CHAPTER 6

Concurrency: Deadlock and Starvation

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

STATUTE PASSED BY THE KANSAS STATE LEGISLATURE, EARLY IN THE 20TH CENTURY
—*A TREASURY OF RAILROAD FOLKLORE*,
B. A. BOTKIN AND ALVIN F. HARLOW

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- List and explain the conditions for deadlock.
- Define deadlock prevention and describe deadlock prevention strategies related to each of the conditions for deadlock.
- Explain the difference between deadlock prevention and deadlock avoidance.
- Understand two approaches to deadlock avoidance.
- Explain the fundamental difference in approach between deadlock detection and deadlock prevention or avoidance.
- Understand how an integrated deadlock strategy can be designed.
- Analyze the dining philosophers problem.
- Explain the concurrency and synchronization methods used in UNIX, Linux, Solaris, and Windows 7.

This chapter examines two problems that plague all efforts to support concurrent processing: deadlock and starvation. We begin with a discussion of the underlying principles of deadlock and the related problem of starvation. Then we examine the three common approaches to dealing with deadlock: prevention, detection, and avoidance. We then look at one of the classic problems used to illustrate both synchronization and deadlock issues: the dining philosophers problem.

As with Chapter 5, the discussion in this chapter is limited to a consideration of concurrency and deadlock on a single system. Measures to deal with distributed deadlock problems are assessed in Chapter 18. An animation illustrating deadlock is available online. Click on the rotating globe at WilliamStallings.com/OS/OS7e.html for access.

6.1 PRINCIPLES OF DEADLOCK

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

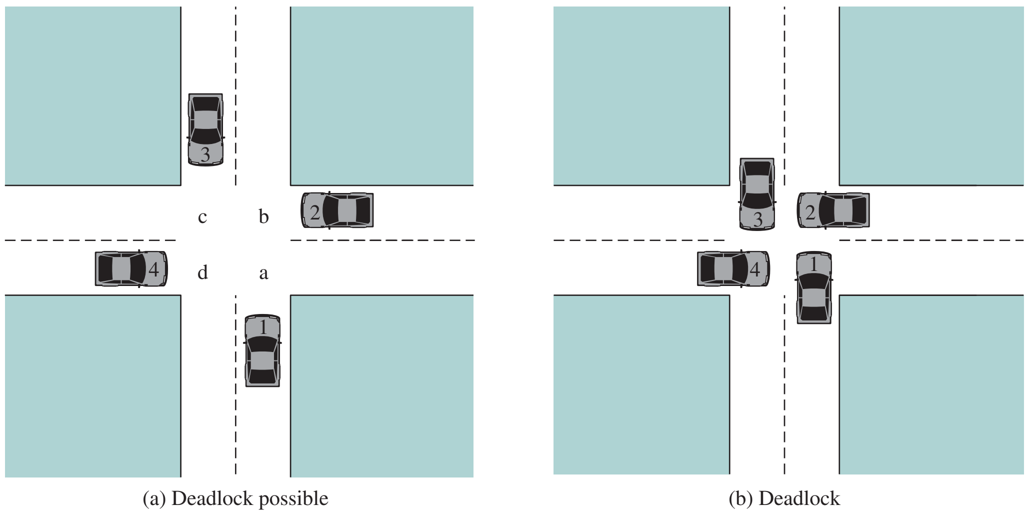

Figure 6.1 **Illustration of Deadlock**

All deadlocks involve conflicting needs for resources by two or more processes. A common example is the traffic deadlock. Figure 6.1a shows a situation in which four cars have arrived at a four-way stop intersection at approximately the same time. The four quadrants of the intersection are the resources over which control is needed. In particular, if all four cars wish to go straight through the intersection, the resource requirements are as follows:

- Car 1, traveling north, needs quadrants a and b.
- Car 2 needs quadrants b and c.
- Car 3 needs quadrants c and d.
- Car 4 needs quadrants d and a.

The rule of the road in the United States is that a car at a four-way stop should defer to a car immediately to its right. This rule works if there are only two or three cars at the intersection. For example, if only the northbound and westbound cars arrive at the intersection, the northbound car will wait and the westbound car proceeds. However, if all four cars arrive at about the same time and all four follow the rule, each will refrain from entering the intersection. This causes a potential deadlock. It is only a potential deadlock, because the necessary resources are available for any of the cars to proceed. If one car eventually chooses to proceed, it can do so.

However, if all four cars ignore the rules and proceed (cautiously) into the intersection at the same time, then each car seizes one resource (one quadrant) but cannot proceed because the required second resource has already been seized by another car. This is an actual deadlock.

Let us now look at a depiction of deadlock involving processes and computer resources. Figure 6.2 (based on one in [BACO03]), which we refer to as a **joint progress diagram**, illustrates the progress of two processes competing for two

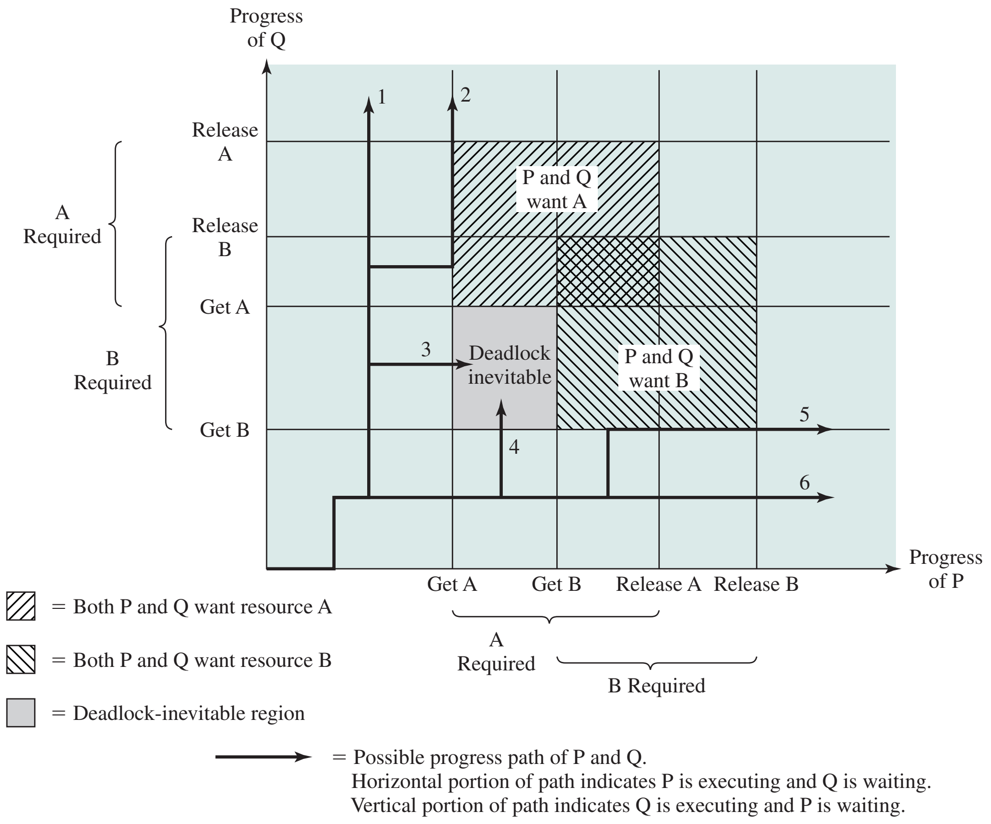
Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
Deadlock inevitable
P and Q want B
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required
= Both P and Q want resource A
= Both P and Q want resource B
= Deadlock-inevitable region
= Possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Figure 6.2 **Example of Deadlock**

resources. Each process needs exclusive use of both resources for a certain period of time. Two processes, P and Q, have the following general form:

Process P	**Process Q**
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

In Figure 6.2, the *x*-axis represents progress in the execution of P and the *y*-axis represents progress in the execution of Q. The joint progress of the two processes is therefore represented by a path that progresses from the origin in a northeasterly direction. For a uniprocessor system, only one process at a time may execute, and the path consists of alternating horizontal and vertical segments, with a horizontal segment representing a period when P executes and Q waits and a vertical segment representing a period when Q executes and P waits. The figure indicates areas in which both P and Q require resource A (upward slanted lines); both P and Q require resource B (downward slanted lines); and both P and Q require both resources. Because we assume that each process requires exclusive control of any resource, these are all forbidden regions; that is, it is impossible for any path representing the joint execution progress of P and Q to enter these regions.

The figure shows six different execution paths. These can be summarized as follows:

1. Q acquires B and then A and then releases B and A. When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
4. P acquires A and then Q acquires B. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
5. P acquires A and then B. Q executes and blocks on a request for B. P releases A and B. When Q resumes execution, it will be able to acquire both resources.
6. P acquires A and then B and then releases A and B. When Q resumes execution, it will be able to acquire both resources.

The gray-shaded area of Figure 6.2, which can be referred to as a **fatal region**, applies to the commentary on paths 3 and 4. If an execution path enters this fatal region, then deadlock is inevitable. Note that the existence of a fatal region depends on the logic of the two processes. However, deadlock is only inevitable if the joint progress of the two processes creates a path that enters the fatal region.

Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application. For example, suppose that P does not need both resources at the same time so that the two processes have the following form:

```
Process P        Process Q
• • •            • • •
Get A            Get B
• • •            • • •
Release A        Get A
• • •            • • •
Get B            Release B
• • •            • • •
Release B        Release A
• • •            • • •
```

This situation is reflected in Figure 6.3. Some thought should convince you that regardless of the relative timing of the two processes, deadlock cannot occur.

As shown, the joint progress diagram can be used to record the execution history of two processes that share resources. In cases where more than two processes

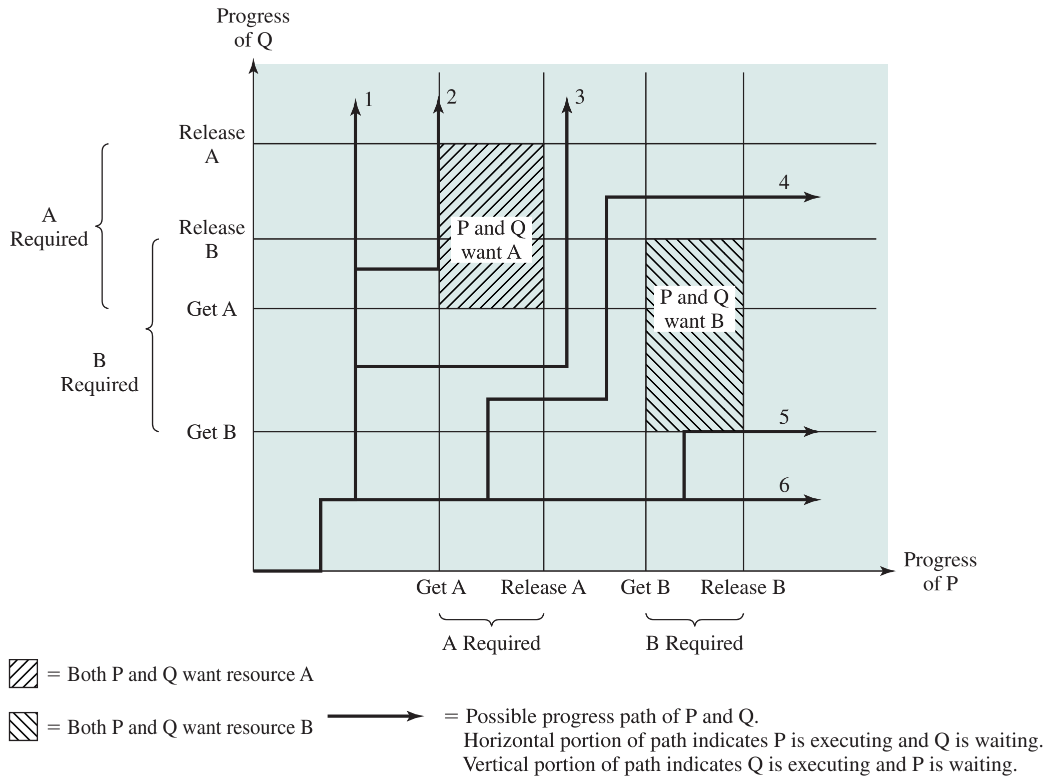
Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
P and Q want B
Progress of P
Get A
Release A
Get B
Release B
A Required
B Required
= Both P and Q want resource A
= Both P and Q want resource B
= Possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Figure 6.3 Example of No Deadlock [BACO03]

may compete for the same resource, a higher-dimensional diagram would be required. The principles concerning fatal regions and deadlock would remain the same.

Reusable Resources

Two general categories of resources can be distinguished: reusable and consumable. A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtain resource units that they later release for reuse by other processes. Examples of reusable resources include processors; I/O channels; main and secondary memory; devices; and data structures such as files, databases, and semaphores.

As an example of deadlock involving reusable resources, consider two processes that compete for exclusive access to a disk file D and a tape drive T. The programs engage in the operations depicted in Figure 6.4. Deadlock occurs if each process holds one resource and requests the other. For example, deadlock occurs if the multiprogramming system interleaves the execution of the two processes as follows:

$$p_0\ p_1\ q_0\ q_1\ p_2\ q_2$$

Step	Process P Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Step	Process Q Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

It may appear that this is a programming error rather than a problem for the OS designer. However, we have seen that concurrent program design is challenging. Such deadlocks do occur, and the cause is often embedded in complex program logic, making detection difficult. One strategy for dealing with such a deadlock is to impose system design constraints concerning the order in which resources can be requested.

Another example of deadlock with a reusable resource has to do with requests for main memory. Suppose the space available for allocation is 200 Kbytes, and the following sequence of requests occurs:

P1	P2
…	…
Request 80 Kbytes;	Request 70 Kbytes;
…	…
Request 60 Kbytes;	Request 80 Kbytes;

Deadlock occurs if both processes progress to their second request. If the amount of memory to be requested is not known ahead of time, it is difficult to deal with this type of deadlock by means of system design constraints. The best way to deal with this particular problem is, in effect, to eliminate the possibility by using virtual memory, which is discussed in Chapter 8.

Consumable Resources

A consumable resource is one that can be created (produced) and destroyed (consumed). Typically, there is no limit on the number of consumable resources of a particular type. An unblocked producing process may create any number of such resources. When a resource is acquired by a consuming process, the resource ceases to exist. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

As an example of deadlock involving consumable resources, consider the following pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1	P2
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);

Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received). Once again, a design error is the cause of the deadlock. Such errors may be quite subtle and difficult to detect. Furthermore, it may take a rare combination of events to cause the deadlock; thus a program

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

could be in use for a considerable period of time, even years, before the deadlock actually occurs.

There is no single effective strategy that can deal with all types of deadlock. Table 6.1 summarizes the key elements of the most important approaches that have been developed: prevention, avoidance, and detection. We examine each of these in turn, after first introducing resource allocation graphs and then discussing the conditions for deadlock.

Resource Allocation Graphs

A useful tool in characterizing the allocation of resources to processes is the **resource allocation graph**, introduced by Holt [HOLT72]. The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted (Figure 6.5a). Within a resource node, a dot is shown for each instance of that resource. Examples of resource types that may have multiple instances are I/O devices that are allocated by a resource management module in the OS. A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted (Figure 6.5b); that is, the process

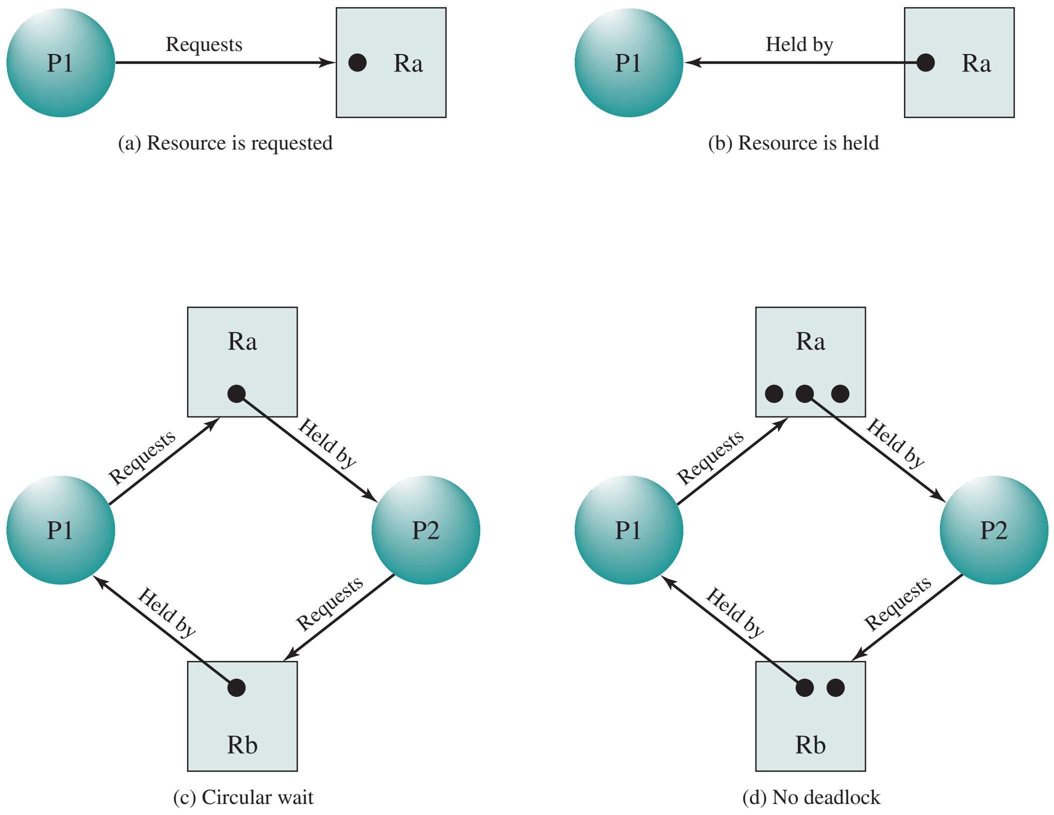

Figure 6.5 Examples of Resource Allocation Graphs

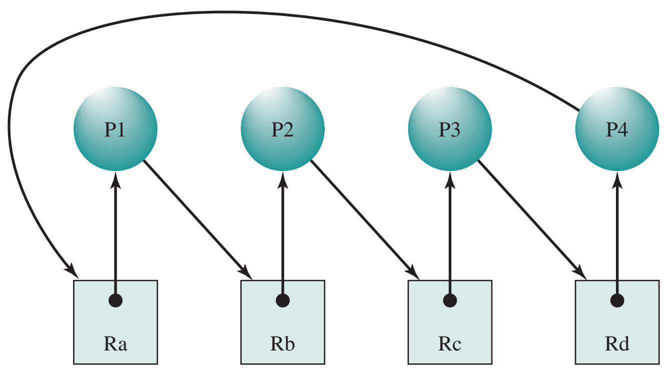

Figure 6.6 **Resource Allocation Graph for Figure 6.1b**

has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.

Figure 6.5c shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb. Figure 6.5d has the same topology as Figure 6.5c, but there is no deadlock because multiple units of each resource are available.

The resource allocation graph of Figure 6.6 corresponds to the deadlock situation in Figure 6.1b. Note that in this case, we do not have a simple situation in which two processes each have one resource the other needs. Rather, in this case, there is a circular chain of processes and resources that results in deadlock.

The Conditions for Deadlock

Three conditions of policy must be present for a deadlock to be possible:

1. **Mutual exclusion.** Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
2. **Hold and wait.** A process may hold allocated resources while awaiting assignment of other resources.
3. **No preemption.** No resource can be forcibly removed from a process holding it.

In many ways these conditions are quite desirable. For example, mutual exclusion is needed to ensure consistency of results and the integrity of a database. Similarly, preemption should not be done arbitrarily. For example, when data resources are involved, preemption must be supported by a rollback recovery mechanism, which restores a process and its resources to a suitable previous state from which the process can eventually repeat its actions.

The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required:

4. **Circular wait.** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain (e.g., Figure 6.5c and Figure 6.6).

The fourth condition is, actually, a potential consequence of the first three. That is, given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait. The unresolvable circular wait is in fact the definition of deadlock. The circular wait listed as condition 4 is unresolvable because the first three conditions hold. Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock.[1]

To clarify this discussion, it is useful to return to the concept of the joint progress diagram, such as the one shown in Figure 6.2. Recall that we defined a fatal region as one such that once the processes have progressed into that region, those processes will deadlock. A fatal region exists only if all of the first three conditions listed above are met. If one or more of these conditions are not met, there is no fatal region and deadlock cannot occur. Thus, these are necessary conditions for deadlock. For deadlock to occur, there must not only be a fatal region, but also a sequence of resource requests that has led into the fatal region. If a circular wait condition occurs, then in fact the fatal region has been entered. Thus, all four conditions listed above are sufficient for deadlock. To summarize,

Possibility of Deadlock	**Existence of Deadlock**
1. Mutual exclusion	**1.** Mutual exclusion
2. No preemption	**2.** No preemption
3. Hold and wait	**3.** Hold and wait
	4. Circular wait

Three general approaches exist for dealing with deadlock. First, one can **prevent** deadlock by adopting a policy that eliminates one of the conditions (conditions 1 through 4). Second, one can **avoid** deadlock by making the appropriate dynamic choices based on the current state of resource allocation. Third, one can attempt to **detect** the presence of deadlock (conditions 1 through 4 hold) and take action to recover. We discuss each of these approaches in turn.

6.2 DEADLOCK PREVENTION

The strategy of deadlock prevention is, simply put, to design a system in such a way that the possibility of deadlock is excluded. We can view deadlock prevention methods as falling into two classes. An indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3). A direct method of deadlock prevention is to prevent the occurrence of a circular wait (item 4). We now examine techniques related to each of the four conditions.

[1]Virtually all textbooks simply list these four conditions as the conditions needed for deadlock, but such a presentation obscures some of the subtler issues. Item 4, the circular wait condition, is fundamentally different from the other three conditions. Items 1 through 3 are policy decisions, while item 4 is a circumstance that might occur depending on the sequencing of requests and releases by the involved processes. Linking circular wait with the three necessary conditions leads to inadequate distinction between prevention and avoidance. See [SHUB90] and [SHUB03] for a discussion.

Mutual Exclusion

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS. Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. Even in this case, deadlock can occur if more than one process requires write permission.

Hold and Wait

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require.

There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

No Preemption

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources. This latter scheme would prevent deadlock only if no two processes possessed the same priority.

This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

Circular Wait

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.

To see that this strategy works, let us associate an index with each resource type. Then resource R_i precedes R_j in the ordering if $i < j$. Now suppose that two processes, A and B, are deadlocked because A has acquired R_i and requested R_j, and B has acquired R_j and requested R_i. This condition is impossible because it implies $i < j$ and $j < i$.

As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

6.3 DEADLOCK AVOIDANCE

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance.[2] In **deadlock prevention**, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. **Deadlock avoidance**, on the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests.

In this section, we describe two approaches to deadlock avoidance:

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Process Initiation Denial

Consider a system of n processes and m different types of resources. Let us define the following vectors and matrices:

Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

The matrix Claim gives the maximum requirement of each process for each resource, with one row dedicated to each process. This information must be

[2]The term *avoidance* is a bit confusing. In fact, one could consider the strategies discussed in this section to be examples of deadlock prevention because they indeed prevent the occurrence of a deadlock.

declared in advance by a process for deadlock avoidance to work. Similarly, the matrix Allocation gives the current allocation to each process. The following relationships hold:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j	All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i,j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i,j	No process is allocated more resources of any type than the process originally claimed to need.

With these quantities defined, we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock. Start a new process P_{n+1} only if

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$$

That is, a process is only started if the maximum claim of all current processes plus those of the new process can be met. This strategy is hardly optimal, because it assumes the worst: that all processes will make their maximum claims together.

Resource Allocation Denial

The strategy of resource allocation denial, referred to as the **banker's algorithm**,[3] was first proposed in [DIJK65]. Let us begin by defining the concepts of state and safe state. Consider a system with a fixed number of processes and a fixed number of resources. At any time a process may have zero or more resources allocated to it. The **state** of the system reflects the current allocation of resources to processes. Thus, the state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation, defined earlier. A **safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion). An **unsafe state** is, of course, a state that is not safe.

The following example illustrates these concepts. Figure 6.7a shows the state of a system consisting of four processes and three resources. The total amount of resources R1, R2, and R3 are 9, 3, and 6 units, respectively. In the current state allocations have been made to the four processes, leaving 1 unit of R2

[3]Dijkstra used this name because of the analogy of this problem to one in banking, with customers who wish to borrow money corresponding to processes and the money to be borrowed corresponding to resources. Stated as a banking problem, the bank has a limited reserve of money to lend and a list of customers, each with a line of credit. A customer may choose to borrow against the line of credit a portion at a time, and there is no guarantee that the customer will make any repayment until after having taken out the maximum amount of loan. The banker can refuse a loan to a customer if there is a risk that the bank will have insufficient funds to make further loans that will permit the customers to repay eventually.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

Figure 6.7 Determination of a Safe State

and 1 unit of R3 available. Is this a safe state? To answer this question, we ask an intermediate question: Can any of the four processes be run to completion with the resources available? That is, can the difference between the maximum requirement and current allocation for any process be met with the available resources? In terms of the matrices and vectors introduced earlier, the condition to be met for process *i* is:

$$C_{ij} - A_{ij} \leq V_j, \quad \text{for all } j$$

Clearly, this is not possible for P1, which has only 1 unit of R1 and requires 2 more units of R1, 2 units of R2, and 2 units of R3. However, by assigning one unit of R3 to process P2, P2 has its maximum required resources allocated and can run to completion. Let us assume that this is accomplished. When P2 completes, its resources can be returned to the pool of available resources. The resulting state is shown in Figure 6.7b. Now we can ask again if any of the remaining processes can be completed. In this case, each of the remaining processes could be completed. Suppose we choose P1, allocate the required resources, complete P1, and return all of P1's resources to the available pool. We are left in the state shown in Figure 6.7c. Next, we can complete P3, resulting in the state of Figure 6.7d. Finally, we can complete P4. At this point, all of the processes have been run to completion. Thus, the state defined by Figure 6.7a is a safe state.

These concepts suggest the following deadlock avoidance strategy, which ensures that the system of processes and resources is always in a safe state. When a process makes a request for a set of resources, assume that the request is granted, update the system state accordingly, and then determine if the result is a safe state. If so, grant the request and, if not, block the process until it is safe to grant the request.

Consider the state defined in Figure 6.8a. Suppose P2 makes a request for one additional unit of R1 and one additional unit of R3. If we assume the request is granted, then the resulting state is that of Figure 6.7a. We have already seen that this is a safe state; therefore, it is safe to grant the request. Now let us return to the state of Figure 6.8a and suppose that P1 makes the request for one additional unit each of R1 and R3; if we assume that the request is granted, we are left in the state of Figure 6.8b. Is this a safe state? The answer is no, because each process will need at least one additional unit of R1, and there are none available. Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

It is important to point out that Figure 6.8b is not a deadlocked state. It merely has the potential for deadlock. It is possible, for example, that if P1 were run from this state it would subsequently release one unit of R1 and one unit of R3 prior to needing these resources again. If that happened, the system would return to a safe state. Thus, the deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State

Figure 6.9 gives an abstract version of the deadlock avoidance logic. The main algorithm is shown in part (b). With the state of the system defined by the data structure `state`, `request[*]` is a vector defining the resources requested by process *i*. First, a check is made to assure that the request does not exceed the original claim of the process. If the request is valid, the next step is to determine if it is possible to fulfill the request (i.e., there are sufficient resources available). If it is not possible, then the process is suspended. If it is possible, the final step is to determine if it is safe to fulfill the request. To do this, the resources are tentatively assigned to process *i* to form `newstate`. Then a test for safety is made using the algorithm in Figure 6.9c.

Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection, and is less restrictive than deadlock prevention. However, it does have a number of restrictions on its use:

- The maximum resource requirement for each process must be stated in advance.
- The processes under consideration must be independent; that is, the order in which they execute must be unconstrained by any synchronization requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) Global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     <error>;                          /* total request > claim*/
else if (request [*] > available [*])
     <suspend process>;
else  {                                      /* simulate alloc */
     <define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*]>;
}
if (safe (newstate))
     <carry out allocation>;
else {
     <restore original state>;
     <suspend process>;
}
```

(b) Resource alloc algorithm

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
       <find a process Pk in rest such that
          claim [k,*] - alloc [k,*]<= currentavail;
       if (found) {               /* simulate execution of Pk */
          currentavail = currentavail + alloc [k,*];
          rest = rest - {Pk};
       }
       else possible = false;
    }
    return (rest == null);
}
```

(c) Test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

6.4 DEADLOCK DETECTION

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes. At the opposite extreme, deadlock detection strategies do not limit resource access or restrict process actions. With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the OS performs an algorithm that allows it to detect the circular wait condition described earlier in condition (4) and illustrated in Figure 6.6.

Deadlock Detection Algorithm

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system. On the other hand, such frequent checks consume considerable processor time.

A common algorithm for deadlock detection is one described in [COFF71]. The Allocation matrix and Available vector described in the previous section are used. In addition, a request matrix **Q** is defined such that *Qij* represents the amount of resources of type *j* requested by process *i*. The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index *i* such that process *i* is currently unmarked and the *i*th row of **Q** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process *i* and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

A deadlock exists if and only if there are unmarked processes at the end of the algorithm. Each unmarked process is deadlocked. The strategy in this algorithm is to find a process whose resource requests can be satisfied with the available resources, and then assume that those resources are granted and that the process runs to completion and releases all of its resources. The algorithm then looks for another process to satisfy. Note that this algorithm does not guarantee to prevent deadlock; that will depend on the order in which future requests are granted. All that it does is determine if deadlock currently exists.

We can use Figure 6.10 to illustrate the deadlock detection algorithm. The algorithm proceeds as follows:

1. Mark P4, because P4 has no allocated resources.
2. Set **W** = (0 0 0 0 1).

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Figure 6.10 Example for Deadlock Detection

3. The request of process P3 is less than or equal to **W**, so mark P3 and set

$$\mathbf{W} = \mathbf{W} + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1).$$

4. No other unmarked process has a row in **Q** that is less than or equal to **W**. Therefore, terminate the algorithm.

The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.

Recovery

Once deadlock has been detected, some strategy is needed for recovery. The following are possible approaches, listed in order of increasing sophistication:

1. Abort all deadlocked processes. This is, believe it or not, one of the most common, if not the most common, solution adopted in operating systems.
2. Back up each deadlocked process to some previously defined checkpoint, and restart all processes. This requires that rollback and restart mechanisms be built in to the system. The risk in this approach is that the original deadlock may recur. However, the nondeterminancy of concurrent processing may ensure that this does not happen.
3. Successively abort deadlocked processes until deadlock no longer exists. The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost. After each abortion, the detection algorithm must be reinvoked to see whether deadlock still exists.
4. Successively preempt resources until deadlock no longer exists. As in (3), a cost-based selection should be used, and reinvocation of the detection algorithm is required after each preemption. A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

For (3) and (4), the selection criteria could be one of the following. Choose the process with the

- least amount of processor time consumed so far
- least amount of output produced so far
- most estimated time remaining

- least total resources allocated so far
- lowest priority

Some of these quantities are easier to measure than others. Estimated time remaining is particularly suspect. Also, other than by means of the priority measure, there is no indication of the "cost" to the user, as opposed to the cost to the system as a whole.

6.5 AN INTEGRATED DEADLOCK STRATEGY

As Table 6.1 suggests, there are strengths and weaknesses to all of the strategies for dealing with deadlock. Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations. [HOWA73] suggests one approach:

- Group resources into a number of different resource classes.
- Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes.
- Within a resource class, use the algorithm that is most appropriate for that class.

As an example of this technique, consider the following classes of resources:

- **Swappable space:** Blocks of memory on secondary storage for use in swapping processes
- **Process resources:** Assignable devices, such as tape drives, and files
- **Main memory:** Assignable to processes in pages or segments
- **Internal resources:** Such as I/O channels

The order of the preceding list represents the order in which resources are assigned. The order is a reasonable one, considering the sequence of steps that a process may follow during its lifetime. Within each class, the following strategies could be used:

- **Swappable space:** Prevention of deadlocks by requiring that all of the required resources that may be used be allocated at one time, as in the hold-and-wait prevention strategy. This strategy is reasonable if the maximum storage requirements are known, which is often the case. Deadlock avoidance is also a possibility.
- **Process resources:** Avoidance will often be effective in this category, because it is reasonable to expect processes to declare ahead of time the resources that they will require in this class. Prevention by means of resource ordering within this class is also possible.
- **Main memory:** Prevention by preemption appears to be the most appropriate strategy for main memory. When a process is preempted, it is simply swapped to secondary memory, freeing space to resolve the deadlock.
- **Internal resources:** Prevention by means of resource ordering can be used.

6.6 DINING PHILOSOPHERS PROBLEM

We now turn to the dining philosophers problem, introduced by Dijkstra [DIJK71]. Five philosophers live in a house, where a table is laid for them. The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.

The eating arrangements are simple (Figure 6.11): a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti. The problem: Devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation (in this case, the term has literal as well as algorithmic meaning!).

This problem may not seem important or relevant in itself. However, it does illustrate basic problems in deadlock and starvation. Furthermore, attempts to develop solutions reveal many of the difficulties in concurrent programming (e.g., see [GING90]). In addition, the dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources, which may

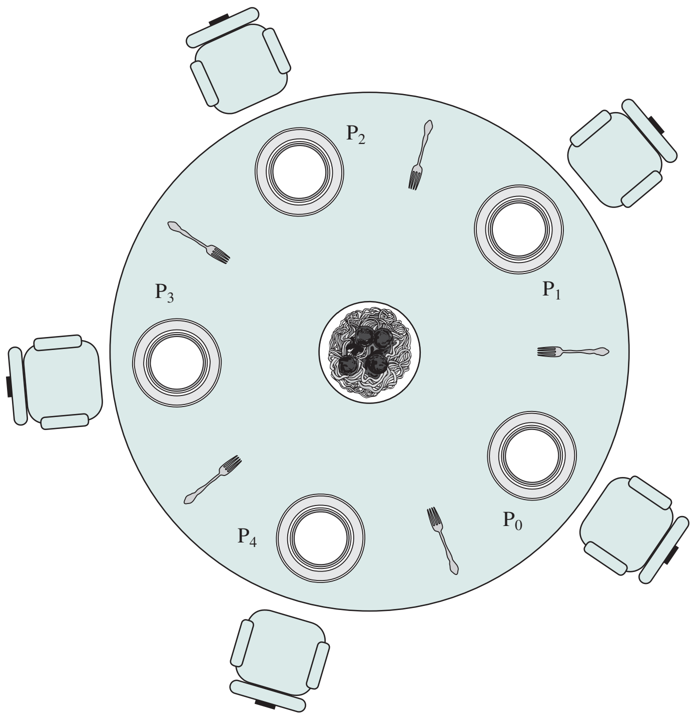

Figure 6.11 Dining Arrangement for Philosophers

occur when an application includes concurrent threads of execution. Accordingly, this problem is a standard test case for evaluating approaches to synchronization.

Solution Using Semaphores

Figure 6.12 suggests a solution using semaphores. Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table. This solution, alas, leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this undignified position, all philosophers starve.

To overcome the risk of deadlock, we could buy five additional forks (a more sanitary solution!) or teach the philosophers to eat spaghetti with just one fork. As another approach, we could consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks. Figure 6.13 shows such a solution, again using semaphores. This solution is free of deadlock and starvation.

Solution Using a Monitor

Figure 6.14 shows a solution to the dining philosophers problem using a monitor. A vector of five condition variables is defined, one condition variable per fork. These condition variables are used to enable a philosopher to wait for the availability of a fork. In addition, there is a Boolean vector that records the availability status of each fork (`true` means the fork is available). The monitor consists of two procedures. The `get_forks` procedure is used by a philosopher to seize his or her left and

```
/* program   diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1),
          philosopher (2),    philosopher (3),
          philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

```
/* program   diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (room);
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal (fork [(i+1) mod 5]);
          signal (fork[i]);
          signal (room);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1),
          philosopher (2), philosopher (3),
          philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

right forks. If either fork is unavailable, the philosopher process is queued on the appropriate condition variable. This enables another philosopher process to enter the monitor. The `release-forks` procedure is used to make two forks available. Note that the structure of this solution is similar to that of the semaphore solution proposed in Figure 6.12. In both cases, a philosopher seizes first the left fork and then the right fork. Unlike the semaphore solution, this monitor solution does not suffer from deadlock, because only one process at a time may be in the monitor. For example, the first philosopher process to enter the monitor is guaranteed that it can pick up the right fork after it picks up the left fork before the next philosopher to the right has a chance to seize its left fork, which is this philosopher's right fork.

6.7 UNIX CONCURRENCY MECHANISMS

UNIX provides a variety of mechanisms for interprocessor communication and synchronization. Here, we look at the most important of these:

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

1 Principles of Deadlock

A deadlock is the permanent blockig of a set of process that are competing for a resource or communicating with each other. Basically processes are all waiting on each other and can never continue execution. We tend to depict deadlocks using a joint progress diagram.

In a joint progress diagram we show potential paths of execution (turning 90 degress to depict wait times). By filling in sections where both processes need the same resource we can narrow down the possible execution paths. If there exists an area where there is no exits (blocked by forbidden regions) any process that enters it will be deadlocked. We call the region of execution the fatal region. If there is no fatal region a deadlock is impossible.

Joint process diagrams only apply in the case of reusable resources. When a resource is consumable the process that gets to it first removes it (consumes it) so another process cannot have it until it has been added to.

Reusable resources are held by processes for periods of time which is what causes deadlocks. One such example is two processes with multiple requests for memory and no frees. A deadlock is reached when we run out of memory and neither process can release any because it is blocked.

Consumable resources are produced and consumed so we can get blocked when we need to consume a resource and there is none. A deadlock occurs when two processes both consume a resource and then produce it. This means that both are blocked and waiting on the other to produce the needed resource.

Approaches to dealing with deadlocks:

- Prevention: under commits resources
  - request all resources at start
    - works well for small processes and preemption is not neccissary
    - inefficient, we dont always know resources needed, delays initialization
  - preemption
    - convenient when used on something whose stat is easily saved
    - preempts too often
  - resource ordering
    - easy to enforce with compile time checks, no runtime overhead
    - doesnt allow incremental resource allocation
- Avoidance
  - manipulate to find a safe path of execution
    - no preemption needed
    - future resource requests must be known, processes can be blocked for a while
- Detection: allocated resources wherever possible
  - invoke periodically to test for deadlocks
    - never delays initialization and allows online handling

    * preemption losses

1.1 Resource Allocation Graphs

A directed graph showing the state of the system's resources and processes. Circle nodes are processes and square nodes are resources. When the edge points at the resource its a request and when it points at the process its a held. Dots are used to denote multiple instances of a resource. These make it easy to spot deadlocks since we know immediately which process are blocked and what they are blocked on.

1.2 Conditions for Deadlocks

In order to have a deadlock you need:

- mutual exclusion: if no resource is blocking we can all access it and never get locked
- hold and wait: if processes can only have one resource at a time they must release before another process runs, so no one gets blocked
- no preemption: if we can forcibly take a resource from another process we would be able to break any deadlocks
- circular wait (this is the one that actually causes the deadlock): we have to have a circle of processes waiting on each other to have a deadlock

The first three conditions make the fourth one unresolvable which is what makes a deadlock. Prevention is the policy of breaking one of the above conditions to stop deadlocks.

2 Deadlock Prevention

This is a system where we design everything such that there is no possibility of deadlocks. Usually done excluding one of the conditions required for deadlocks. The direct method is to remove all circular waits.

Mutual exclusion: this is for the most part necessary for nearly all programs so we cannot exclude it.

Hold and Wait: we can prevent this by requiring that a process request all of its resources one at a time and block the process until all requests can be fulfilled at the same time. This is inefficient because we must wait on all resources when we could proceed with some of them, and resources may remain unused for large periods of time but are considered held so other processes cannot use them. We also run into problems when we can't know what resources will be needed at the start of a process and when threads come into play.

No Preemption: we can prevent this by requiring that if a process is holding certain resources it is denied a further request until it releases those or allow a second process to preempt and force the first process to dump its resources (only works for distinct priorities). Only really works for resources where we can save and restore the state easily.

Circular Wait: we can prevent this by defining a linear ordering of resources.

3 Deadlock Avoidance

This doesnt have the inherent inefficiencies and restrictions as Deadlock Prevention. The trade off is here we need to now about the future executions of each process in order to make the bes decision to maintain concurrency.

3.1 Process Initiation Denial

Say we have n processes and m resources. We defined a:

- R = resource vector = total amount of each resource in the system
- V = available vecotr = free amount of each resource
- C = claim matrix = requirement of process for resource
- A = allocation matrix = current allocation to process of resource

The claim matrix is the max requirement of a process which we would need to know at the start which doesnt always happen.

From this arises some relationships:

- $R_j = V_j + \sum_{i=1}^{n} A_{ij}$ all resources are either available or allocated
- $C_{ij} \leq R_j$ no process can claim more resource than there exists
- $A_{ij} \leq C_{ij}$ no process can be allocated more memory that it claimed

From these we can say that you may only start a process P_{n+1} if:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \text{ for all j}$$

Which basically means that there must be enough available resources left after all other claims have been filled to fulfill the claims made by this process. This is called **the banker's algorithm**.

A **safe state** is when a there is at least one sequence of resource allocation to a process that does not result in a deadlock.

Running through the algorithm:

1. Get your claim and allocation matrix, subtract them to get the current needs
2. find a process where there is enough resources in the available vector to satisfy its current needs
3. remove those resources
4. run to completion
5. set its row in the claim matrix to zero (its done and doesnt need anything) and add those removed resources to the available vector

If ever you reach a point where no process can run you have hit a deadlock.

We can do a similar process for only a few actions at a time and not run the process to completion. Here we just take from the available queue and dont return any unless specified. This only identifies the possibility of a deadlock since we cannot predict what the next lines of execution will be (they could free resources we need).

Deadlock avoidance does have some restrictions:

- The maximum resource requirement for each process must be stated in advance.
- The processes under consideration must be independent; that is, the order in which they execute must be unconstrained by any synchronization requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.

4 Deadlock Detection

Here we grant resources whenever we can and once in a while the OS runs a detections algorithm that kills any circular waits. The interval between checks varies based on implementation.

The most common detection algorithm uses the allocation matrix and available vector from the banker's algorithm and it adds a Q matrix representing the amount of resources requested by a process at that moment. As it runs it:

1. mark each process that whose row in the allocation matrix of zeros (this is basically marking processes that have already run to completion)
2. initialize a temporary vector W equal to the available vector
3. find a process that is not marked and whose row in Q fits in the available vector, if no such vector is found terminate
4. mark the first process whose Q row fits in V and add its allocation row to W
5. loop back to three

Here a deadlock exists if there are unmarked processes at the end of the algorithm. We work by finding a process that can run, simulating its run to completion and releasing of resources, very similar to banker's algorithm

Once we find a deadlock there are a few ways to fix it:

- kill all deadlocked processes (most common method)
- back up each deadlocked process to some checkpoint and restart them all (may get the deadlock again, but nondeterminancy of concurrent processing can ensure that doesnt happen)
- abort deadlocked processes one at a time until the deadlock is broken, go in order of minimal cost.
- keep preempting resources until the deadlock is broken in order of minimal cost

Two of the above methods require you to chose a process to minimize cost, this can be by:

- least processor time consumed
- least output produced
- most estimated time remaining

- least total resources allocated
- lowest priority

Basically we want to kill the restart the process that has done the least work so far.

5 An Integrated Deadlock Strategy

We want to mix and match design strategies for dealing with deadlocks to achieve the most optimal OS.

- group resources into classes
- linearly order classes
- within each class apply the most appropriate algorithm

Some example groups:

- **swappable space**: blocks of memory for swapping processes
  - Prevention of hold and wait: require all required resources to be allocated at the start since the maximum requirements is known
- **process resources**
  - Avoidance: it is reasonable to expect processes to declare ahead of time what devices they will be using
- **main memory**
  - Prevention by preemption: when a process is preempted it is just swapped for main memory which frees space to solve deadlock
- **internal resources**: I/O channels and such
  - Prevention by resource ordering

6 Dining Philosopher Problem

We have five philosopher each want to eat spaghetti with two forks, but there is one fork between each philosopher.

We could solve this using semaphores: each philosopher picks up the fork on their left and waits for the fork on their right to be available. This would lead to a deadlock if all philosophers wanted to eat at the same time. We add a semaphore that only allows four philosophers in the dinning room at the same time this way at least one philosopher can eat.

```
semaphore fork [5] = {1};
int i;
void philosopher (int i){
  while (true) {
    think();
    wait (room);
    wait (fork[i]);
    wait (fork [(i+1) mod 5]);
    eat();
    signal (fork [(i+1) mod 5]);
    signal (fork[i]);
    signal (room);
  }
}
```

We could use a monitor: we have one condition variable for each fork and have a get forks functions that only allows you to pick up forks if two are available

---

```
cond ForkReady[5];
/* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */
void get_forks(int pid) /* pid is the philosopher id number
{
  int left = pid;
  int right = (++pid) % 5;
  /*grant the left fork*/
  if (!fork(left)
    cwait(ForkReady[left]); /* queue on condition variable */
  fork(left) = false;
  /*grant the right fork*/
  if (!fork(right)
    cwait(ForkReady(right); /* queue on condition variable */
  fork(right) = false:
}
void release_forks(int pid)
{
  int left = pid;
  int right = (++pid) % 5;
  /*release the left fork*/
  if (empty(ForkReady[left])/*no one is waiting for this fork */
    fork(left) = true;
  else
    /* awaken a process waiting on this fork
    csignal(ForkReady[left]);
  /*release the right fork*/
  if (empty(ForkReady[right])/*no one is waiting for this fork */
    fork(right) = true;
  else
    /* awaken a process waiting on this fork */
    csignal(ForkReady[right]);
}
```
