means that results are widely applicable and do not depend on the characteristics of particular operating systems. This first, completely general, model captures the nature of deadlock but, unfortunately, fails to help us develop algorithms for coping with the problem. Section 4.5 considers some restrictions on the general model. These models depict frequently occurring special situations wherein efficient and useful approaches to the deadlock problem are available. Recovery from deadlock is considered in Section 4.6. Deadlock can be prevented by system design; this is the subject of Section 4.7. Section 4.8 considers how the methods considered in this chapter are combined in the design of complete operating systems.

4.2 The Deadlock Problem

In this section we define terms used in the rest of the chapter, give examples of deadlock, explain the characteristics of resource types, and introduce policies for dealing with deadlock.

4.2.1 Definition of Deadlock

A computer system may be abstractly represented by a pair of sets (Σ, Π), where

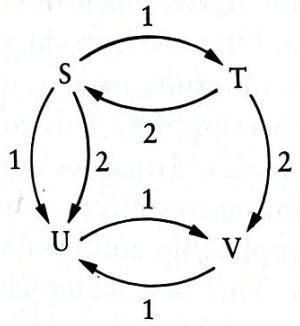
Σ = {All possible allocation states of all system resources}
Π = {Processes}

Each element in Σ represents one possible state in the distribution of the resources. Each process in Π is a function that, for each system state in Σ, maps to another set of states (possibly empty).

For example, let $\Sigma = \{S, T, U, V\}$ and $\Pi = \{P_1, P_2\}$. There are only four possible system states; actions by the two processes cause the system to change from state to state. Suppose the possible actions by the two processes are:

$$P_1(S) = \{T, U\} \quad P_2(S) = \{U\}$$
$$P_1(T) = \varnothing \quad P_2(T) = \{S, V\}$$
$$P_1(U) = \{V\} \quad P_2(U) = \varnothing$$
$$P_1(V) = \{U\} \quad P_2(V) = \varnothing$$

where, for example, $P_1(S) = \{T, U\}$ means that when P_1 is in state S, it may operate to change the system to state T or state U. When the range is $\varnothing$, the process may not operate when the system is in the given state. One may show the states graphically by using nodes for the possible states and arcs for the possible state changes. The arcs are labeled with the process that can effect that state change. The example above is also defined by:



An *operation* by process i changes the system state from, say, S to T. We abbreviate this by writing S→i→T. In the figure, S→1→U, T→2→V, and so forth. If a sequence of operations by processes i, j, . . . , k is possible (S→i→T, T→j→U, . . . , V→k→W), we abbreviate sequences by S→*→W.

With this minimal setting, we can define some terms relating to deadlock in an unambiguous way.

- A process P_i is *blocked in state S* if there exists no T so that S→i→T. In the figure, P_1 is blocked in state T because there is no arc labeled 1 starting at node T.
- Process P_i is *deadlocked in state S* if P_i is blocked in S and, for all states T with S→*→T, P_i is blocked in T. No matter how other processes change the system state, there will never be an opportunity for P_i to perform an operation. In the figure, P_2 is deadlocked in states U and V. P_1 is not deadlocked in T because, for example, T→2→S unblocks P_1.
- If there is a process P_i deadlocked in S, then S is a *deadlock state*.
- If all processes P_i are deadlocked in S, then S is a *total deadlock state*. In the figure, there are no total deadlock states, but U and V are deadlock states.

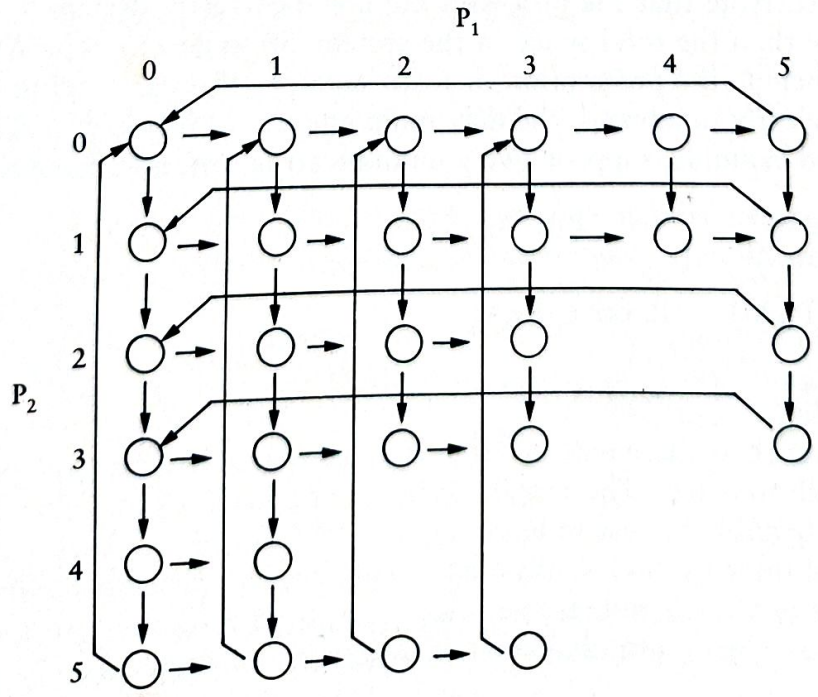
- State S is *secure* if S is not a deadlock state and, for any state T reachable from S ($S \rightarrow^* \rightarrow T$), T is not a deadlock state.

Here is an example that will help make these abstract ideas more concrete: Two processes compete for exclusive access to a disk file D and the only tape drive T in a system. Both cyclic programs perform these operations:

```
                P1:                          P2:
0:          ...                  0:      ...
            while (true) do              while (true) do
1:            Request(D);        1:      Request(T);
2:          ...                  2:      ...
3:            Request(T);        3:      Request(D);
4:          ...                  4:      ...
            Release(T);                  Release(D);
5:          ...                  5:      ...
            Release(D);                  Release(T);
            endwhile                     endwhile
```

As each process executes its cycle, it may be in one of six states relative to the ownership of system resources, as seen here:

	P_1:		P_2:
0:	Holds no resources	0:	Holds no resources
1:	Holds none, requests D	1:	Holds none, requests T
2:	Holds D	2:	Holds T
3:	Holds D, requests T	3:	Holds T, requests D
4:	Holds D and T	4:	Holds T and D
5:	Holds D, T released	5:	Holds T, D released



A state of this system S_{ij} reflects that P_1 is in state i while P_2 is in state j. The possible system states are found in the preceding graph. Horizontal arrows (left or right) are state changes due to actions by P_1, and vertical arrows (up or down) are state changes due to the action of P_2.

Both P_1 and P_2 are deadlocked in S_{33}: It is a total deadlock state. States S_{23} and S_{32} are deadlock states because P_2 and P_1 are deadlocked in the states, respectively. P_1 is blocked in states S_{14}, S_{32}, and S_{35}. P_2 is blocked in states S_{23}, S_{41}, and S_{53}. There are no secure states.

4.2.2 Examples of Deadlock

Deadlock may involve any number of processes and resources in simple or complicated ways.

As a first example, consider the two processes competing for disk file D and tape drive T in the example of the preceding section. Deadlock occurs if each process holds one resource and requests the other. Although this example can be regarded as a system design error, its occurrence in practice is real and often embedded in complicated program logic to the extent that a priori detection is difficult, if not impossible. Strategies to cope with this type of problem include imposing constraints on system design so that certain resources are requested in a particular order.

For a second example, suppose the main memory space required for activation records of processes is dynamically allocated. Suppose the total space consists of 20K bytes and two processes require memory in the following way:

P_1:	P_2:
Request 8K bytes	Request 7K bytes
Request 6K bytes	Request 8K bytes

As in the previous example, deadlock occurs if both processes progress to their second request. Note that the processes are not incorrectly designed since neither requests more than the total space in the system. Strategies to cope with this type of problem include the preemption of main memory through paging or requiring processes to specify in advance the maximum amount of memory space needed.

As a third example, suppose two communicating processes have the structure:

P_1:	P_2:
. . .	. . .
Receive (P_2,M)	Receive (P_1,M)
. . .	. . .
Send (P_2,M′)	Send (P_1,M′)

Design errors such as these may occur at isolated places in very large programs and may be difficult to detect. The actual occurrence of deadlock may be infrequent and may occur only after the system has been in service for many years.

In each of these examples, deadlock occurs because processes request resources held by other processes and, at the same time, hold resources requested by these same processes. This is a fundamental characteristic of deadlock.

Deadlock is similar to *starvation*, since each of these involves one or more processes that are permanently blocked and waiting for the availability of resources. The two, however, are distinctly different phenomena. A deadlocked process waits for resources (held by another process) that will never be released. Starvation occurs when some process waits for resources that periodically become available but are never allocated to that process due to some scheduling policy. An example of starvation is a process waiting for the simultaneous availability of two tape drives. An endless sequence of processes request, are allocated, and release tape drives such that either two drives are never simultaneously available or, if they are available, they are allocated to a higher priority process.

4.2.3 Resource Types

Resources can be divided into two classes: *reusable* and *consumable*. Each class has distinct properties that are reflected in the various strategies designed to deal with the deadlock problem.

A reusable (serially reusable) resource is characterized by the following properties:

- There is a fixed total inventory. Additional units are neither created nor destroyed.
- Units are requested and acquired by processes from a pool of available units and, after use, are returned to the pool for use by other processes.

Examples of serially reusable resources are processors, I/O channels, main and secondary memory, devices, channels, busses, and information such as files, databases and mutual exclusion semaphores. The first two examples of the last section illustrate deadlock involving serially reusable resources.

A consumable resource type is characterized by the following properties:

- There is no fixed total number of units. Units may be created (produced or released) or acquired (consumed) by processes.
- An unblocked producer of the resource may release any number of units. These units then become immediately available to consumers of the resource.
- An acquired unit ceases to exist.

Examples of consumable resources are interrupts and signals, messages, and information in I/O buffers. The third example in the previous section illustrates deadlock involving messages.

In general, deadlock may involve any combination of classes of resources, both reusable and consumable. The classes of resources present in any system or subsystem affects the manner in which the deadlock problem can be handled. This will become clear in subsequent sections.

4.2.4 Deadlock Policies

Methods for coping with deadlock fall into three categories. In this chapter we will see examples from each class. The first policy is *detection and recovery*. Here no

action is taken to keep deadlock from occurring. Rather, system events may trigger the execution of a detection algorithm. When the group of deadlocked processes is identified, some of them must be terminated (or rolled back to an earlier state if checkpoint information is available) in order to break the deadlock. This approach is satisfactory in some computer systems if the frequency of deadlock is low and the cost of recovery is reasonable because the utilization of system resources is not degraded during normal operation. Efficient detection algorithms will be discussed in Sections 4.5.1 and 4.5.3.

A second class of deadlock policies is *prevention*. Here the system design prevents entry into a state from which future deadlock is inevitable. This is accomplished by denying at least one of the four following conditions, all of which are necessary for deadlock to occur:

1. *Mutual Exclusion:* Processes hold resources exclusively, making them unavailable to other processes.
2. *Nonpreemption:* Resources are not taken away from a process holding them; only processes can release resources they hold.
3. *Resource Waiting:* Processes that request unavailable units of resources block until they become available.
4. *Partial Allocation:* Processes may hold some resources when they request additional units of the same or other resources.

Deadlock is prevented by designing the resource management sections of an operating system so that one of the conditions cannot occur. Denying any condition inevitably degrades utilization of system resources, but is appropriate in systems for which deadlock carries a heavy penalty (real-time systems controlling chemical or nuclear processes, or systems that monitor or control hospital intensive care units, for example). We will see examples of prevention policies in Sections 4.4 and 4.7.

Avoidance is the third type of deadlock policy. This refers to methods that rely on some knowledge of future process behavior to constrain the pattern of resource allocation. Once again a degradation in resource utilization is inevitable. Often, a subset of resources for which deadlock is especially expensive is managed with an avoidance policy, while detection and recovery suffices for other resources in the same system. Future information may be reasonably easy for a system to deduce or for a user to supply in some instances, but may be impossible to deduce or obtain from a user in other cases. A generalization of a method you may already know (the Banker's Algorithm) is found in Section 4.5.3.

4.3 Concepts from Graph Theory

Before we can introduce the model, we must discuss a few terms and results from graph theory.

- A *directed graph* (digraph) is a pair (N,E), where N is a nonempty set of nodes and E is a set of edges. Each edge is an ordered pair of nodes.

- A *bipartite graph* is one in which all the nodes in N divide into two disjoint subsets π and ρ so that all edges consist of a node from each subset. An interesting result of bipartite graphs will help us determine the efficiency of a detection algorithm later; that is, if the subsets contain m and n nodes, respectively, then the maximum number of nonidentical edges is 2mn.
- Node z is a *sink* if there are no edges (z,b).
- Node z is an *isolated node* if there are no edges (z,b) or (b,z).
- A *path* is a sequence (a,b,c, . . . , y, z) of at least two nodes for which (a,b), (b,c), . . . , (y,z) are edges.
- A *cycle* is a path with the same first and last node.
- The *reachable set* of a node z is the set of all nodes to which there is a path beginning with z. The reachable set of z may contain z itself if there is a cycle.
- A *knot* K is a nonempty set of nodes with the property that, for each node z in K, the reachable set of z is exactly the knot K.

In Figure 4.1 we see a bipartite graph with nodes N = {a,b,c,d} and edges E = {(a,b), (a,b), (a,c), (c,d), (d,c)}. All edges connect between nodes in the two sets {a,d} and {b,c}. Node b is a sink, (c,d,c) is a cycle, and {c,d} is a knot. Note that from now on we will allow multiple edges from one node to another.

Some results of knots will be the basis for future deadlock detection algorithms:

- If a digraph has a knot, then it has a cycle.
- No node in a knot is a sink.
- There is no path from a node in a knot to a sink.
- A digraph is free of knots if and only if, for each node z, z is a sink or there is a path from z to a sink.

In the next section we introduce the graph model of completely general systems of processes and resources.

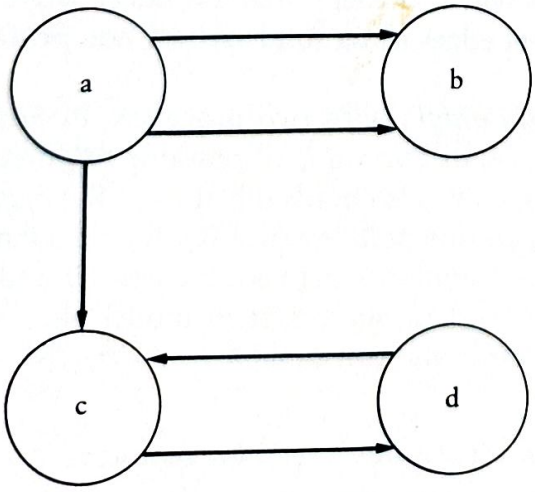


Figure 4.1 A Bipartite Digraph.

4.4 The General Model

A general resource system model consists of a nonempty set $\pi = \{P_1, \ldots, P_n\}$ of processes and a nonempty set $\rho = \{R_1, \ldots, R_m\}$ of resources. The set ρ is partitioned into two disjoint subsets, ρ_r and ρ_c, representing reusable and consumable resources, respectively. Associated with each resource R_j is the current number of available units $r_j \geq 0$. Associated with each reusable resource R_j is the total number of units $t_j > 0$. For each consumable resource R_j there is a nonempty set of processes that produce units of R_j.

4.4.1 General Resource Graph

A particular state in the general resource system model is completely described by the number of units of each resource that each process requests, the number of units of each reusable resource held by each process, and the current available inventory of each resource. For each state there is a corresponding bipartite digraph:

- Nodes N = {Processes} ∪ {Resources}. To distinguish between them in figures, we draw processes as square boxes, □, and resources as round circles, ○. For reusable resources R_j, we represent the total inventory t_j by placing small tokens in the circle for R_j. For consumable resources, the tokens represent the number of currently available units r_j.
- Edges E are of three types:

  Request edges (P_i,R_j) connect process to resources. They represent units of R_j that have been requested but not yet obtained.

  Assignment edges (R_j,P_i) connect resources to processes. These arcs signify units of reusable resources R_j currently held by P_i.

  Producer edges (R_j,P_i) connect consumable resources to processes that produce them. These are permanent identifiers of the producers P_i of R_j.

Notice that the graph is bipartite. Since request and assignment edges may appear and disappear with state changes (as we will soon see), and producer edges are permanent, we draw request and assignment edges using solid arrows, and producer edges with dashed arrows.

In Figure 4.2 we see a *general resource graph*, with two processes, P_1 and P_2, and two resources, R_1 and R_2. Process P_1 holds two units of reusable resource R_1. R_1 has a total inventory of four units. Process P_2 also holds one unit of R_1, requests one unit of R_2, and is the only producer of consumable resource R_2. R_2 has a current inventory of one unit. Observe the structural similarity between Figures 4.1 and 4.2.

Graphs representing states in the general resource system model obey some reasonable restrictions with respect to reusable and consumable resources. For reusable resources:

- The number of assignment edges directed from R_j cannot exceed t_j.
- At all times the available units $r_j = t_j -$ (number of edges directed from R_j).

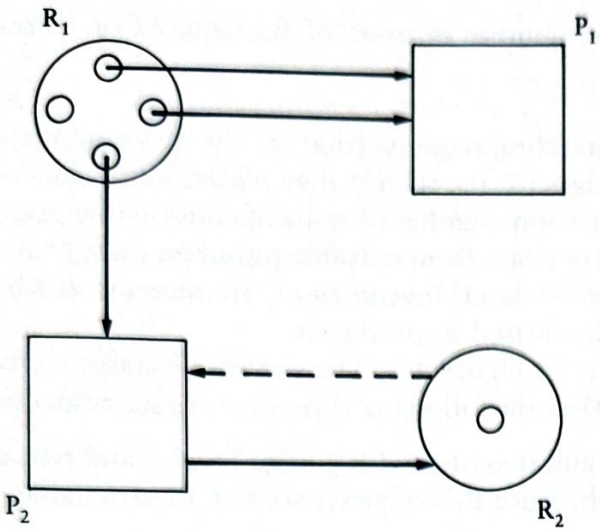


Figure 4.2 A General Resource Graph.

- For each process P_i, [number of request edges (P_i,R_j)] + [number of assignment edges (R_j,P_i)] $\leq t_j$.

For consumable resources:

- Edge (R_j,P_i) exists if and only if P_i produces R_j.
- The inventory r_j at any time is constrained only to be nonnegative. This means that systems containing consumable resources may have an infinite number of states.

4.4.2 Operations on Resources

Processes perform operations that change the state of the general resource system. Each state has a corresponding graph. Here we describe the operations that processes execute—*requests, acquisitions,* and *releases*—and the corresponding graph alterations necessary to reflect these new states. All the operations are constrained by the resource restrictions above.

REQUESTS If process P_i has *no* outstanding requests (that is, if it is executable), then it may request units of any number of resources R_j, R_k, ... To reflect this in the graph, add edges (P_i,R_j), (P_i,R_k), . . . in multiplicities corresponding to the number of units of each resource requested.

ACQUISITIONS If process P_i has outstanding requests, and for each requested resource R_j, the number of requested units does not exceed the current inventory r_j (that is, if *all* requests are grantable), then P_i may acquire *all* requested resources. The graph is altered as follows. For each request edge (P_i,R_j) to a reusable resource, reverse the edge direction to make it an assignment edge (R_j,P_i). Each request edge to a consumable resource disappears, simulating the consumption of units by P_i. In either case

each inventory r_j is reduced by the number of units of R_j acquired or consumed by P_i.

RELEASES If process P_i has no outstanding requests (that is, if it is executable), and there are assignment or producer edges (R_j,P_i), then P_i may release any subset of the reusable resources it holds or produce any number of units of consumable resources for which it is a producer. Assignment edges from reusable resources disappear from the graph, but producer edges are permanent. Inventories r_j are incremented by the number of units of each resource R_j released or produced.

Consider again the system state in Figure 4.2. The successive states shown in Figures 4.3(a), 4.3(b), and 4.3(c) reflect the following three operations, respectively:

(a) P_1 requests one unit of R_1 and two units of R_2. Edge (P_1,R_1) and two edges (P_1,R_2) appear in the graph. Since P_1's requests are not all satisfiable, P_1 is blocked in this state.

(b) P_2's request for one unit of R_2 is granted. Since R_2 is consumable, the request edge (P_2,R_2) disappears. A token disappears from R_2.

(c) P_2 produces three units of R_2. Three tokens appear in R_2. P_1 is no longer blocked in this state.

The next subsection discusses the characterization of deadlock in terms of general resource system graphs.

4.4.3 Necessary and Sufficient Conditions for Deadlock

In this section we introduce the concept of graph reduction and derive conditions that relate reducibility of a resource graph to deadlock. A *graph reduction* reflects an optimistic view about the behavior of a process. In particular, a reduction by a process P_i simulates the acquisition of any outstanding requests, the return of any

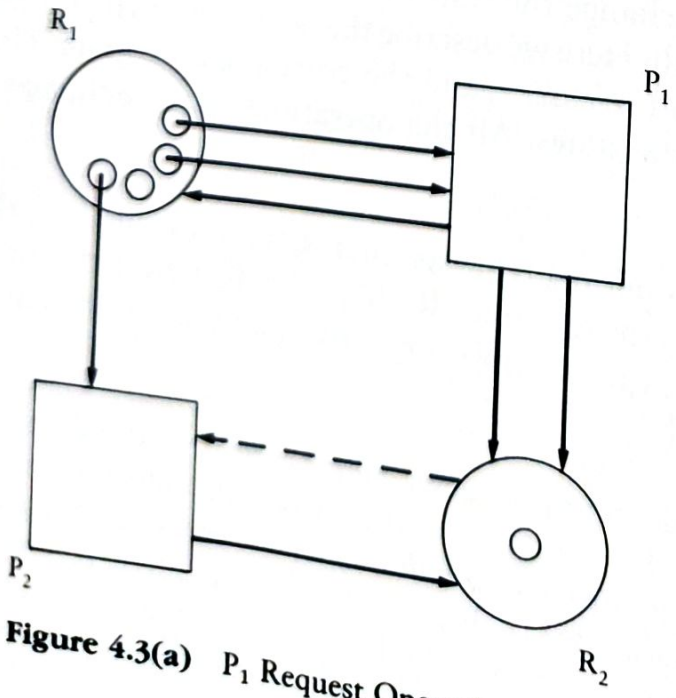


Figure 4.3(a) P_1 Request Operation.

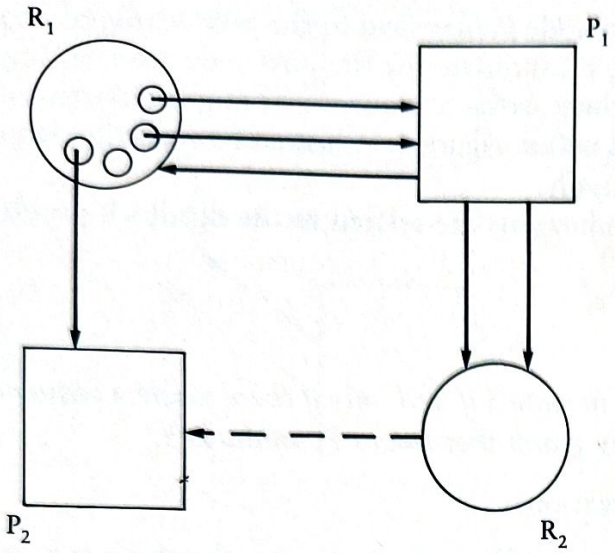


Figure 4.3(b) P_2 Acquisition Operation.

allocated units of a reusable resource, and, if P_i is a producer of a consumable resource, the production of a "sufficient" number of units to satisfy all subsequent requests by consumers. In the case of a consumable resource R_j, the new inventory is represented by ∞ to indicate that all future requests for R_j are grantable.

Formally, a graph may be *reduced* by a nonisolated node, representing an unblocked process P_i, in the following manner:

- For each resource R_j, delete all edges (P_i,R_j) and if R_j is consumable, decrement r_j by the number of deleted request edges.
- For each resource R_j, delete all edges (R_j,P_i). If R_j is reusable, then increment r_j by the number of deleted edges. If R_j is consumable, set $r_j = \infty$.

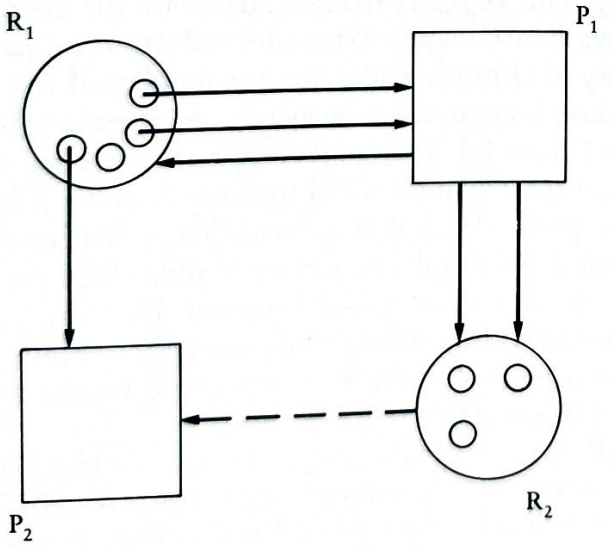


Figure 4.3(c) P_2 Release Operation.

A reduction of a graph by a process node P_i may lead to the unblocking of another process node P_j, thereby making P_j a candidate for the next reduction. A graph is said to be *completely reducible* if there exists a sequence of graph reductions that reduces the graph to a set of isolated nodes. Figure 4.4 illustrates a reduction sequence applied to a completely reducible graph.

The manner in which graph reductions are related to the deadlock problem is stated in the following result:

THEOREM 4.1

A process P_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding graph that leaves P_i unblocked.

This result follows from two observations:

1. Such a sequence of reductions explicitly demonstrates the manner in which a blocked process P_i may become a runnable process.
2. If a blocked process is to become runnable, then some sequence of process executions must exist to unblock it.

An immediate consequence of this result is:

COROLLARY

If a graph is completely reducible, then the state it represents is not deadlocked.

While Theorem 4.1 and its corollary are important for characterizing properties of deadlocked states, they do not, by themselves, provide the basis for practical algorithms for deadlock detection. The reasons for this impracticality are illustrated in two examples.

First, reducibility may be dependent on the order of the reductions. This is illustrated in Figure 4.5, where the only possible sequence of reductions leading to a completely reduced graph is P_1 P_2. This suggests that an algorithm for detecting deadlock may have to, in the most general case, inspect n! possible reduction sequences, yielding a worst-case time complexity of O(mn!), where m is the number of resource types and n is the number of processes. Second, the converse to the corollary is not true, as illustrated by the example in Figure 4.6. This graph may be reduced by either process node P_1 or process node P_2, after which no reductions are possible. Clearly neither process is deadlocked, although deadlock is imminent. Thus, the corollary provides a conclusion of no deadlock if the graph is completely reducible, but fails to provide a conclusion if the graph is not reducible (see, however, Problem 3).

In order to derive practical algorithms for dealing with deadlock, certain constraints must be imposed on the behavior of the system of processes. The first such constraint deals with the occurrence of grantable requests.

A system state is *expedient* if all processes requesting resources are blocked. In such systems, a new allocation of resources can take place only at the time of a request or at the time of a release. Expediency is not considered a major restriction since most practical systems naturally follow such an allocation policy.

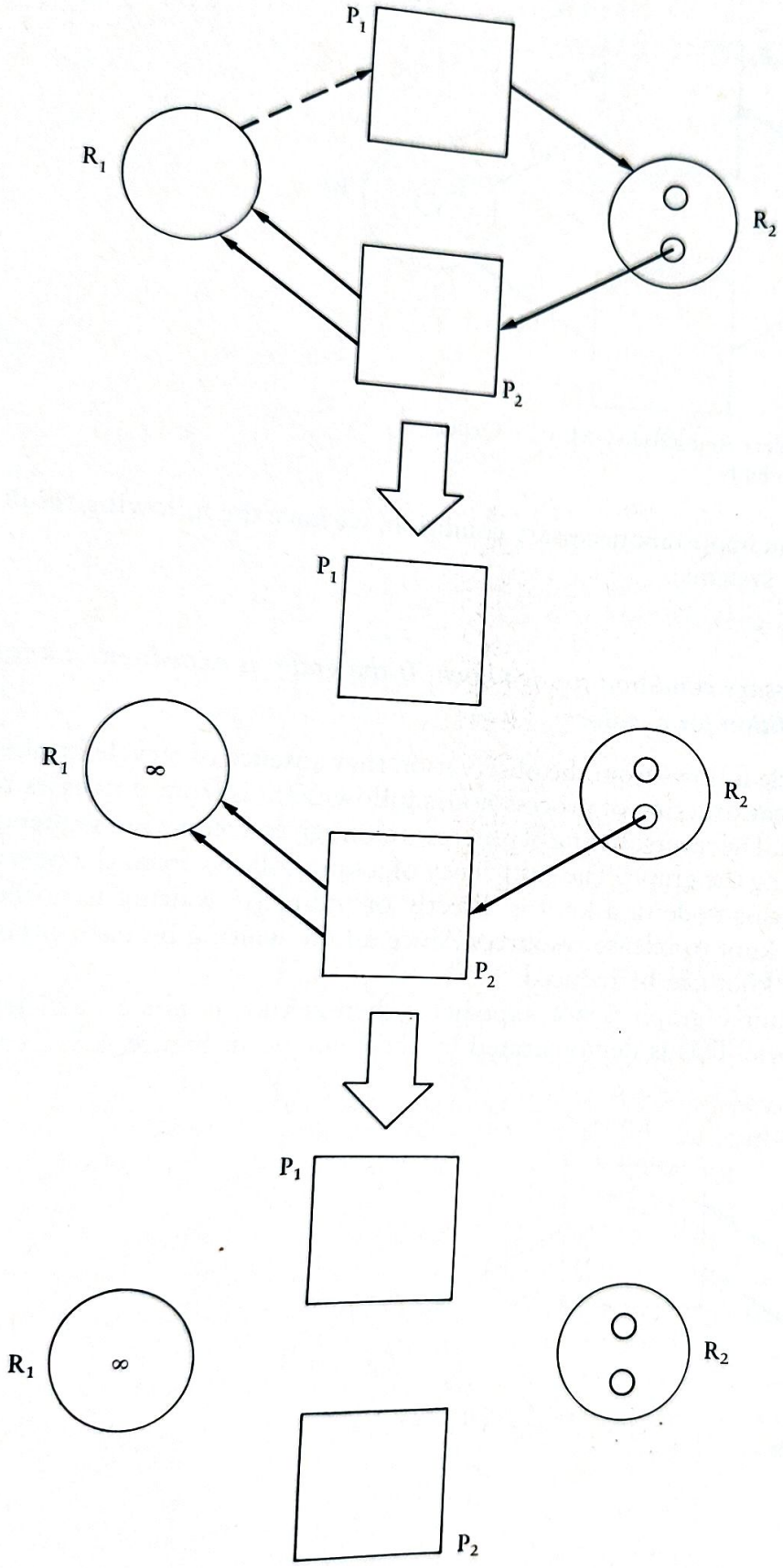


Figure 4.4 A Reduction Sequence.