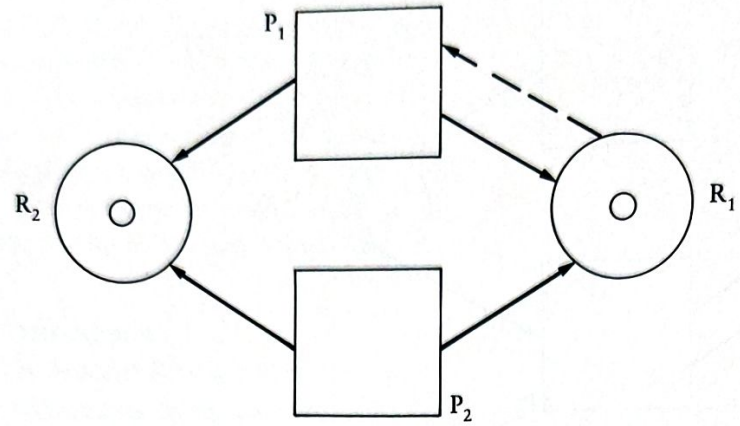

Figure 4.5 Complete Reducibility May Be Order-Dependent.

Combined with an important necessary condition, we have the following result for a general resource system:

THEOREM 4.2

A cycle is a necessary condition for deadlock. If the graph is expedient, a knot is sufficient condition for deadlock.

The necessity of a cycle follows from the observation that absence of a cycle implies the existence of a linear ordering of process nodes following arcs from processes to resources to processes. The reverse listing of processes alone gives a reduction sequence for completely reducing the graph. The sufficiency of a knot follows from the observation that each process node in a knot is directly or indirectly waiting for other process nodes in the knot to release resources. Since all are waiting on each other, no process node in a knot can be reduced.

If a general resource graph is not expedient, then a knot is not a sufficient condition for deadlock. This is demonstrated by the example in Figure 4.7. Expe-

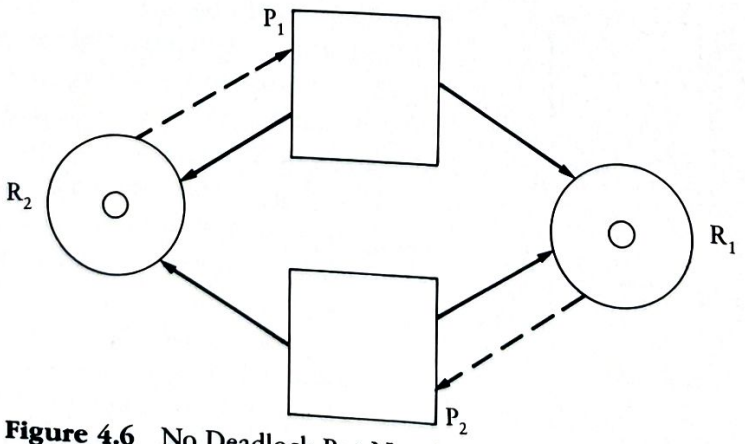

Figure 4.6 No Deadlock But Not Completely Reducible.

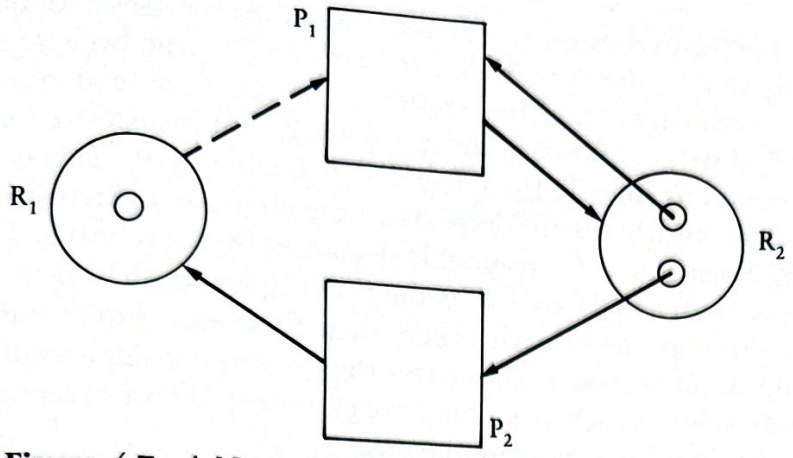

Figure 4.7 A Nonexpedient General Resource Graph.

diency will be assumed throughout the remainder of this chapter. Figure 4.7 also demonstrates that the presence of a cycle in a general resource graph is not a sufficient condition for deadlock. This is true even for expedient systems.

4.5 Special Cases with Useful Results

By further restricting general resource systems, we can obtain useful approaches to managing deadlock in situations common to current computer systems.

4.5.1 Single-Unit Requests

In these systems a process may have at most one outstanding request for a single unit of some resource. These *single-unit request* systems are also expedient. Some message-passing systems are examples of systems with these characteristics. Deadlock may be efficiently detected in this case.

In the most general case of Section 4.4, the existence of a knot in an expedient graph implied a deadlock state. The following three facts lead to two efficient detection algorithms.

First, if a graph is a deadlock state, then a knot exists. If we were to assume that there is no knot, then each process P_i is either a sink (it is not blocked), or there is a path (P_i, R_j, P_k, . . . , P_x, R_y, P_z) so that node P_z is a sink. The sink must be a process because of expedience. Then P_z is not blocked and we can reduce the graph by P_z. Node R_y is a resource, and reduction by P_z must have increased R_y's inventory. Since P_x is requesting just one unit of R_y, it is no longer blocked and we reduce by P_x. We continue up the path in this way until P_i is no longer blocked. Thus, there is no deadlock. Combined with the second part of Theorem 4.2, we see that the existence of a knot is equivalent to deadlock in single-unit request systems, and graph-knot detection algorithms can be used to determine the existence of deadlock.

Second, any two reduction sequences lead to the same irreducible state. The truth of this statement relies on two propositions. (1) If two sequences of reductions

are applied to a state S, and the two sequences contain the same processes in different order, then the resulting state is the same in either case. This is true because each reduction removes the same edges in either sequence, so the same end result is achieved. (2) In a state S, if we could reduce the graph by P_i, but instead we reduce the graph by other processes to state T, then T is still reducible by P_i. This is true because reusable inventories clearly do not decrease in the alternate reduction to T, and expedience prevents inventories of consumable resources from declining. These propositions allow one to verify that any two reduction sequences that yield irreducible states must be permutations of each other; then these two states must be identical. This fact is important because it eliminates the need to consider reduction sequences that are permutations of each other, and the exponential cost of searching for an irreducible state is reduced to a polynomial cost.

Third, process P is not deadlocked in state S if and only if P is a sink or on a path to a sink. This fact leads to an effective method for checking if a particular process is deadlocked.

On the basis of these facts, two useful algorithms are now available. The first detects the presence of a knot to determine the presence or absence of deadlock. The expression "L || F" appends the node F to the node list L (if it does not already exist in L).

ALGORITHM 4.1

Is state S a deadlock state?

```
L := [List of sinks in state S]
for the next N in L do
  for F so that (F,N) is an edge do
    L := L||F
  endfor
endfor
Deadlock := {Nodes} ≠ L
```

List L begins as a list of initial sinks. The outer loop processes through L to find nodes F that are on a path of length 1 from sink nodes (or paths to sinks) already found. These nodes are appended to L. When the outer loop reaches these added nodes, new nodes that are two steps from the original sinks are appended. When the entire list L has been processed, it will contain all nodes that are not participating in a knot. If this is not all nodes, then {Nodes} − L is the group of processes and resources defining the deadlock condition.

The second algorithm attempts to find a path from a process P to a sink in order to establish that P is not deadlocked.

ALGORITHM 4.2

Is process P deadlocked?

```
Deadlock := true; L := [P]
for the next N in L while Deadlock do
  for each F so that (N,F) is an edge do
```

```
        if F is a sink then Deadlock := false
        else L := L ||F
        endif
      endfor
    endfor
```

The list L initially contains P, the process in question. We travel on all the paths from nodes in L to neighbor nodes F. If a neighbor is a sink, the algorithm terminates, reporting no deadlock. Otherwise the neighbor is appended to L for possible later processing. If the entire list is processed and no sink is found, then P is part of a knot and is deadlocked.

Each of these algorithms executes in time proportional to the number of edges in the graph. Since our graphs are bipartite with n processes and m resources, we know that the algorithms execute in O(mn) time. Further, the second algorithm is likely to terminate quickly under most circumstances.

Thus, continual deadlock detection can be done at a reasonable cost. When process P requests an unavailable unit, the second algorithm should be executed. As a potentially cheaper but less immediate alternative, the first algorithm should "occasionally" be executed.

4.5.2 Consumable Resources Only

In some system models, reusable resources may not be present. For example, models of operating systems that rely solely on message passing may depict only consumable resources. We assume that each process is a producer or consumer of at least one resource. Although we cannot obtain an efficient detection algorithm for reasons explained in Section 4.4, we can *prevent* deadlock by a conservative system design method.

We define the *claim-limited graph* of the *consumable resource system* to be the graph corresponding to the special state in which all inventories are exhausted and each consumer of each resource requests one unit. Figure 4.8 shows a simple example. Clearly, this may be a desperate situation. If there are no producer processes that consume no resources, all processes must remain blocked and hence deadlocked. The result that leads to a scheme for preventing deadlock is:

> A consumable resource system is secure from deadlock if and only if its claim-limited graph is completely reducible.

This result can be used to analyze a computer system during its design. When all resources, and consumer and producer processes are known, the claim-limited graph is constructible. Its complete reducibility means that the system cannot enter a deadlock state. However, this is a very strong deadlock prevention criterion and thus results in poor resource utilization. It is useful in consumable resource systems for which deadlock prevention is an absolute requirement.

The following example demonstrates why the result is often too strong to be of practical value. It is assumed that a "send" operation does not cause a process to

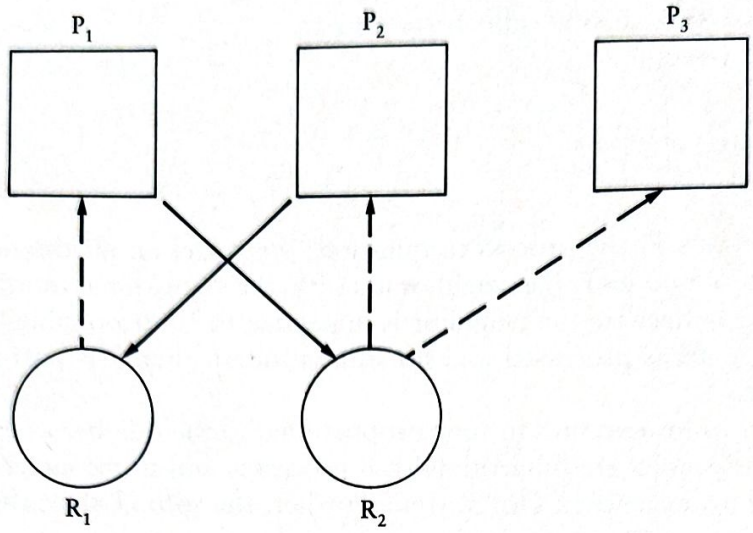

Figure 4.8 A Claim-Limited Graph.

become blocked. Here the claim-limited graph of the message-passing system is not reducible even though the system is deadlock-free:

```
P1: while true do
    begin
        produce M
        send (P2,M)
        receive (P2,L')
    end

P2: while true do
    begin
        produce L
        send (P1,L)
        receive (P1, M')
    end
```

This system results in the claim-limited graph shown in Figure 4.9.

4.5.3 Reusable Resources Only

A *reusable resource system* is a special case of a general resource system in which there are only reusable resource types. Assuming expediency, efficient algorithms can be defined for all three policies: detection, prevention, and avoidance. Each of these policies will be examined in subsequent sections.

Detection in Reusable Resource Systems

The following major result provides necessary and sufficient conditions for the existence of deadlock, establishing the basis for efficient deadlock detection:

THEOREM 4.3

Let S be any state of a reusable resource system. Any sequence of reductions of the corresponding graph leads to a unique graph that cannot be reduced. S is not deadlocked if and only if S is completely reducible.

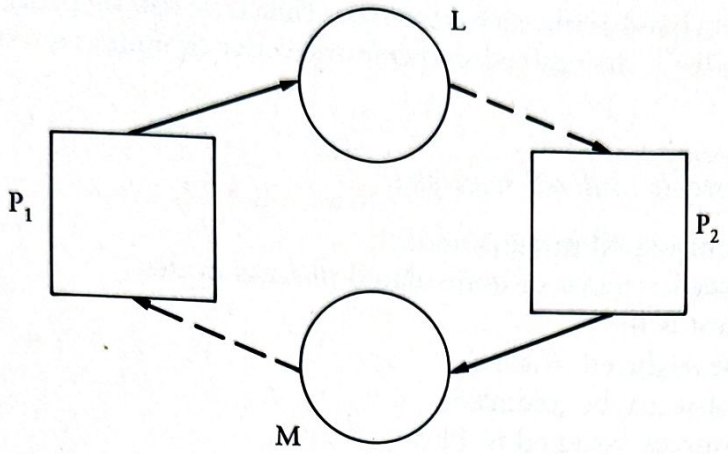

Figure 4.9 Irreducible Claim-Limited Graph of a Deadlock-Free System.

The first statement follows from the observation that a reduction never decreases the number of available units of a resource. As a consequence, reducibility is not order-dependent and any two reduction sequences that leave the graph in an irreducible state must necessarily involve the same set of process nodes. For the second statement, it was shown in Section 4.4.3 that complete reducibility implies absence of deadlock. To establish the necessity of the condition, note that if S is not deadlocked, then no process is deadlocked. From the first statement, it follows that any reduction sequence leads to a completely reduced graph.

Theorem 4.3 provides the basis for efficient deadlock-detection algorithms. The first of these algorithms is appealing because of its simplicity:

ALGORITHM 4.3

Detection in reusable systems (simple version):

```
L := [List of process nodes];
finished := false;
while ((L ≠ ∅) and not finished) do
   P := First process in L by which graph can be reduced;
   if P ≠ nil then Reduce graph by P; Remove P from L
   else finished := true
   endif
endwhile
```

Upon completion of Algorithm 4.3, L ≡ [List of deadlocked processes]. The algorithm has an $O(mn^2)$ worst-case time complexity since a reduction can involve m resource nodes, the selection of P on the *ith* pass of the loop may require inspection of $(n - i + 1)$ nodes, and n total passes may be required.

At the expense of extra storage, a more efficient algorithm is available. The following data structures are assumed to be maintained by this system:

- *wait_count:* Associated with each process, this denotes the number of resources for which the process is currently waiting.

- *ordered_requests:* Associated with each resource, this is a list of processes that request units. This list is maintained in increasing order of units requested.

ALGORITHM 4.4
Detection in reusable systems (ordered requests):

```
L := [List of nonisolated process nodes];
list_to_be_reduced := [List of nonisolated process nodes
whose wait_count is 0];
while (list_to_be_reduced ≠ ∅) do
  Select P from list_to_be_reduced;
  for R ∈ {Resources assigned to P} do
    Increase available units of R by number of units assigned to P;
    for each process Q in ordered_requests for this R
    whose request can be satisfied do
      Decrease wait_count of Q by 1;
      if wait_count of Q = 0 then
        Add Q to list_to_be_reduced
      endif
    endfor
  endfor;
  Remove P from L
endwhile
```

Upon completion of Algorithm 4.4, L ≡ [Deadlocked processes]. Since the selection of Q does not require a search of *ordered_requests*, the time complexity of this algorithm is O(mn).

The next example illustrates the use of Algorithm 4.4. Consider a system of three processes, P_1, P_2, and P_3, and three reusable resources, R_1, R_2, and R_3, with total units of 2, 3, and 2, respectively.

1. Initial State
   wait_count: 0 for all processes
   ordered_requests: empty for all resources
2. P_1 requests and acquires two units of R_3.
   wait_count: 0 for all processes
   ordered_requests remains unchanged
   The deadlock detection algorithm is not applied since all *wait_counts* are zero.
3. P_3 requests and acquires two units of R_2.
   The *wait_counts* and *ordered_requests* remain unchanged
4. P_3 requests two units of R_3.
   wait_count[P_3] = 1, all other *wait_counts* are 0
   ordered_request[R_3] = {P_3}, all other *ordered_requests* are empty
   The deadlock detection algorithm is applied with L = {P_1,P_3} and

list_to_be_reduced = $\{P_1\}$. The algorithm terminates in one pass with the conclusion that deadlock does not exist.

5. P_2 requests and acquires two units of R_1.
   The *wait_counts* and *ordered_requests* remain unchanged
6. P_2 requests two units of R_2.
   wait_count$[P_1] = 0$, *wait_count*$[P_2]$ = *wait_count*$[P_3] = 1$
   ordered_requests$[R_1] = \emptyset$, *ordered_requests*$[R_2] = \{P_2\}$,
   ordered_requests$[R_3] = \{P_3\}$
   The deadlock detection algorithm is applied with $L = \{P_1,P_2,P_3\}$ and list_to_be_reduced = $\{P_1\}$. The algorithm reduces P_1, P_3, and finally P_2.
7. P_1 requests a unit of R_1.
   wait_count: 1 for all processes
   ordered_requests$[R_1] = \{P_1\}$, *ordered_requests*$[R_2] = \{P_2\}$
   ordered_requests$[R_3] = \{P_3\}$
   The deadlock algorithm is applied with $L = \{P_1,P_2,P_3\}$ and list_to_be_reduced = $\emptyset$. No processes are reducible and the algorithm terminates with the conclusion that all three processes are deadlocked.

A requirement of Algorithm 4.4 is that the system maintain the *wait_count* and *ordered_requests* data structures. However, this information allows efficient allocation to waiting processes when units of a resource are released.

Since only unsatisfiable requests for resources can cause a deadlock in reusable resource systems, detection needs to be performed only at request time and, consequently, deadlock is detected as soon as it occurs. Assuming that the system does not already contain a deadlock at the time of such a request, Algorithms 4.3 and 4.4 can be made more efficient by terminating them as soon as it has been determined that a reduction sequence involves the requesting process.

Avoidance in Reusable Resource Systems

The method presented in this section requires that each process be able to anticipate its maximum total resource requirements, called its *claim*, at any point during execution. Clearly no process can be allocated more than its claimed resources. The original algorithm for this process was first proposed for a single resource by Dijkstra [1968], and was called the "Banker's Algorithm." It was extended to multiple resources by Habermann [1969].

Formally, let c_{ij} denote the claim of process P_i for resource R_j, where $0 \leq c_{ij} \leq t_j$, and let C[1:n,1:m] denote the claim matrix for n processes and m resources. For a given state, the *maximum-claim graph* reflects the projected worst-case future state and is constructed from the graph of the current state by adding additional request edges (P_i,R_j), called *claim edges*, until the number of request edges plus the number of assignment edges (R_j,P_i) is equal to c_{ij}. These claim edges are denoted by dotted edges. A state is defined as *safe* if its corresponding maximum-claim graph is deadlock-free. A *maximum-claim reusable resource system* is one in which all states are safe.

Whenever a process makes a request, the following algorithm is executed.

ALGORITHM 4.5

Avoidance

1. Project the future state by changing the request edge to an assignment edge.
2. Construct the maximum-claim graph for this state and analyze it for deadlock. If deadlock exists, then defer granting the request; otherwise, grant the request.

Figures 4.10(a), 4.10(b), and 4.10(c) show an execution of the deadlock-avoidance algorithm for a system of two processes, P_1 and P_2, and two reusable resource types, R_1 and R_2, consisting of 3 and 2 units, respectively. Figure 4.10(a) depicts the initial safe state along with the claim matrix. Figure 4.10(b) depicts a request by P_1 for a unit of R_2, along with the corresponding maximum-claim graph for the projected state. Since the maximum-claim graph contains a deadlock, the request by

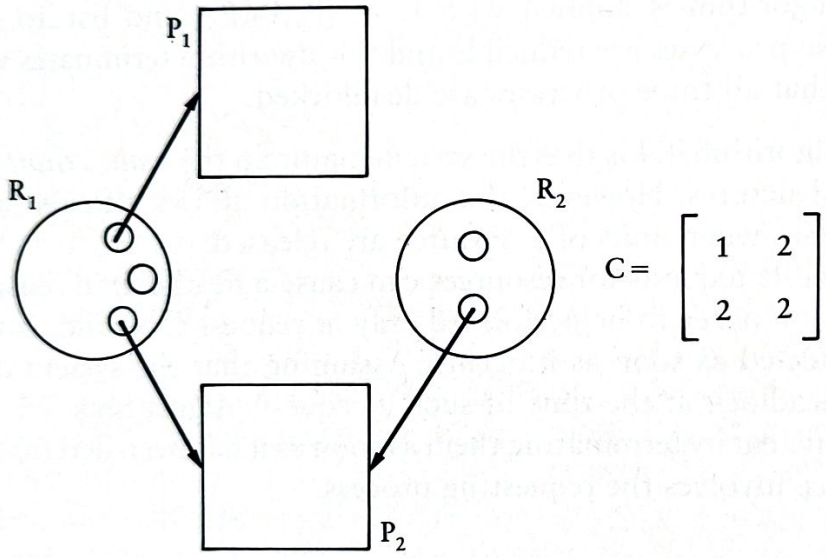

Figure 4.10(a) Initial Safe State and Claim Matrix.

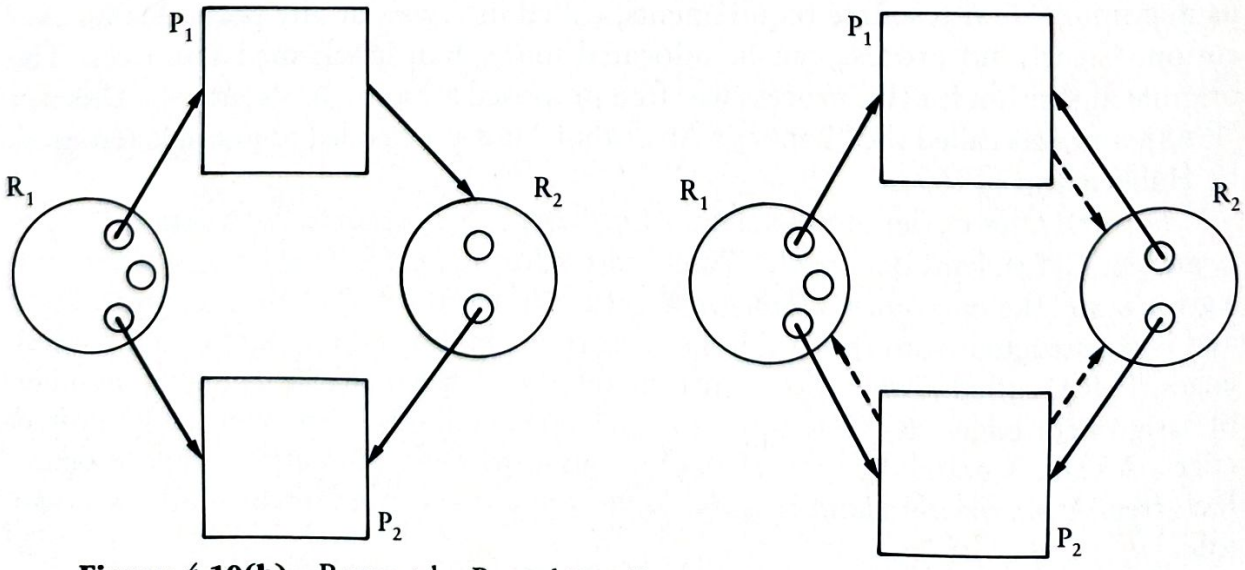

Figure 4.10(b) Request by P_1 and Deadlocked Claim-Limited Graph.

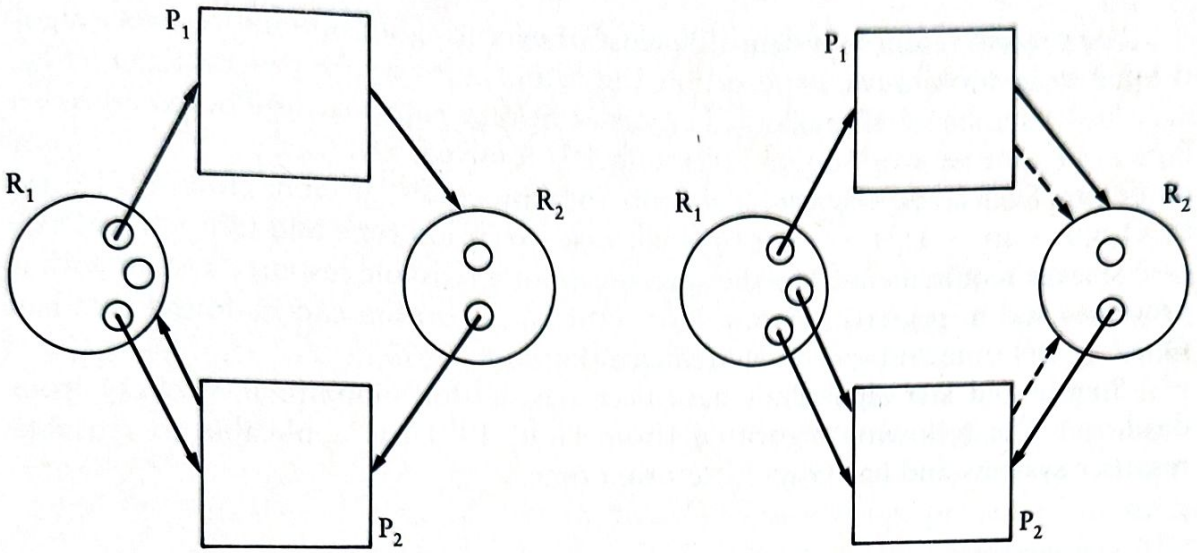

Figure 4.10(c) Request by P_2 and Reducible Claim-Limited Graph.

P_1 cannot be safely granted now. Figure 4.10(c) shows that a subsequent request by P_2 for a unit of R_1 can be safely granted.

Given an initial safe state, the deadlock-avoidance algorithm will ensure that every successive state will also be safe. However, although the algorithm is useful in specific circumstances involving particular resource types, there may be major costs associated with its use. First, the algorithm must be executed for every request prior to granting it. Second, the restriction on resource allocation may severely degrade system resource utilization. Unless the claims are simultaneously realized by the processes, it is impossible to actually reach the pessimistic state represented by a maximum-claim graph. Thus, if the claims are not precise, many requests may be deferred when they might in fact be safely granted. In many cases it is impossible for a user to estimate resource requirements accurately.

4.6 Recovery from Deadlock

Recovery from deadlock requires the *rollback* of one or more deadlocked processes. An extreme case of rollback is to abort a process and restart it at its beginning. If the system has a checkpoint/restart facility, then only partial rollback may be required, although the rollback may involve more than the deadlocked processes. In many database processing systems, discriminate rollback of an application is possible; this typically requires a restart only of the current transaction. Whenever rollback occurs, the system must guard against repeated deadlock involving the same processes. The countermeasures may involve raising the priority according to the number of rollbacks or the use of timestamps.

Regardless of the degree of rollback, the ideal selection is one that minimizes the cost of reexecuting the rolled-back processes. Numerous factors may be involved in determining the cost of restarting an individual process: priority, type and amount of current resource allocation, number of processes affected, amount of service received, and amount of service required to finish.

For general resource systems, the cost of executing an optimal recovery algorithm may be prohibitive. Let c_i denote the cost of recovery for process i and let $P_1, \ldots, P_k$ denote the set of deadlocked processes. Select a subset of deadlocked processes $P_{i_1}, \ldots, P_{i_q}$ for removal so that (1) deadlock is resolved, and (2) $c_{i_1} + \cdots + c_{i_q}$ is a minimum. Such a cost-recovery algorithm for n processes, given in Holt [1971], has an $O(n(n + (n - 1) + \cdots + n!))$ worst-case execution time and $O(n * n!)$ worst-case storage requirements. For the special case of a reusable resource system with n processes and m resource types, a more efficient algorithm can be found that has $O((m + n)^2)$ time and space requirements [Purdom, 1968].

Simple and fast algorithms have been devised for suboptimal recovery from deadlock. The following algorithm [from Holt, 1971], is applicable to reusable resource systems and has O(mn) execution time.

ALGORITHM 4.6

Recovery:

```
L = [Processes ordered by increasing termination cost];
while (L ≠ ∅) do
  Select next P from L;
  Terminate P and remove P from L;
  Use deadlock-detection Algorithm 4.4 to reduce as many processes as
  possible, removing all liberated processes from L
end
```

If continuous deadlock detection is feasible, then a simple, fast suboptimal recovery algorithm is to abort the requesting process.

4.7 Prevention by System Design

As mentioned in Section 4.2.4, deadlock prevention is accomplished by designing the system so that one of four necessary deadlock conditions is denied. A prevention policy is attractive from the point of view that run-time overhead, required for deadlock testing in each of the other two policies, is avoided. In general, it is not practical to execute a deadlock-detection algorithm each time a process requests, acquires, or releases a shared variable. There are three important prevention methods: collective requests, ordered requests, and preemption [Havender, 1968].

In the *collective-requests method,* a process requests and is allocated all resources that it will need during any moment of its execution. New requests are allowed only if the process first releases everything that it has been allocated. This method denies the partial allocation condition necessary for deadlock to exist. The method is simple and effective, but may seriously degrade utilization of system resources since a process may hold resources for extended periods of time during which they are not needed. Starvation is another potential difficulty since a process with large resource requirements may be indefinitely blocked. For example, processes with large memory requirements may fit into this category. The usual countermeasures to starvation

are *aging* or running such a job at a designated period of time when demand is usually low. If a large job is run during a peak period, an accounting question is raised since it is not clear who should be charged for other idled resources. Although the method of collective requests has clear disadvantages, it is a particularly useful approach in dealing with shared variables when it is known that the duration of use will be short.

In the *ordered requests method,* a fixed ordering $C_1, \ldots, C_k$ is imposed on resource classes, each containing one or more resources. If a process holds a resource in class C_i, it can only request resources of classes C_j for $j > i$. This method denies the existence of a cycle in the resource graph that Theorem 4.2 established as a necessary condition for deadlock. Although the ordered-request method is more efficient than the collective-request method, it still may seriously degrade system performance. Since the order of resource usage differs from one process to another, processes may be forced to request and be allocated resources unnecessarily early. If resources are reordered, programs optimized for an old ordering may need to be redesigned. Since requests for an already allocated resource are not allowed without release of all resources in this and subsequent classes, a typical approach may be for a process to request the maximum quantity at each stage. In spite of the potentially serious impact on resource utilization, the ordered-resource approach is a common technique used to prevent deadlocks.

Preemption is a third method used to prevent deadlock. For some resource types or processes, however, the penalty is too high to make this method effective. This is particularly true for real-time applications, where it may be impossible to recover a previous system state. Even if recovery is possible, the cost of restarting a long-running process may be too high. Resources commonly considered preemptible include the processor in time-shared systems, main memory in swapping systems or virtual memory systems, and access to data in transaction-based systems.

4.8 Total System Design

Deadlock control in operating systems typically involves a mixture of several policies, and several methods within each policy. Most operating systems are hierarchically organized in layers, each of which modifies and extends the capabilities of the underlying layer. A primitive operation in one layer is implemented in terms of operations in lower layers. This leads to a natural ordering of groups of resources—resources used in higher levels are requested before resources in lower levels. Within each group, other techniques described in this chapter for dealing with deadlock may be employed. The following is a list of resource types and commonly used policies [Howard, 1973]:

- *Swap space in secondary memory:* Preallocate the maximum amount of space needed by each process.
- *Job or job-step resources:* Resources, such as files and special I/O devices, are typically needed for the duration of job or job step. A common approach in

batch-processing systems is to use an avoidance strategy since considerable information about future resource usage can be deduced from job-control statements. Another approach might be to use a collective-request prevention strategy, with new allocations occurring at the beginning of each job step and all allocations released at the end of each job step. Some systems use only deadlock detection for files.

- *Main memory for user jobs:* Preemption is the most effective approach in a paging, segmentation, or swapping system. If this is not possible, main memory should be included in the class of job resources.
- *Internal system resources:* Resources such as control blocks, buffers, and semaphores are included in this class. Since access is frequent, a prevention strategy such as resource ordering is a typical choice. The hierarchical nature of the system may provide a natural choice for the ordering. If ordering is difficult, a collective-request method may be used. For example, see the simultaneous P operation in Chapter 2.

4.9 Summary

Deadlock is a difficult problem with no single solution to fit all circumstances. The three general policies are detection, avoidance, and prevention. In a general resource system, a cycle in the resource graph is a necessary but not sufficient condition for deadlock existence. On the other hand, complete reducibility of the resource graph is a sufficient but not necessary condition for absence of deadlock. As a consequence, there are no known efficient algorithms for the completely general case. Imposing certain constraints, however, allows for the derivation of efficient algorithms for detection and avoidance.

Assuming expedience and single unit requests, a knot in the resource graph and complete reducibility are both necessary and sufficient conditions for deadlock. Necessary and sufficient conditions for a consumable resource system to be secure from deadlock is the total reducibility of the claim-limited graph. These latter two results are important in the design of message-based systems.

In reusable resource systems, complete reducibility of the resource graph is both necessary and sufficient for existence of deadlock. Efficient algorithms can be devised for both deadlock detection and deadlock avoidance.

Three important prevention methods employed in many operating systems are collective resource requests, ordered resource requests, and preemption. Most operating systems are designed using combinations of policies and methods.

Key Words

acquisition operation
aging
assignment edges
avoidance
bipartite graph
blocked
claim
claim-limited graph