CHAPTER 6

CONCURRENCY: DEADLOCK AND STARVATION

6.1 **Principles of Deadlock**
- Reusable Resources
- Consumable Resources
- Resource Allocation Graphs
- The Conditions for Deadlock

6.2 **Deadlock Prevention**
- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

6.3 **Deadlock Avoidance**
- Process Initiation Denial
- Resource Allocation Denial

6.4 **Deadlock Detection**
- Deadlock Detection Algorithm
- Recovery

6.5 **An Integrated Deadlock Strategy**

6.6 **Dining Philosophers Problem**
- Solution Using Semaphores
- Solution Using a Monitor

6.7 **UNIX Concurrency Mechanisms**
- Pipes
- Messages
- Shared Memory
- Semaphores
- Signals

6.8 **Linux Kernel Concurrency Mechanisms**
- Atomic Operations
- Spinlocks
- Semaphores
- Barriers

6.9 **Solaris Thread Synchronization Primitives**
- Mutual Exclusion Lock
- Semaphores
- Readers/Writer Lock
- Condition Variables

6.10 **Windows Concurrency Mechanisms**
- Wait Functions
- Dispatcher Objects
- Critical Sections
- Slim Read-Writer Locks and Condition Variables

6.11 **Summary**

6.12 **Recommended Reading**

6.13 **Key Terms, Review Questions, and Problems**

This chapter continues our survey of concurrency by looking at 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.

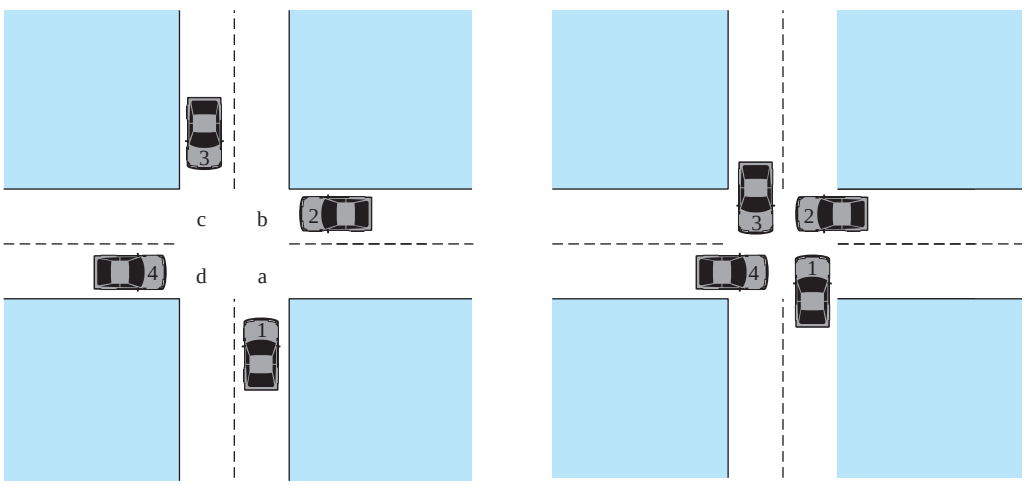
As with Chapter 5, the discussion in this chapter is limited to a consideration of concurrency and deadlock on a single system. Measures to deal with distributed deadlock problems are assessed in Chapter 18.

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.

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.



(a) Deadlock possible (b) Deadlock

Figure 6.1 Illustration of Deadlock

- Car 3 needs quadrants c and d.
- Car 4 needs quadrants d and a.

The typical 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, each will refrain from entering the intersection, this causes a potential deadlock. The deadlock is only potential, not actual, because the necessary resources are available for any of the cars to proceed. If one car eventually does 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

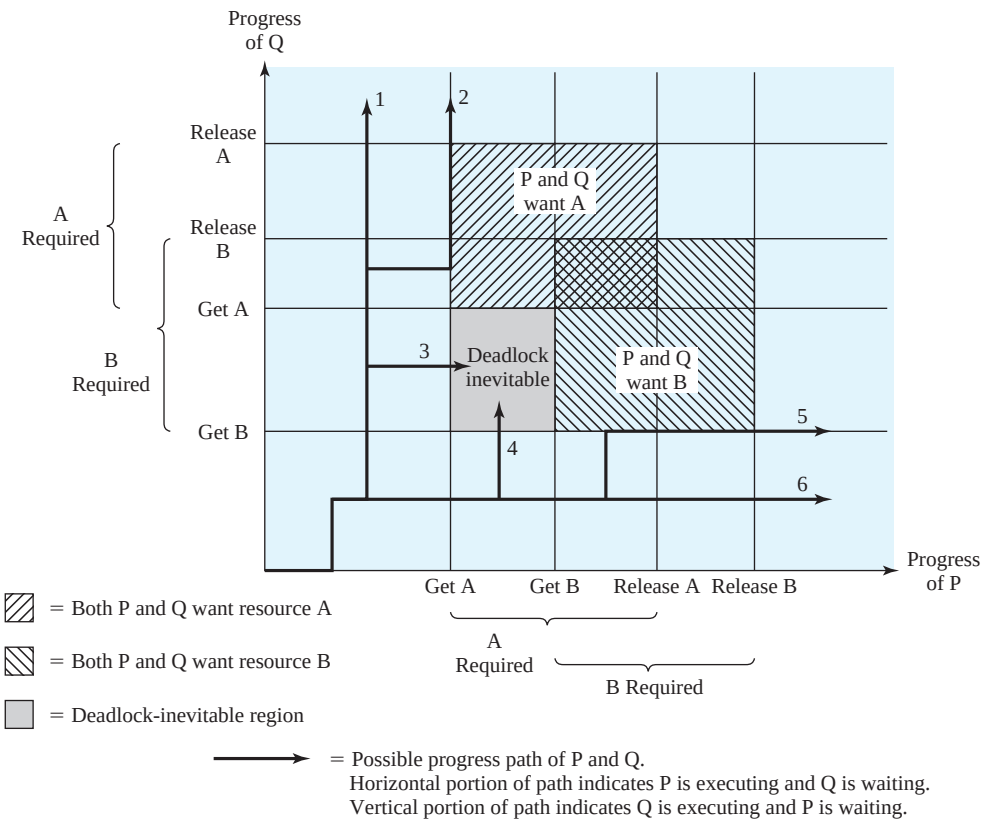


Figure 6.2 Example of Deadlock

two 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 may compete for the same resource, a higher-dimensional diagram

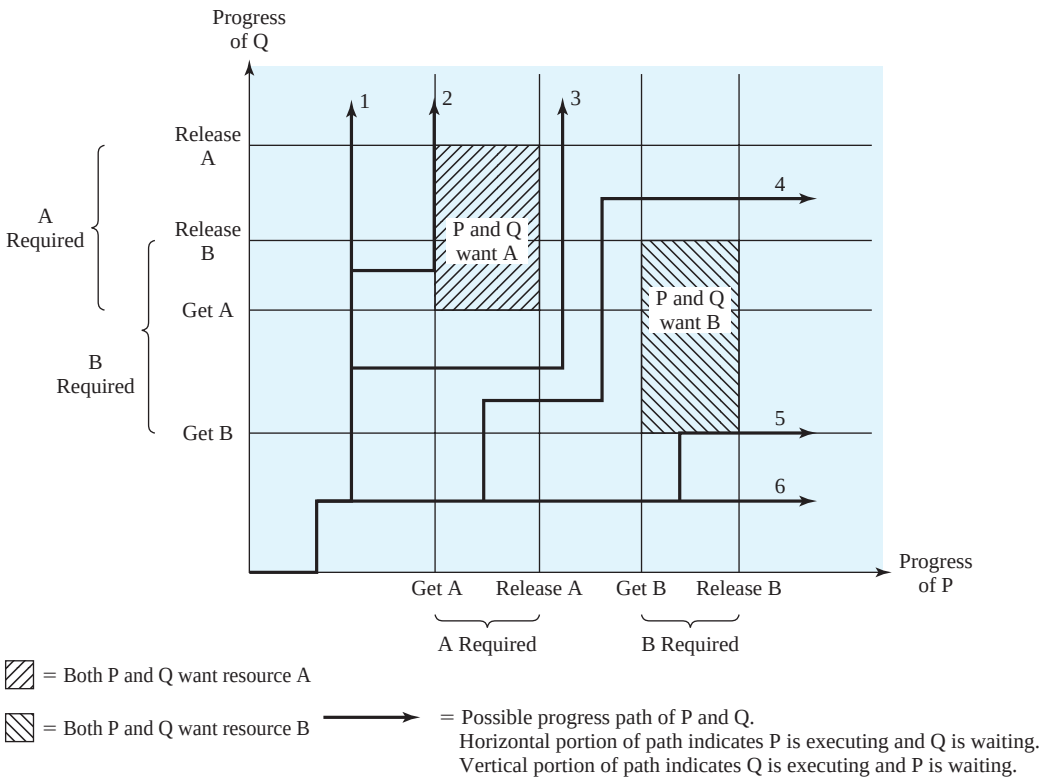


Figure 6.3 Example of No Deadlock [BACO03]

Process P	
Step	**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)

Process Q	
Step	**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**

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:

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

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 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

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

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 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.

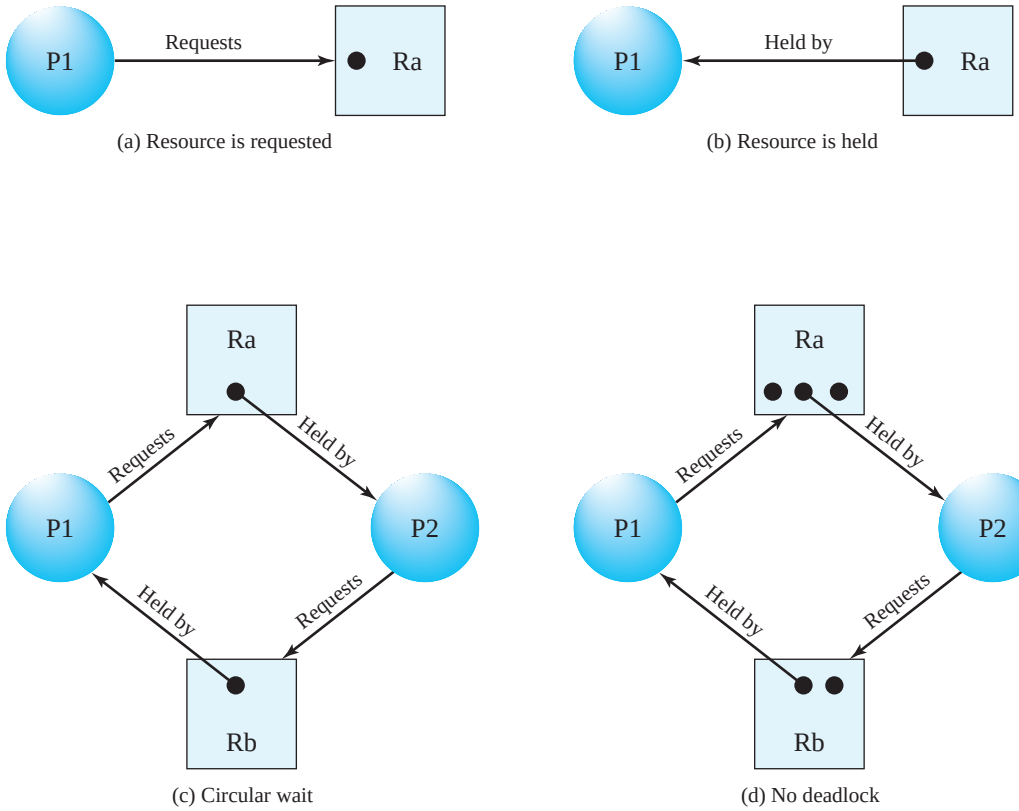


Figure 6.5 **Examples of Resource Allocation Graphs**

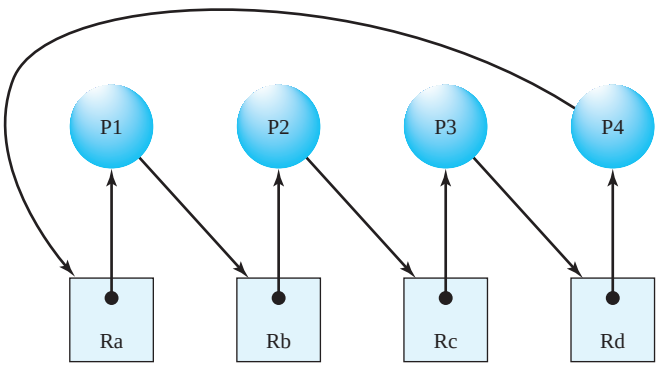


Figure 6.6 **Resource Allocation Graph for Figure 6.1b**

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 on 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

[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.

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.

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.

[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.

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 = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	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 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

Animation: Banker's Algorithm

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 and 1 unit of R3 available. The question is: 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.

---

[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.

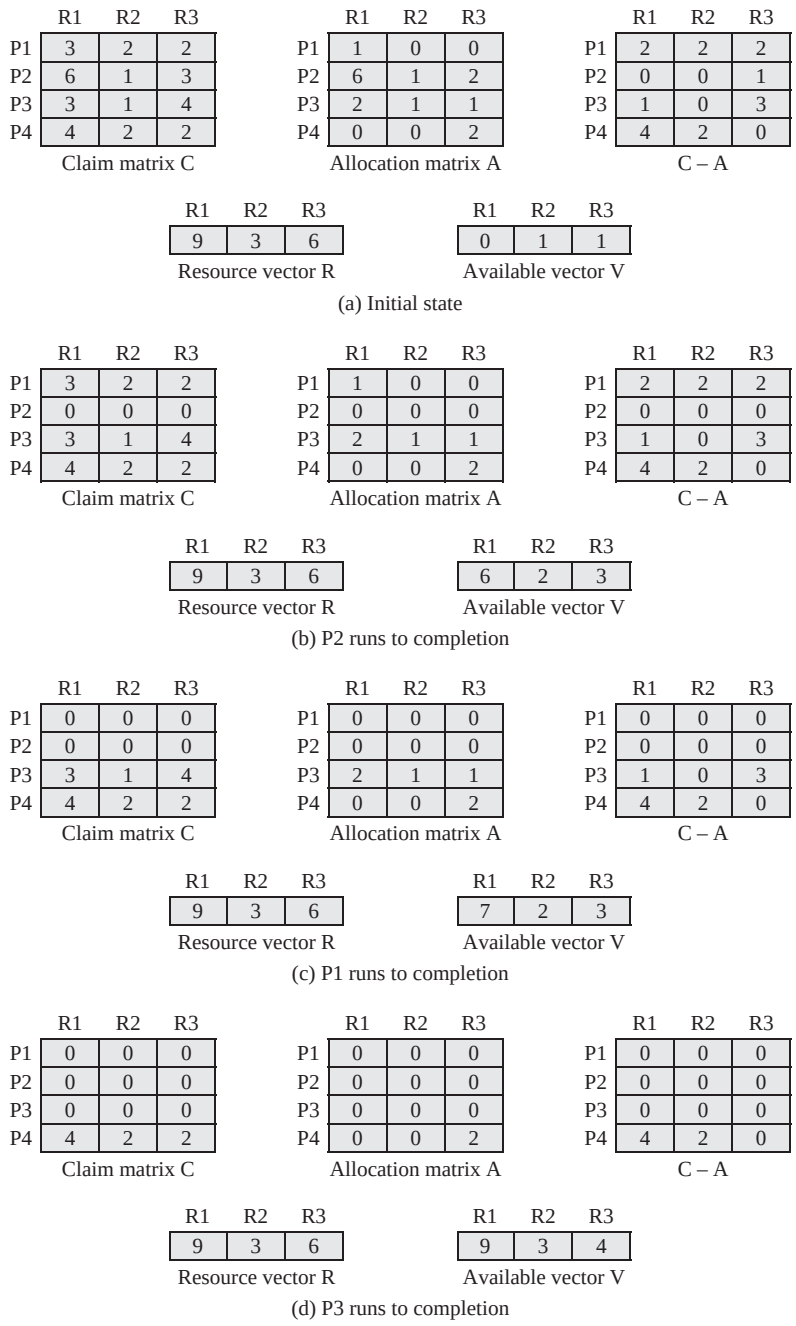


Figure 6.7 **Determination of a Safe State**

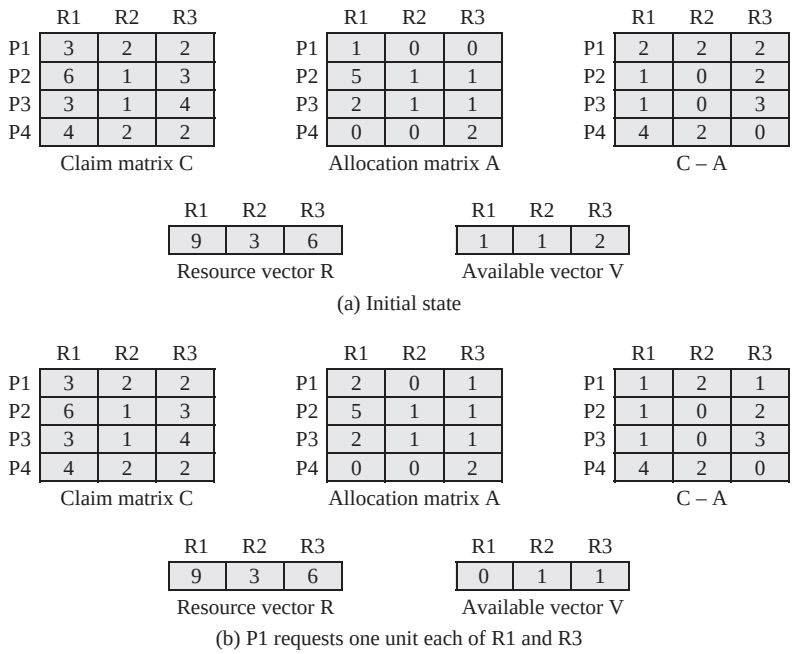


Figure 6.8 **Determination of an Unsafe State**

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.

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

```
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**

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.

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 Q_{ij} 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 ith 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

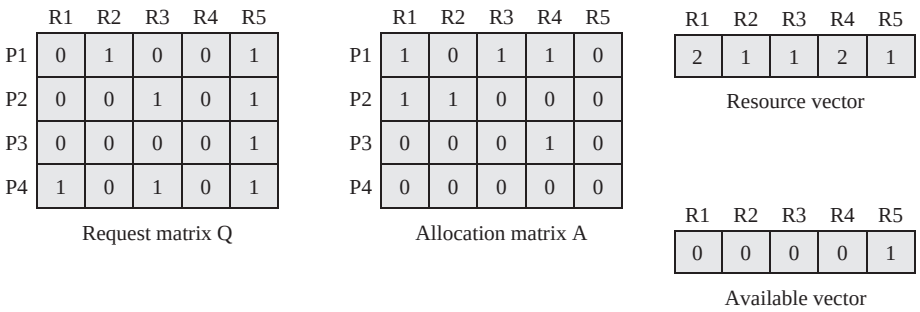


Figure 6.10 **Example for Deadlock Detection**

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

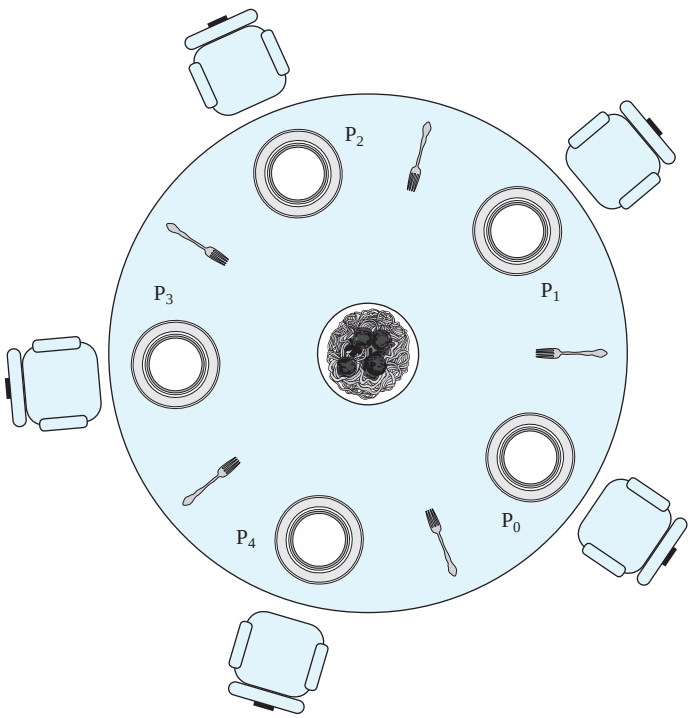


Figure 6.11 Dining Arrangement for Philosophers

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 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.

```
/* 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**

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 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

```
monitor dining_controller;
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]);
}
```

```
void philosopher[k=0 to 4]   /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);  /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);/* client releases forks via the monitor */
   }
}
```

Figure 6.14 **A Solution to the Dining Philosophers Problem Using a Monitor**

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

Pipes, messages, and shared memory can be used to communicate data between processes, whereas semaphores and signals are used to trigger actions by other processes.

Pipes

One of the most significant contributions of UNIX to the development of operating systems is the pipe. Inspired by the concept of coroutines [RITC84], a pipe is a circular buffer allowing two processes to communicate on the producer-consumer model. Thus, it is a first-in-first-out queue, written by one process and read by another.

When a pipe is created, it is given a fixed size in bytes. When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked. Similarly, a reading process is blocked if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed. The OS enforces mutual exclusion: that is, only one process can access a pipe at a time.

There are two types of pipes: named and unnamed. Only related processes can share unnamed pipes, while either related or unrelated processes can share named pipes.

Messages

A message is a block of bytes with an accompanying type. UNIX provides `msgsnd` and `msgrcv` system calls for processes to engage in message passing. Associated with each process is a message queue, which functions like a mailbox.

The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver. The receiver can either retrieve messages in first-in-first-out order or by type. A process will block when trying to send a message to a full queue. A process will also block when trying to read

from an empty queue. If a process attempts to read a message of a certain type and fails because no message of that type is present, the process is not blocked.

Shared Memory

The fastest form of interprocess communication provided in UNIX is shared memory. This is a common block of virtual memory shared by multiple processes. Processes read and write shared memory using the same machine instructions they use to read and write other portions of their virtual memory space. Permission is read-only or read-write for a process, determined on a per-process basis. Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory.

Semaphores

The semaphore system calls in UNIX System V are a generalization of the `semWait` and `semSignal` primitives defined in Chapter 5; several operations can be performed simultaneously and the increment and decrement operations can be values greater than 1. The kernel does all of the requested operations atomically; no other process may access the semaphore until all operations have completed.

A semaphore consists of the following elements:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Associated with the semaphore are queues of processes blocked on that semaphore.

Semaphores are actually created in sets, with a semaphore set consisting of one or more semaphores. There is a `semctl` system call that allows all of the semaphore values in the set to be set at the same time. In addition, there is a `sem_op` system call that takes as an argument a list of semaphore operations, each defined on one of the semaphores in a set. When this call is made, the kernel performs the indicated operations one at a time. For each operation, the actual function is specified by the value `sem_op`. The following are the possibilities:

- If `sem_op` is positive, the kernel increments the value of the semaphore and awakens all processes waiting for the value of the semaphore to increase.
- If `sem_op` is 0, the kernel checks the semaphore value. If the semaphore value equals 0, the kernel continues with the other operations on the list. Otherwise, the kernel increments the number of processes waiting for this semaphore to be 0 and suspends the process to wait for the event that the value of the semaphore equals 0.
- If `sem_op` is negative and its absolute value is less than or equal to the semaphore value, the kernel adds `sem_op` (a negative number) to the semaphore value. If the result is 0, the kernel awakens all processes waiting for the value of the semaphore to equal 0.

- If `sem_op` is negative and its absolute value is greater than the semaphore value, the kernel suspends the process on the event that the value of the semaphore increases.

This generalization of the semaphore provides considerable flexibility in performing process synchronization and coordination.

Signals

A signal is a software mechanism that informs a process of the occurrence of asynchronous events. A signal is similar to a hardware interrupt but does not employ priorities. That is, all signals are treated equally; signals that occur at the same time are presented to a process one at a time, with no particular ordering.

Processes may send each other signals, or the kernel may send signals internally. A signal is delivered by updating a field in the process table for the process to which the signal is being sent. Because each signal is maintained as a single bit, signals of a given type cannot be queued. A signal is processed just after a process wakes up to run or whenever the process is preparing to return from a system call. A process may respond to a signal by performing some default action (e.g., termination), executing a signal handler function, or ignoring the signal.

Table 6.2 lists signals defined for UNIX SVR4.

Table 6.2 UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

6.8 LINUX KERNEL CONCURRENCY MECHANISMS

Linux includes all of the concurrency mechanisms found in other UNIX systems, such as SVR4, including pipes, messages, shared memory, and signals. In addition, Linux 2.6 includes a rich set of concurrency mechanisms specifically intended for use when a thread is executing in kernel mode. That is, these are mechanisms used within the kernel to provide concurrency in the execution of kernel code. This section examines the Linux kernel concurrency mechanisms.

Atomic Operations

Linux provides a set of operations that guarantee atomic operations on a variable. These operations can be used to avoid simple race conditions. An atomic operation executes without interruption and without interference. On a uniprocessor system, a thread performing an atomic operation cannot be interrupted once the operation has started until the operation is finished. In addition, on a multiprocessor system, the variable being operated on is locked from access by other threads until this operation is completed.

Two types of atomic operations are defined in Linux: integer operations, which operate on an integer variable, and bitmap operations, which operate on one bit in a bitmap (Table 6.3). These operations must be implemented on any architecture that implements Linux. For some architectures, there are corresponding assembly language instructions for the atomic operations. On other architectures, an operation that locks the memory bus is used to guarantee that the operation is atomic.

For **atomic integer operations**, a special data type is used, `atomic_t`. The atomic integer operations can be used only on this data type, and no other operations are allowed on this data type. [LOVE04] lists the following advantages for these restrictions:

1. The atomic operations are never used on variables that might in some circumstances be unprotected from race conditions.
2. Variables of this data type are protected from improper use by nonatomic operations.
3. The compiler cannot erroneously optimize access to the value (e.g., by using an alias rather than the correct memory address).
4. This data type serves to hide architecture-specific differences in its implementation.

A typical use of the atomic integer data type is to implement counters.

The **atomic bitmap operations** operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable. Thus, there is no equivalent to the `atomic_t` data type needed for atomic integer operations.

Atomic operations are the simplest of the approaches to kernel synchronization. More complex locking mechanisms can be built on top of them.

Spinlocks

The most common technique used for protecting a critical section in Linux is the spinlock. Only one thread at a time can acquire a spinlock. Any other thread attempting to

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t*v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i,atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

acquire the same lock will keep trying (spinning) until it can acquire the lock. In essence a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section. If the value is 0, the thread sets the value to 1 and enters its critical section. If the value is nonzero, the thread continually checks the value until it is zero. The spinlock is easy to implement but has the disadvantage that locked-out threads continue to execute in a busy-waiting mode. Thus spinlocks are most effective in situations where the wait time for acquiring a lock is expected to be very short, say on the order of less than two context changes.

The basic form of use of a spinlock is the following:

```
spin_lock(&lock)
/* critical section */
spin_unlock(&lock)
```

Table 6.4 Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore (spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Basic Spinlocks The basic spinlock (as opposed to the reader-writer spinlock explained subsequently) comes in four flavors (Table 6.4):

- **Plain:** If the critical section of code is not executed by interrupt handlers or if the interrupts are disabled during the execution of the critical section, then the plain spinlock can be used. It does not affect the interrupt state on the processor on which it is run.
- **_irq:** If interrupts are always enabled, then this spinlock should be used.
- **_irqsave:** If it is not known if interrupts will be enabled or disabled at the time of execution, then this version should be used. When a lock is acquired, the current state of interrupts on the local processor is saved, to be restored when the lock is released.
- **_bh:** When an interrupt occurs, the minimum amount of work necessary is performed by the corresponding interrupt handler. A piece of code, called the *bottom half*, performs the remainder of the interrupt-related work, allowing the current interrupt to be enabled as soon as possible. The `_bh` spinlock is used to disable and then enable bottom halves to avoid conflict with the protected critical section.

The plain spinlock is used if the programmer knows that the protected data is not accessed by an interrupt handler or bottom half. Otherwise, the appropriate nonplain spinlock is used.

Spinlocks are implemented differently on a uniprocessor system versus a multiprocessor system. For a uniprocessor system, the following considerations apply. If kernel preemption is turned off, so that a thread executing in kernel mode cannot be interrupted, then the locks are deleted at compile time; they are not needed. If kernel preemption is enabled, which does permit interrupts, then the spinlocks again compile away (that is, no test of a spinlock memory location occurs) but are simply implemented as code that enables/disables interrupts. On a multiple processor system, the spinlock is compiled into code that does in fact test the spinlock location. The use of the spinlock mechanism in a program allows it to be independent of whether it is executed on a uniprocessor or multiprocessor system.

Reader-Writer Spinlock The reader-writer spinlock is a mechanism that allows a greater degree of concurrency within the kernel than the basic spinlock. The reader-writer spinlock allows multiple threads to have simultaneous access to the same data structure for reading only but gives exclusive access to the spinlock for a thread that intends to update the data structure. Each reader-writer spinlock consists of a 24-bit reader counter and an unlock flag, with the following interpretation:

Counter	Flag	Interpretation
0	1	The spinlock is released and available for use
0	0	Spinlock has been acquired for writing by one thread
n ($n > 0$)	0	Spinlock has been acquired for reading by n threads
n ($n > 0$)	1	Not valid

As with the basic spinlock, there are plain, `_irq`, and `_irqsave` versions of the reader-writer spinlock.

Note that the reader-writer spinlock favors readers over writers. If the spinlock is held for readers, then so long as there is at least one reader, the spinlock cannot be preempted by a writer. Furthermore, new readers may be added to the spinlock even while a writer is waiting.

Semaphores

At the user level, Linux provides a semaphore interface corresponding to that in UNIX SVR4. Internally, Linux provides an implementation of semaphores for its own use. That is, code that is part of the kernel can invoke kernel semaphores. These kernel semaphores cannot be accessed directly by the user program via system calls. They are implemented as functions within the kernel and are thus more efficient than user-visible semaphores.

Linux provides three types of semaphore facilities in the kernel: binary semaphores, counting semaphores, and reader-writer semaphores.

Binary and Counting Semaphores The binary and counting semaphores defined in Linux 2.6 (Table 6.5) have the same functionality as described for such

Table 6.5 Linux Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers

semaphores in Chapter 5. The function names `down` and `up` are used for the functions referred to in Chapter 5 as `semWait` and `semSignal`, respectively.

A counting semaphore is initialized using the `sema_init` function, which gives the semaphore a name and assigns an initial value to the semaphore. Binary semaphores, called MUTEXes in Linux, are initialized using the `init_MUTEX` and `init_MUTEX_LOCKED` functions, which initialize the semaphore to 1 or 0, respectively.

Linux provides three versions of the `down` (`semWait`) operation.

1. The `down` function corresponds to the traditional `semWait` operation. That is, the thread tests the semaphore and blocks if the semaphore is not available. The thread will awaken when a corresponding up operation on this semaphore occurs. Note that this function name is used for an operation on either a counting semaphore or a binary semaphore.
2. The `down_interruptible` function allows the thread to receive and respond to a kernel signal while being blocked on the down operation. If the thread is woken up by a signal, the `down_interruptible` function increments the

count value of the semaphore and returns an error code known in Linux as -EINTR. This alerts the thread that the invoked semaphore function has aborted. In effect, the thread has been forced to "give up" the semaphore. This feature is useful for device drivers and other services in which it is convenient to override a semaphore operation.

3. The `down_trylock` function makes it possible to try to acquire a semaphore without being blocked. If the semaphore is available, it is acquired. Otherwise, this function returns a nonzero value without blocking the thread.

Reader-Writer Semaphores The reader-writer semaphore divides users into readers and writers; it allows multiple concurrent readers (with no writers) but only a single writer (with no concurrent readers). In effect, the semaphore functions as a counting semaphore for readers but a binary semaphore (MUTEX) for writers. Table 6.5 shows the basic reader-writer semaphore operations. The reader-writer semaphore uses uninterruptible sleep, so there is only one version of each of the `down` operations.

Barriers

In some architectures, compilers and/or the processor hardware may reorder memory accesses in source code to optimize performance. These reorderings are done to optimize the use of the instruction pipeline in the processor. The reordering algorithms contain checks to ensure that data dependencies are not violated. For example, the code:

```
a = 1;
b = 1;
```

may be reordered so that memory location `b` is updated before memory location `a` is updated. However, the code

```
a = 1;
b = a;
```

will not be reordered. Even so, there are occasions when it is important that reads or writes are executed in the order specified because of use of the information that is made by another thread or a hardware device.

To enforce the order in which instructions are executed, Linux provides the memory barrier facility. Table 6.6 lists the most important functions that are defined

Table 6.6 Linux Memory Barrier Operations

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a rmb( ) and on UP provides a barrier( )
`smp_wmb()`	On SMP, provides a wmb( ) and on UP provides a barrier( )
`smp_mb()`	On SMP, provides a mb( ) and on UP provides a barrier( )

SMP = symmetric multiprocessor
UP = uniprocessor

for this facility. The `rmb()` operation insures that no reads occur across the barrier defined by the place of the `rmb()` in the code. Similarly, the `wmb()` operation insures that no writes occur across the barrier defined by the place of the `wmb()` in the code. The `mb()` operation provides both a load and store barrier.

Two important points to note about the barrier operations:

1. The barriers relate to machine instructions, namely loads and stores. Thus the higher-level language instruction `a = b` involves both a load (read) from location `b` and a store (write) to location `a`.
2. The `rmb`, `wmb`, and `mb` operations dictate the behavior of both the compiler and the processor. In the case of the compiler, the barrier operation dictates that the compiler not reorder instructions during the compile process. In the case of the processor, the barrier operation dictates that any instructions pending in the pipeline before the barrier must be committed for execution before any instructions encountered after the barrier.

The `barrier()` operation is a lighter-weight version of the `mb()` operation, in that it only controls the compiler's behavior. This would be useful if it is known that the processor will not perform undesirable reorderings. For example, the Intel x86 processors do not reorder writes.

The `smp_rmb`, `smp_wmb`, and `smp_mb` operations provide an optimization for code that may be compiled on either a uniprocessor (UP) or a symmetric multiprocessor (SMP). These instructions are defined as the usual memory barriers for an SMP, but for a UP, they are all treated only as compiler barriers. The `smp_` operations are useful in situations in which the data dependencies of concern will only arise in an SMP context.

6.9 SOLARIS THREAD SYNCHRONIZATION PRIMITIVES

In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

Solaris implements these primitives within the kernel for kernel threads; they are also provided in the threads library for user-level threads. Figure 6.15 shows the data structures for these primitives. The initialization functions for the primitives fill in some of the data members. Once a synchronization object is created, there are essentially only two operations that can be performed: enter (acquire lock) and release (unlock). There are no mechanisms in the kernel or the threads library to enforce mutual exclusion or to prevent deadlock. If a thread attempts to access a piece of data or code that is supposed to be protected but does not use the appropriate synchronization primitive, then such access occurs. If a thread locks an object and then fails to unlock it, no kernel action is taken.

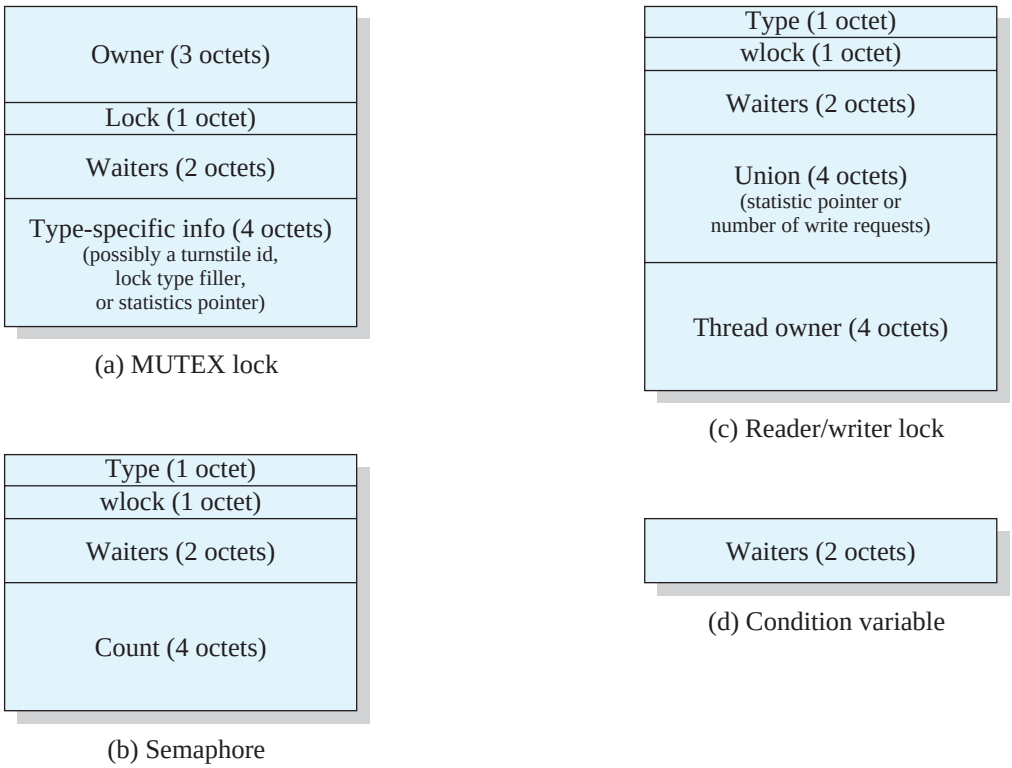


Figure 6.15 **Solaris Synchronization Data Structures**

All of the synchronization primitives require the existence of a hardware instruction that allows an object to be tested and set in one atomic operation.

Mutual Exclusion Lock

A mutex is used to ensure only one thread at a time can access the resource protected by the mutex. The thread that locks the mutex must be the one that unlocks it. A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive. If `mutex_enter` cannot set the lock (because it is already set by another thread), the blocking action depends on type-specific information stored in the mutex object. The default blocking policy is a spin lock: a blocked thread polls the status of the lock while executing in a busy waiting loop. An interrupt-based blocking mechanism is optional. In this latter case, the mutex includes a `turnstile id` that identifies a queue of threads sleeping on this lock.

The operations on a mutex lock are as follows:

`mutex_enter()`	Acquires the lock, potentially blocking if it is already held
`mutex_exit()`	Releases the lock, potentially unblocking a waiter
`mutex_tryenter()`	Acquires the lock if it is not already held

The `mutex_tryenter()` primitive provides a nonblocking way of performing the mutual exclusion function. This enables the programmer to use a busy-wait approach for user-level threads, which avoids blocking the entire process because one thread is blocked.

Semaphores

Solaris provides classic counting semaphores, with the following primitives:

`sema_p()`	Decrements the semaphore, potentially blocking the thread
`sema_v( )`	Increments the semaphore, potentially unblocking a waiting thread
`sema_tryp( )`	Decrements the semaphore if blocking is not required

Again, the `sema_tryp()` primitive permits busy waiting.

Animation: Solaris RW Lock

Readers/Writer Lock

The readers/writer lock allows multiple threads to have simultaneous read-only access to an object protected by the lock. It also allows a single thread to access the object for writing at one time, while excluding all readers. When the lock is acquired for writing it takes on the status of `write lock`: All threads attempting access for reading or writing must wait. If one or more readers have acquired the lock, its status is `read lock`. The primitives are as follows:

`rw_enter()`	Attempts to acquire a lock as reader or writer.
`rw_exit()`	Releases a lock as reader or writer.
`rw_tryenter()`	Acquires the lock if blocking is not required.
`rw_downgrade()`	A thread that has acquired a write lock converts it to a read lock. Any waiting writer remains waiting until this thread releases the lock. If there are no waiting writers, the primitive wakes up any pending readers.
`rw_tryupgrade()`	Attempts to convert a reader lock into a writer lock.

Condition Variables

A condition variable is used to wait until a particular condition is true. Condition variables must be used in conjunction with a mutex lock. This implements a monitor of the type illustrated in Figure 6.14. The primitives are as follows:

`cv_wait()`	Blocks until the condition is signaled
`cv_signal()`	Wakes up one of the threads blocked in `cv_wait()`
`cv_broadcast()`	Wakes up all of the threads blocked in `cv_wait()`

`cv_wait()` releases the associated mutex before blocking and reacquires it before returning. Because reacquisition of the mutex may be blocked by other

threads waiting for the mutex, the condition that caused the wait must be retested. Thus, typical usage is as follows:

```
mutex_enter(&m)
* *
while (some_condition) {
  cv_wait(&cv, &m);
}
* *
mutex_exit(&m);
```

This allows the condition to be a complex expression, because it is protected by the mutex.

6.10 WINDOWS CONCURRENCY MECHANISMS

Windows provides synchronization among threads as part of the object architecture. The most important methods of synchronization are Executive dispatcher objects, user mode critical sections, slim reader-writer locks, and condition variables. Dispatcher objects make use of wait functions. We first describe wait functions and then look at the synchronization methods.

Wait Functions

The wait functions allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

The most straightforward type of wait function is one that waits on a single object. The `WaitForSingleObject` function requires a handle to one synchronization object. The function returns when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to `INFINITE` to specify that the wait will not time out.

Dispatcher Objects

The mechanism used by the Windows Executive to implement synchronization facilities is the family of dispatcher objects, which are listed with brief descriptions in Table 6.7.

The first five object types in the table are specifically designed to support synchronization. The remaining object types have other uses but also may be used for synchronization.

Each dispatcher object instance can be in either a signaled or unsignaled state. A thread can be blocked on an object in an unsignaled state; the thread is released when the object enters the signaled state. The mechanism is straightforward: A thread issues

WINDOWS/LINUX COMPARISON	
Windows	**Linux**
Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying wait/signal mechanism	Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying sleep/wakeup mechanism
Many kernel objects are also *dispatcher objects*, meaning that threads can synchronize with them using a common event mechanism, available at user-mode. Process and thread termination are events, I/O completion is an event	
Threads can wait on multiple dispatcher objects at the same time	Processes can use the select() system call to wait on I/O from up to 64 file descriptors
User-mode reader/writer locks and condition variables are supported	User-mode reader/writer locks and condition variables are supported
Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported	Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported
A non-locking atomic LIFO queue, called an SLIST, is supported using compare-and-swap; widely used in the OS and also available to user programs	
A large variety of synchronization mechanisms exist within the kernel to improve scalability. Many are based on simple compare-and-swap mechanisms, such as push-locks and fast references of objects	
Named pipes, and sockets support remote procedure calls (RPCs), as does an efficient Local Procedure Call mechanism (ALPC), used within a local system. ALPC is used heavily for communicating between clients and local services	Named pipes, and sockets support remote procedure calls (RPCs)
Asynchronous Procedure Calls (APCs) are used heavily within the kernel to get threads to act upon themselves (e.g. termination and I/O completion use APCs since these operations are easier to implement in the context of a thread rather than cross-thread). APCs are also available for user-mode, but user-mode APCs are only delivered when a user-mode thread blocks in the kernel	Unix supports a general *signal* mechanism for communication between processes. Signals are modeled on hardware interrupts and can be delivered at any time that they are not blocked by the receiving process; like with hardware interrupts, signal semantics are complicated by multi-threading
Hardware support for deferring interrupt processing until the interrupt level has dropped is provided by the Deferred Procedure Call (DPC) control object	Uses *tasklets* to defer interrupt processing until the interrupt level has dropped

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

a wait request to the Windows Executive, using the handle of the synchronization object. When an object enters the signaled state, the Windows Executive releases one or all of the thread objects that are waiting on that dispatcher object.

The **event object** is useful in sending a signal to a thread indicating that a particular event has occurred. For example, in overlapped input and output, the system sets a specified event object to the signaled state when the overlapped operation has been completed. The **mutex object** is used to enforce mutually exclusive access to a resource, allowing only one thread object at a time to gain access. It therefore functions as a binary semaphore. When the mutex object enters the signaled state, only one of the threads waiting on the mutex is released. Mutexes can be used to synchronize threads running in different processes. Like mutexes, **semaphore objects** may be shared by threads in multiple processes. The Windows semaphore is a counting semaphore. In essence, the **waitable timer object** signals at a certain time and/or at regular intervals.

Critical Sections

Critical sections provide a synchronization mechanism similar to that provided by mutex objects, except that critical sections can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical sections provide a much faster, more efficient mechanism for mutual-exclusion synchronization.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type `CRITICAL_SECTION`. Before the threads of the process can use it, initialize the critical section by using the `InitializeCriticalSection or InitializeCriticalSectionAndSpinCount` function.

A thread uses the `EnterCriticalSection` or `TryEnterCriticalSection` function to request ownership of a critical section. It uses the `LeaveCriticalSection` function to release ownership of a critical section. If the critical section is currently owned by another thread, `EnterCriticalSection` waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functions accept a specified time-out interval. The `TryEnterCriticalSection` function attempts to enter a critical section without blocking the calling thread.

Critical sections use a sophisticated algorithm when trying to acquire the mutex. If the system is a multiprocessor, the code will attempt to acquire a spin-lock. This works well in situations where the critical section is acquired for only a short time. Effectively the spinlock optimizes for the case where the thread that currently owns the critical section is executing on another processor. If the spinlock cannot be acquired within a reasonable number of iterations, a dispatcher object is used to block the thread so that the Kernel can dispatch another thread onto the processor. The dispatcher object is only allocated as a last resort. Most critical sections are needed for correctness, but in practice are rarely contended. By lazily allocating the dispatcher object the system saves significant amounts of kernel virtual memory.

Slim Read-Writer Locks and Condition Variables

Windows Vista added a user mode reader-writer. Like critical sections, the reader-writer lock enters the kernel to block only after attempting to use a spin-lock. It is *slim* in the sense that it normally only requires allocation of a single pointer-sized piece of memory.

To use an SRW a process declares a variable of type SRWLOCK and a calls `InitializeSRWLock` to initialize it. Threads call `AcquireSRWLockExclusive` or `AcquireSRWLockShared` to acquire the lock and `ReleaseSRWLockExclusive` or `ReleaseSRWLockShared` to release it.

Windows Vista also added condition variables. The process must declare a CONDITION_VARIABLE and initialize it in some thread by calling `InitializeConditionVariable`. Condition variables can be used with either critical sections or SRW locks, so there are two methods, `SleepConditionVariableCS` and `SleepConditionVariableSRW`, which sleep on the specified condition and releases the specified lock as an atomic operation.

There are two wake methods, `WakeConditionVariable` and `WakeAllConditionVariable`, which wake one or all of the sleeping threads, respectively. Condition variables are used as follows:

1. Acquire exclusive lock
2. While (predicate() == FALSE) SleepConditionVariable()
3. Perform the protected operation
4. Release the lock

6.11 SUMMARY

Deadlock is the blocking of a set of processes that either compete for system resources or communicate with each other. The blockage is permanent unless the OS takes some extraordinary action, such as killing one or more processes or forcing one or more processes to backtrack. Deadlock may involve reusable resources or consumable resources. A reusable resource is one that is not depleted or destroyed by use, such as an I/O channel or a region of memory. A consumable resource is one that is destroyed when it is acquired by a process; examples include messages and information in I/O buffers.

There are three general approaches to dealing with deadlock: prevention, detection, and avoidance. Deadlock prevention guarantees that deadlock will not occur, by assuring that one of the necessary conditions for deadlock is not met. Deadlock detection is needed if the OS is always willing to grant resource requests; periodically, the OS must check for deadlock and take action to break the deadlock. Deadlock avoidance involves the analysis of each new resource request to determine if it could lead to deadlock, and granting it only if deadlock is not possible.

6.12 RECOMMENDED READING

The classic paper on deadlocks, [HOLT72], is still well worth a read, as is [COFF71]. Another good survey is [ISLO80]. [CORB96] is a thorough treatment of deadlock detection. [DIMI98] is a nice overview of deadlocks. Two recent papers by Levine [LEVI03a, LEVI03b] clarify some of the concepts used in discussions of deadlock. [SHUB03] is a useful overview of deadlock. [ABRA06] describes a deadlock detection package.

The concurrency mechanisms in UNIX SVR4, Linux, and Solaris 2 are well covered in [GRAY97], [LOVE05], and [MCDO07], respectively.

ABRA06 Abramson, T. "Detecting Potential Deadlocks." *Dr. Dobb's Journal*, January 2006.

COFF71 Coffman, E.; Elphick, M.; and Shoshani, A. "System Deadlocks." *Computing Surveys,* June 1971.

CORB96 Corbett, J. "Evaluating Deadlock Detection Methods for Concurrent Software." *IEEE Transactions on Software Engineering*, March 1996.

DIMI98 Dimitoglou, G. "Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems." *Operating Systems Review*, July 1998.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies.* Upper Saddle River, NJ: Prentice Hall, 1997.

HOLT72 Holt, R. "Some Deadlock Properties of Computer Systems." *Computing Surveys,* September 1972.

ISLO80 Isloor, S., and Marsland, T. "The Deadlock Problem: An Overview." *Computer*, September 1980.

LEVI03a Levine, G. "Defining Deadlock." *Operating Systems Review*, January 2003.

LEVI03b Levine, G. "Defining Deadlock with Fungible Resources." *Operating Systems Review*, July 2003.

LOVE05 Love, R. *Linux Kernel Development.* Indianapolis, IN: Novell Press, 2005.

MCDO07 McDougall, R., and Mauro, J. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture.* Palo Alto, CA: Sun Microsystems Press, 2007.

SHUB03 Shub, C. "A Unified Treatment of Deadlock." *Journal of Computing in Small Colleges*, October 2003. Available through the ACM digital library.

6.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

banker's algorithm	deadlock prevention	pipe
circular wait	hold and wait	preemption
consumable resource	joint progress diagram	resource allocation graph
deadlock	memory barrier	reusable resource
deadlock avoidance	message	spinlock
deadlock detection	mutual exclusion	starvation

Review Questions

6.1 Give examples of reusable and consumable resources.

6.2 What are the three conditions that must be present for deadlock to be possible?

6.3 What are the four conditions that create deadlock?

6.4 How can the hold-and-wait condition be prevented?

6.5 List two ways in which the no-preemption condition can be prevented.

6.6 How can the circular wait condition be prevented?

6.7 What is the difference among deadlock avoidance, detection, and prevention?

Problems

6.1 Show that the four conditions of deadlock apply to Figure 6.1a.

6.2 Show how each of the techniques of prevention, avoidance, and detection can be applied to Figure 6.1.

6.3 For Figure 6.3, provide a narrative description of each of the six depicted paths, similar to the description of the paths of Figure 6.2 provided in Section 6.1.

6.4 It was stated that deadlock cannot occur for the situation reflected in Figure 6.3. Justify that statement.

6.5 Consider the following snapshot of a system. There are no outstanding unsatisfied requests for resources.

available			
r1	r2	r3	r4
2	1	0	0

	current allocation				maximum demand				still needs			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2				
p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

a. Compute what each process still might request and display in the columns labeled "still needs."
b. Is this system currently in a safe or unsafe state? Why?
c. Is this system currently deadlocked? Why or why not?
d. Which processes, if any, are or may become deadlocked?
e. If a request from p3 arrives for (0, 1, 0, 0), can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting that whole request leave the system? Which processes, if any, are or may become deadlocked if this whole request is granted immediately?

6.6 Apply the deadlock detection algorithm of Section 6.4 to the following data and show the results.

$$\text{Available} = (2 \quad 1 \quad 0 \quad 0)$$

$$\text{Request} = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \quad \text{Allocation} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

6.7 A spooling system (Figure 6.16) consists of an input process I, a user process P, and an output process O connected by two buffers. The processes exchange data in blocks of equal size. These blocks are buffered on a disk using a floating boundary between the input and the output buffers, depending on the speed of the processes. The communication primitives used ensure that the following resource constraint is satisfied:

$$i + o \leq max$$

where

max = maximum number of blocks on disk
i = number of input blocks on disk
o = number of output blocks on disk

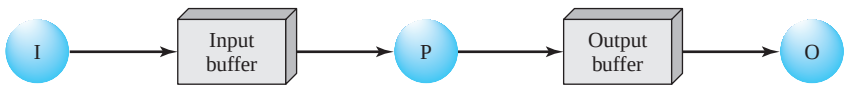


Figure 6.16 **A Spooling System**

The following is known about the processes:

1. As long as the environment supplies data, process I will eventually input it to the disk (provided disk space becomes available).
2. As long as input is available on the disk, process P will eventually consume it and output a finite amount of data on the disk for each block input (provided disk space becomes available).
3. As long as output is available on the disk, process O will eventually consume it.

Show that this system can become deadlocked.

6.8 Suggest an additional resource constraint that will prevent the deadlock in Problem 6.6 but still permit the boundary between input and output buffers to vary in accordance with the present needs of the processes.

6.9 In THE multiprogramming system [DIJK68], a drum (precursor to the disk for secondary storage) is divided into input buffers, processing areas, and output buffers, with floating boundaries, depending on the speed of the processes involved. The current state of the drum can be characterized by the following parameters:

max = maximum number of pages on drum
i = number of input pages on drum
p = number of processing pages on drum
o = number of output pages on drum
$reso$ = minimum number of pages reserved for output
$resp$ = minimum number of pages reserved for processing

Formulate the necessary resource constraints that guarantee that the drum capacity is not exceeded and that a minimum number of pages is reserved permanently for output and processing.

6.10 In THE multiprogramming system, a page can make the following state transitions:

1. empty → input buffer (input production)
2. input buffer → processing area (input consumption)
3. processing area → output buffer (output production)
4. output buffer → empty (output consumption)
5. empty → processing area (procedure call)
6. processing area → empty (procedure return)

a. Define the effect of these transitions in terms of the quantities i, o, and p.
b. Can any of them lead to a deadlock if the assumptions made in Problem 6.6 about input processes, user processes, and output processes hold?

6.11 Consider a system with a total of 150 units of memory, allocated to three processes as shown:

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Apply the banker's algorithm to determine whether it would be safe to grant each of the following requests. If yes, indicate a sequence of terminations that could be guaranteed possible. If no, show the reduction of the resulting allocation table.

a. A fourth process arrives, with a maximum memory need of 60 and an initial need of 25 units.
b. A fourth process arrives, with a maximum memory need of 60 and an initial need of 35 units.

6.12 Evaluate the banker's algorithm for its usefulness in an OS.

6.13 A pipeline algorithm is implemented so that a stream of data elements of type T produced by a process P_0 passes through a sequence of processes $P_1, P_2, \ldots, P_{n-1}$, which operates on the elements in that order.

a. Define a generalized message buffer that contains all the partially consumed data elements and write an algorithm for process Pi ($0 \leq i \leq n - 1$), of the form

```
repeat
   receive from predecessor;
   consume element;
   send to successor:
forever
```

Assume P_0 receives input elements sent by P_{n-1}. The algorithm should enable the processes to operate directly on messages stored in the buffer so that copying is unnecessary.

b. Show that the processes cannot be deadlocked with respect to the common buffer.

6.14 **a.** Three processes share four resource units that can be reserved and released only one at a time. Each process needs a maximum of two units. Show that a deadlock cannot occur.

b. N processes share M resource units that can be reserved and released only one at a time. The maximum need of each process does not exceed M, and the sum of all maximum needs is less than $M + N$. Show that a deadlock cannot occur.

6.15 Consider a system consisting of four processes and a single resource. The current state of the claim and allocation matrices are

$$\mathbf{C} = \begin{pmatrix} 3 \\ 2 \\ 9 \\ 7 \end{pmatrix} \qquad \mathbf{A} = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$

What is the minimum number of units of the resource needed to be available for this state to be safe?

6.16 Consider the following ways of handling deadlock: (1) banker's algorithm, (2) detect deadlock and kill thread, releasing all resources, (3) reserve all resources in advance, (4) restart thread and release all resources if thread needs to wait, (5) resource ordering, and (6) detect deadlock and roll back thread's actions.

a. One criterion to use in evaluating different approaches to deadlock is which approach permits the greatest concurrency. In other words, which approach allows the most threads to make progress without waiting when there is no deadlock? Give a rank order from 1 to 6 for each of the ways of handling deadlock just listed, where 1 allows the greatest degree of concurrency. Comment on your ordering.

b. Another criterion is efficiency; in other words, which requires the least processor overhead. Rank order the approaches from 1 to 6, with 1 being the most efficient, assuming that deadlock is a very rare event. Comment on your ordering. Does your ordering change if deadlocks occur frequently?

6.17 Comment on the following solution to the dining philosophers problem. A hungry philosopher first picks up his left fork; if his right fork is also available, he picks up his right fork and starts eating; otherwise he puts down his left fork again and repeats the cycle.

6.18 Suppose that there are two types of philosophers. One type always picks up his left fork first (a "lefty"), and the other type always picks up his right fork first (a

"righty"). The behavior of a lefty is defined in Figure 6.12. The behavior of a righty is as follows:

```
begin
  repeat
        think;
        wait ( fork[ (i+1) mod 5] );
        wait ( fork[i] );
        eat;
        signal ( fork[i] );
        signal ( fork[ (i+1) mod 5] );
  forever
end;
```

Prove the following:

a. Any seating arrangement of lefties and righties with at least one of each avoids deadlock.

b. Any seating arrangement of lefties and righties with at least one of each prevents starvation.

6.19 Figure 6.17 shows another solution to the dining philosophers problem using monitors. Compare to Figure 6.14 and report your conclusions.

```
monitor dining_controller;
enum states {thinking, hungry, eating} state[5];
cond needFork[5]                          /* condition variable */

void get_forks(int pid)   /* pid is the philosopher id number */
{
   state[pid] = hungry;   /* announce that I'm hungry */
   if (state[(pid+1) % 5] == eating || (state[(pid-1) % 5] == eating)
   cwait(needFork[pid]);  /* wait if either neighbor is eating */
   state[pid] = eating;  /* proceed if neither neighbor is eating */
}

void release_forks(int pid)
{
   state[pid] = thinking;
   /* give right (higher) neighbor a chance to eat */
   if (state[(pid+1) % 5] == hungry) && (state[(pid+2)
   % 5]) != eating)
   csignal(needFork[pid+1]);
   /* give left (lower) neighbor a chance to eat */
   else if (state[(pid-1) % 5] == hungry) && (state[(pid-2)
   % 5]) != eating)
   csignal(needFork[pid-1]);
}
```

Figure 6.17 Another Solution to the Dining Philosophers Problem Using a Monitor

```
void philosopher[k=0 to 4]   /* the five philosopher clients */
{
   while (true) {
     <think>;
     get_forks(k);   /* client requests two forks via monitor */
     <eat spaghetti>;
     release_forks(k);  /* client releases forks via the monitor */
   }
}
```

Figure 6.17 (*continued*)

6.20 In Table 6.3, some of the Linux atomic operations do not involve two accesses to a variable, such as `atomic_read(atomic_t *v)`. A simple read operation is obviously atomic in any architecture. Therefore, why is this operation added to the repertoire of atomic operations?

6.21 Consider the following fragment of code on a Linux system.

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

where `mr_rwlock` is a reader-writer lock. What is the effect of this code?

6.22 The two variables `a` and `b` have initial values of 1 and 2, respectively. The following code is for a Linux system:

Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`b=4;`	`c = b;`
—	`rmb();`
—	`d = a;`

What possible errors are avoided by the use of the memory barriers?