5.8 References

1. Andrews, G. (1991). *Concurrent Programming: Principles and Practice*. The Benjamin-Cimmungs Publishing Company, Inc., Redwood City, CA.
2. Ben-Ari (1990). *Principles of Concurrent and Distributed Programming*. Prentice Hall, New York, NY.
3. Gehani, N. H. and Roome, W. D. (1989). *Concurrent C*. Silicon Press.
4. Lampson, B. W. and Redell, D. D. (1980). Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117.
5. Wirth, N. (1985). *Programming in MODULA-2*. Springer-Verlag, New York, NY.

CHAPTER 6

Deadlock Analysis

Deadlock among a group of processes or threads is a situation in which each of them waits for a condition that can only be satisfied by the others. This situation usually arises when threads exhaust a finite pool of shared system resources, and the unmet condition is the release of some resources back into the pool. Since the deadlocked threads all wait for the others to release the needed resources, none of them makes any forward progress. In this chapter we theoretically examine resource requirements and conditions that create deadlock. Using Dijkstra's dining philosophers example, we discuss the strategies and techniques commonly used to detect, prevent, and avoid system deadlocks.

6.1 Examples of Deadlock

Deadlock comes in many forms, but it can always be described in terms of a resource *requirement, acquisition,* and *release.* All of us have encountered and possibly been frustrated by deadlocks in one form or another. Let us examine a

few:

- The working of the U.S. Congress. We can view each political party as a subsystem that requires the resources held by the other subsystems: power, concessions, or compromises. With every subsystem tightly holding onto its own while resources waiting for the others to give up theirs, it is clear why there is a perpetual deadlock on Capitol Hill.
- Another example is an automobile traffic deadlock (also known as gridlock). Figure 6–1 illustrates such a situation where drivers enter an already congested intersection, blocking the flow of traffic around a block. In this example, the individual automobiles are the consuming processes and the resources are the spaces they occupy.

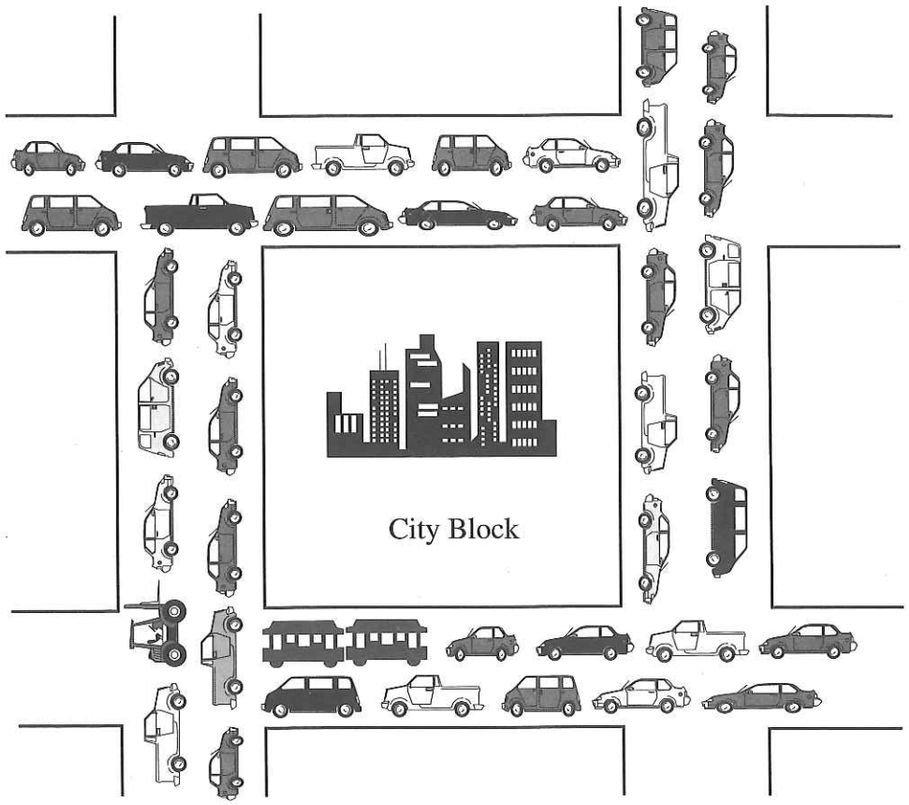

Figure 6–1. Traffic Gridlock.

- In a multithreaded program, a deadlock could result from two threads requesting file descriptors, usually a limited resource. Suppose a system has ten file descriptors, where thread T_1 was granted five and thread T_2 was granted four. If each thread makes an additional request for two file descriptors in order to finish their respective tasks, the system cannot satisfy either request with the one remaining file descriptor. We thus have a deadlock, even though each thread individually made a reasonable request and did not require more file descriptors than the system's total resources.

In the following sections, we will explore the requirements and conditions for deadlock, and what we can do to detect, prevent, or avoid it.

6.2 System Resources

Our definition of "resources" is not limited to hardware devices (printers, terminals, etc.), processors (CPU, FPU), or storage media (memory, disks, tapes). It also includes stored information such as programs, subroutines, files, and data. A resource can exist alone or as a collection of several similar instances. For example, in a single-processor machine, the CPU is the only resource of its type; but in a multiprocessor computer, a CPU is one resource unit among several of the same kind.

Computer programs need these resources to complete their tasks. When a resource is needed, a thread asks for it, uses it, and returns it back to the system when done. When a resource request is issued, the system may grant the resource if available, or deny the request when it cannot be satisfied.

Since each resource unit can only be used by one thread at a time, and threads may execute in parallel, access to resources must be carefully managed, and the potential for deadlock must be considered when determining the sequence in which requests are granted.

6.3 Conditions for Deadlock

The early work of Coffman et al. [1971] showed that in order for deadlock to occur, four conditions must exist:

1. **Mutual Exclusion**: A thread can seize exclusive control of an object, and no other thread can have access to it.
2. **Hold and Wait**: A thread can hold locked resources while waiting to acquire more.
3. **No Preemption**: Once a certain resource is held by a thread, it cannot be involuntarily reassigned to another thread.
4. **Circular Wait**: Two or more threads can hold some resources and await some held by other threads. If this dependency is circular, the threads will all be waiting for others to release needed resources, and none will make progress.

The following example shows three threads in a potential deadlock:

- Thread T_1 wants resources R_1, R_2
- Thread T_2 wants resources R_2, R_3
- Thread T_3 wants resources R_3, R_1

Each thread wants to acquire the resources it needs, perform some computation, and then return the resources to the system when finished (Figure 6–2).

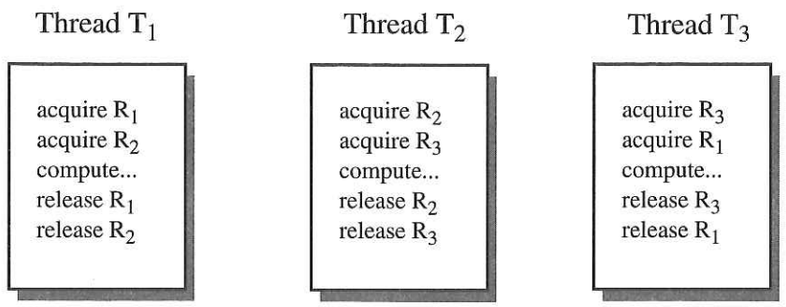

Figure 6–2. Individual Threads and their Resource Requirements.

Most of the time, threads T_1, T_2, and T_3 will each be able to seize all of their required resources in some order and proceed with their computations. Then there is no deadlock. For example:

T_1 gets R_1 & R_2, starts to compute
T_2 wants R2 $\Rightarrow$ blocks on R_2
T_3 wants R_3 & R_1 $\Rightarrow$ gets R_3 $\Rightarrow$ blocks on R_1
T_1 finishes and releases R_1 & R_2
T_2 gets R_2, wants R_3 $\Rightarrow$ blocks on R_3
T_3 gets R_1 and starts to compute
T_3 finishes and releases R_3, R_1
T_2 gets R_3 and starts to compute
T_2 finishes and releases R_2, R_3

This order of execution is just one possible permutation of the scheduling order, but as long as one thread is able to obtain its required resources there is no deadlock. Other threads might have to wait, but eventually they will finish.

However, deadlock occurs if the scheduler happens to interleave them like this:

T_1 gets R_1
T_2 gets R_2
T_3 gets R_3
T_1 wants R_2 $\Rightarrow$ block (T_2 has it)

T_2 wants R_3 $\Rightarrow$ block (T_3 has it)
T_3 wants R_1 $\Rightarrow$ block (T_1 has it)
Deadlock!

Since we usually cannot control the order of thread execution, we must be prepared for the worst and handle this case if we want to avoid deadlock unconditionally. This deadlock example satisfies all four deadlock requirements discussed earlier:

Mutual exclusion: Threads T_1, T_2, and T_3 lock and hold exclusive access to resources R_1, R_2, and R_3, respectively.

Hold and wait: T_1, T_2, and T_3 individually hold R_1, R_2, and R_3, while waiting for R_2, R_3, and R_1, respectively.

No preemption: Neither the system nor the thread themselves voluntarily reassign resources held by the other threads. Once each thread waits for the others in a deadlock, there is no mechanism for the system to automatically seize a resource from one thread and give it to another thread in order to break the deadlock.

Circular wait: Each thread depends on resources held by other threads to make progress. In this case, the dependency is circular. Since T_1 depends on T_2 to release R_2, T_2 depends on T_3 to release R_3, and T_3 depends on T_1 to release R_1, all threads wait and none makes progress.

6.4 The Dining Philosophers Problem

A classic example of deadlock was first described by E. Dijkstra [1968]. In this problem, there are five philosophers sitting around a round table eating a meal of spaghetti. The table has been arranged such that each philosopher has one plate and between each pair of plates is a fork to be used for eating spaghetti. Each philosopher alternately eats spaghetti and thinks. The constraint is that each philosopher needs both a left and a right fork to eat the spaghetti. As each philosopher picks up the forks, eats, then sets them down to think, there is a possibility that none of them can eat again because of deadlock. This situation occurs when all five philosophers simultaneously pick up their left forks, and forever wait for the others to release another fork needed to begin eating. The same situation will also happen when all five philosophers simultaneously pick up their right forks. Figure 6–3 shows the table arrangement for the five philosophers. The philosophers are represented by P_0 to P_4, and the forks are represented by F_0 to F_4.

The following naive implementation of this problem demonstrates the occurrence of deadlock:

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4   #define PCOUNT 5
5
6   HANDLE hForkMutex[PCOUNT]; // array of mutex representing forks
```

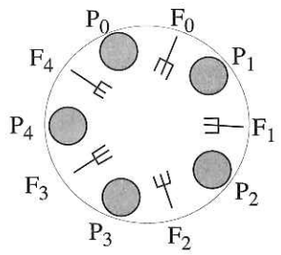

Figure 6–3. The Dining Philosophers' Table.

```
7
8   VOID Philosopher(LPVOID id)
9   {
10     int left = (int) id;
11     int right = (left + 1) % PCOUNT;
12
13     while(1){
14        // pick up left fork
15        WaitForSingleObject(hForkMutex[left],INFINITE);
16        printf("Philosopher #%d: picked up left fork\n",(int) id);
17        // pick up right fork
18        WaitForSingleObject(hForkMutex[right],INFINITE);
19        printf("Philosopher #%d: picked right fork & started eating\n",
20                    (int) id);
21        ReleaseMutex(hForkMutex[left]);
22        ReleaseMutex(hForkMutex[right]);
23        printf("Philosopher #%d: released forks and now thinking\n",
24                    (int) id);
25     }
26  }
27
28  void main()
29  {
30     HANDLE hThreadVector[PCOUNT];
31     DWORD ThreadID;
32     int i;
33
34     for (i=0; i<PCOUNT; i++)
35        hForkMutex[i] = CreateMutex(NULL, FALSE, NULL);
36
37     for (i=0; i<PCOUNT; i++)
38        hThreadVector[i] = CreateThread (NULL, 0,
39                (LPTHREAD_START_ROUTINE)Philosopher,
40                (LPVOID) i, 0, (LPDWORD)&ThreadID);
41
42     WaitForMultipleObjects(PCOUNT,hThreadVector,TRUE,INFINITE);
43  }
```

We will revise this example in the next few sections to demonstrate several deadlock prevention strategies.

6.5 Handling Deadlocks

In general, there are several ways to deal with deadlocks:

- **Ignoration**: Handling deadlock is costly, and sometimes it is not practical to spend much computing resource to handle it, particularly in systems and situations where deadlock possibility is small. Since deadlock is a benign and easily detectable problem, in some cases one may choose not to handle deadlock in a program, and then manually break the cycle (usually by terminating one or several deadlocked threads or processes) when a deadlock is detected.
- **Detection**: There are algorithms to search for cycles of deadlocked threads and processes. When such cycles are detected, we can sometimes preempt some threads to release their resources, roll back their execution state, or simply terminate them.
- **Prevention**: Through a set of rigid rules and restrictive resource allocation requirements, we can prevent deadlock from occurring by precluding any one of the four conditions for deadlock at every point in a program.
- **Avoidance**: We can implement algorithms and services to carefully manage resource allocation in order to avoid getting into a potential deadlock situation. For efficiency, this solution does not rule out all possibility of deadlock like the *deadlock prevention* strategy; it simply monitors the state of the system's resource allocation and circumvents deadlock when imminent by being very conservative about resource allocation. Djikstra's *banker's algorithm* is a classic example of this approach.

Although these strategies provide the desired result, they have associated costs. If we decide to ignore the deadlock problem, we may eventually have to face a deadlock situation, and the cost will be the loss of work due to terminating some of the deadlocked threads. Using the deadlock detection, prevention, and avoidance strategies, the cost involves lengthy algorithms to detect deadlock, inefficient resource usage, or an elaborate resource allocation scheme, respectively. Furthermore, each strategy has other limitations. In all cases, implementing these strategies will no doubt result in programs that are more complex and slower. We now examine each strategy and its weaknesses in more details.

6.6 Deadlock Prevention

Prevention is the most conservative and costly strategy for eliminating deadlock. Of the four conditions that must be present for deadlock to occur, we try to preclude at least one at each point in a program.

Removing the first condition, mutual exclusion, is not desirable or feasible since we need it in order to run threads without corrupting shared resources. That leaves three remaining conditions. Havender [1968] proposed the following

strategies, each aimed at precluding one of them:

- **Prevent hold-and-wait**: Each thread must request all resources at once, and it cannot proceed until all resources are granted.
- **Allow preemption**: If a thread holding a resource is denied further resource requests, it must release all resources that it holds and request them later with the additional resource.
- **Prevent circular wait**: Impose a linear ordering on all resources R_j such that the order of resource request and allocation follows this ordering. For example, if a task has been allocated resources of type R_j, it can subsequently request only resource R_{j+1} or, later in the ordering, R_1 to R_n.

Although it is theoretically possible to eliminate deadlock with these strategies, implementing any of these in practice can involve cost and inefficiency. In the following sections, we examine these strategies in more detail.

6.6.1 Preventing the Hold-and-Wait Condition

The first strategy proposed by Havender, aimed at denying the hold-and-wait condition, requires that a process requests all resources at once, and that the system grants either all resources or none. Using this strategy, threads no longer hold resources while trying to acquire other resources, and the possibility of deadlock is eliminated.

In the following example, we modify the program of Section 6.4 to prevent a hold-and-wait condition. Note that in lines 12–14, we try to get both forks at once. Since Windows NT does not provide any function that can guarantee an all-or-nothing lock on multiple resources, we simulate it with our GetBothForks function, which either successfully locks both resources or indicates its failure to do so.

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define PCOUNT 5
6
7   HANDLE hForkMutex[PCOUNT]; // array of mutex representing forks
8
9   int GetForks(HANDLE fork1, HANDLE fork2)
10  {
11     DWORD status;
12
13     status = WaitForSingleObject(fork1,0);
14     if (status)  return 0;
15     // got first resource, try for the second
16     status = WaitForSingleObject(fork2,0);
17     if (status){
18        // failed to get second resource, release all held resources
19        // and abort
20        ReleaseMutex(fork1);
21        return 0;
```



```
22     }
23     //else we have successfully gotten both resources,announce success
24     return 1;
25  }
26
27  VOID Philosopher(LPVOID id)
28  {
29     int left = (int) id;
30     int right = (left + 1) % PCOUNT;
31     int status = 0;
32
33     while(1){
34        // pick up both forks at once, or nothing at all
35        status = GetForks(hForkMutex[left],hForkMutex[right]);
36        if (status == 0){
37           printf("Philosopher #%d: couldn't get forks, try again...\n",
38                  (int) id);
39           continue; // did not get both forks, try again
40        }
41           printf("Philosopher #%d: picked up forks & started eating\n",
42                  (int) id);
43           ReleaseMutex(hForkMutex[left]);
44           ReleaseMutex(hForkMutex[right]);
45           printf("Philosopher #%d: released forks and now thinking\n",
46                  (int) id);
47     }
48  }
49  void main()
50  {
51     HANDLE hThreadVector[PCOUNT];
52     DWORD ThreadID;
53     int i;
54
55     for (i=0; i<PCOUNT; i++)
56        hForkMutex[i] = CreateMutex(NULL, FALSE, NULL);
57     for (i=0; i<PCOUNT; i++)
58        hThreadVector[i] = CreateThread (NULL, 0,
59                (LPTHREAD_START_ROUTINE)Philosopher,
60                (LPVOID) i, 0, (LPDWORD)&ThreadID);
61     WaitForMultipleObjects(PCOUNT,hThreadVector,TRUE,INFINITE);
62  }
```

In practice, this solution can lead to inefficient resource usage. For example, suppose a batch job requires a plotter and a printer, but it only needs to use the printer after it has plotted data, which takes several hours. During this time, other threads that want the printer cannot be started because the printer is unavailable. Starvation can also occur with this solution, since each thread must be granted all requested resources in order to start, but it is possible that not all are available at once, so a thread might never run.

6.6.2 Allowing the Preemption Condition

In allowing preemption, Havender's strategy requires each thread to release all resources held upon denial of an additional resource request. This

strategy introduces (voluntary) preemption and eliminates deadlock. Below, we modify the dining philosophers program to demonstrate this strategy:

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define PCOUNT 5
6
7   HANDLE hForkMutex[PCOUNT]; // array of mutex representing forks
8
9   VOID Philosopher(LPVOID id)
10  {
11     int left = (int) id;
12     int right = (left + 1) % PCOUNT;
13     DWORD status;
14
15     while(1){
16        status = WaitForSingleObject(hForkMutex[left],0);
17        if (status)  continue;
18        printf("Philosopher #%d: picked up left fork\n",(int) id);
19        // got first resource, try for the second
20        status = WaitForSingleObject(hForkMutex[right],0);
21        if (status){
22           // failed to get second resource, release all held resources
23           // and retry
24           ReleaseMutex(hForkMutex[left]);
25           printf("Philosopher #%d: gave up left fork\n",(int) id);
26           continue;
27        }
28        // else we successfully got the resources
29        printf("Philosopher #%d: picked up forks & started eating\n",
30                  (int) id);
31        ReleaseMutex(hForkMutex[left]);
32        ReleaseMutex(hForkMutex[right]);
33        printf("Philosopher #%d: released forks and now thinking\n",
34                  (int) id);
35     }
36  }
37
38  void main()
39  {
40     HANDLE hThreadVector[PCOUNT];
41     DWORD ThreadID;
42     int i;
43
44     for (i=0; i<PCOUNT; i++)
45        hForkMutex[i] = CreateMutex(NULL, FALSE, NULL);
46
47     for (i=0; i<PCOUNT; i++)
48        hThreadVector[i] = CreateThread (NULL, 0,
49                (LPTHREAD_START_ROUTINE)Philosopher,
50                (LPVOID) i, 0, (LPDWORD)&ThreadID);
51     WaitForMultipleObjects(PCOUNT,hThreadVector,TRUE,INFINITE);
52  }
```


Here, we try to get each of the two forks (lines 9 and 13), and if we fail to get either one (lines 11 and 15), we back off (lines 12 and 17), release all previously held resources (line 16), and start over again.

This technique can lead to wasted work because threads may have to roll back their execution state when releasing held resources. This may be tolerated if it happens infrequently, but it can be very disruptive otherwise. Furthermore, starvation and livelocks are possible, since several threads can simultaneously request resources held by others, have their requests denied, release their resources, acquire resources again, be denied other resources, release their resources... again and again, running but making no progress.

6.6.3 Precluding Circular Waiting

To prevent circular waiting, Havender's third strategy requires that resources be ordered linearly, according to which they are requested, granted, and released. For example, threads holding resource R_i can only make subsequent requests for resource R_{i+1} or later in the ordering scheme. It can be proven that this strategy is deadlock-free, and we can intuitively see from Figure 6–4 that because the resource dependency arrows only flow one way, a closed circuit of dependency leading to deadlock is not possible.

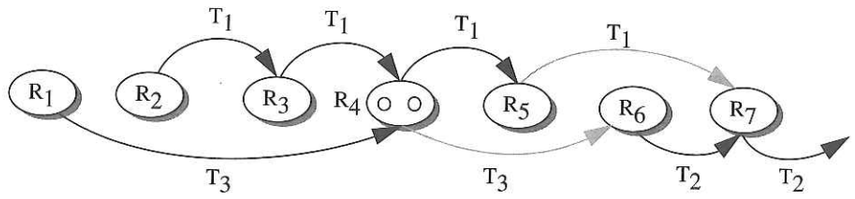

Figure 6–4. Deadlock Prevention by Resource Ordering.

In Figure 6–4, thread T_1 acquires resources R_2, R_3, R_4, and R_5, and is blocked on R_7 because T_2 has it. T_3 has R_1 and R_4, but blocks on R_6, also because of T_2. Eventually, T_2 finishes and releases its resources, allowing T_1 and T_3 to finish.

The following implementation of the dining philosophers program is deadlock-free because we use resource ordering:

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define PCOUNT 5
6
7   HANDLE hForkMutex[PCOUNT]; // array of mutex representing forks
```

```
8   VOID Philosopher(LPVOID id)
9   {
10     int i = (int) id;
11     int j = (i + 1) % PCOUNT;
12     int smaller = (i<j)? i : j;
13     int bigger = (i>j)? i : j;
14
15     while(1){
16        // pick up left fork
17        WaitForSingleObject(hForkMutex[smaller],INFINITE);
18        printf("Philosopher #%d: picked up first fork\n",(int) id);
19        // pick up right fork
20        WaitForSingleObject(hForkMutex[bigger],INFINITE);
21        printf("Philosopher #%d: picked up forks & start eating\n",
22                  (int) id);
23        ReleaseMutex(hForkMutex[smaller]);
24        ReleaseMutex(hForkMutex[bigger]);
25        printf("Philosopher #%d: released forks and now thinking\n",
26                  (int) id);
27     }
28  }
29
30  void main()
31  {
32     HANDLE hThreadVector[PCOUNT];
33     DWORD ThreadID;
34     int i;
35
36     for (i=0; i<PCOUNT; i++)
37        hForkMutex[i] = CreateMutex(NULL, FALSE, NULL);
38
39     for (i=0; i<PCOUNT; i++)
40        hThreadVector[i] = CreateThread (NULL, 0,
41                (LPTHREAD_START_ROUTINE)Philosopher,
42                (LPVOID) i, 0, (LPDWORD)&ThreadID);
43
44     WaitForMultipleObjects(PCOUNT,hThreadVector,TRUE,INFINITE);
45  }
```

In this code, we assign each fork a unique sequence number and force each philosopher thread to take a lower number fork first. Following this fork acquisition order, the philosophers will never deadlock.

Like the two earlier deadlock prevention strategies, this one also suffers from the undesirable characteristics similar to the other two strategies: inflexibility and inefficiency. Not every thread needs resources in the same order, and forcing each to obtain resources in order means that some are allocated well before use, possibly causing other threads to wait for arbitrarily long periods. Furthermore, ordering works well only for a fixed set of resources, and poorly for a changing or dynamic set.

6.7 Deadlock Detection and Recovery

One way of handling deadlock is to let it occur, then detect and break it. When there is a circular dependency, early research on deadlock Holt [1972] showed that we can use directed graphs to model and reveal it more clearly.

6.7.1 Resource Allocation Graph

Let's define a directed graph in which each node represents a thread or a resource to be acquired. Furthermore, let's define an arc from node T_i to node R_j to denote that *thread* T_i *is waiting for resource* R_j, and an arc from R_j to T_i to denote that *thread* T_i *is holding resource* R_j. Using this definition to model our running example, we have the **resource allocation graph** in Figure 6–5.

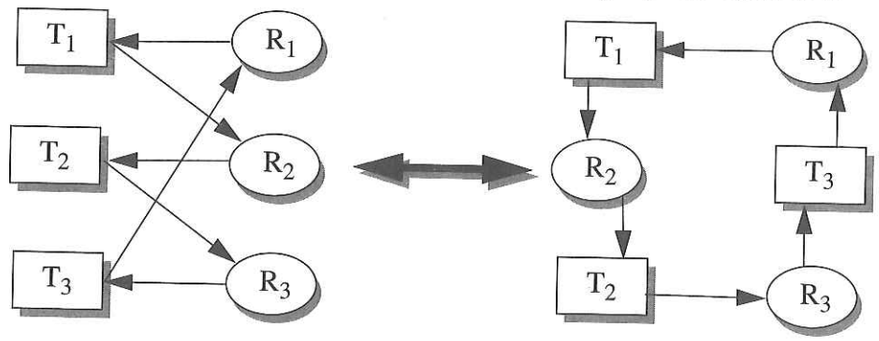

Figure 6–5. Two Views of the Same *Resource Allocation Graph.*

This graph has a closed circuit (a circular path), which means that a condition for deadlock is possible among this set of threads and their resource demands. With a single resource of each type, this closed circuit of resource dependency is a necessary and sufficient condition for deadlock. Note, however, that having the closed circuit does not mean that deadlock will occur every time, but rather that the potential for deadlock exists. The three-thread system in our example clearly shows that despite a cycle in its resource allocation graph, deadlock only happens when the threads' instructions are perfectly interleaved.

With multiple resources of each type, having a closed circuit is still a necessary condition for deadlock, but we need to extend our graph model to account for this property. In our extended graph, the resource node R_j represents a group of one or more resources of the same type. In this case, our graphical depiction of the resource node contains a number or several small circles to indicate the quantity of resources. Naturally, with more than one resource of a type, there can be more than one outgoing arc from the resource R_i to the threads T_i. To be exact, if R_j contains n resources, then there can be from 1 to n outgoing arcs from node R_j

Using the same example, let's change R_1 to be a set of two resource units. We now have a slightly different resource allocation graph, as illustrated in Fig-

ure 6–6. Even with the same scheduling sequence that led to deadlock earlier, no deadlock is possible here since there is no closed circuit in this resource allocation graph. This time, T_3 will be able to have all of its required resources (R_3, R_1) and finish its computation. The release of R_3 and R_1 upon the completion of T_3 enables T_2 to complete and release R_2, which let T_1 finish.

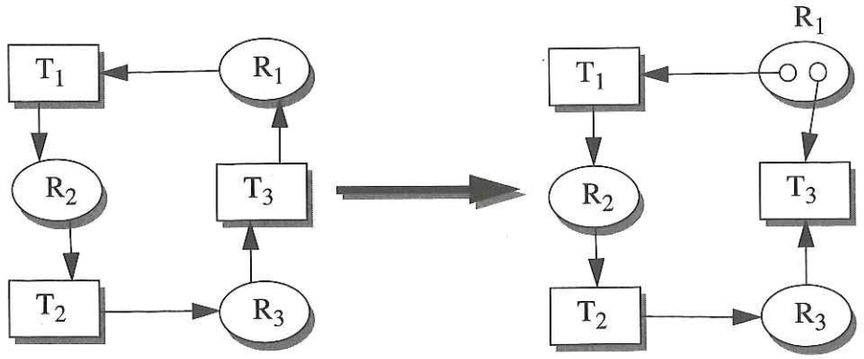

Figure 6–6. A Resource Allocation With Multiple Resource Units of R_1.

6.7.2 Graph Reduction

This analysis can be done with a technique called **graph reduction**, consisting of two phases: *resource allocation* (see Figure 6–7) and arc/node *reduction*. Given a set of threads and their required resources, we can allocate resources to threads in different ways, and for each allocation we can run a reduction algorithm to see if there is a deadlock. Given a system of N threads, the computation cost to determine if it is deadlock free is $O(N^2)$, which is not cheap.

6.7.3 Deadlock Recovery

If deadlock occurs, there are a few recovery techniques we can use. Which to choose depends upon the relative cost and benefits. The common options are:

1. **Termination**. The simplest method of recovery is to terminate one or more threads in the deadlock cycle and allow others to proceed using the freed resources. But how do we decide which threads to terminate? One strategy is to terminate one thread at a time until the deadlock is broken. At the other extreme, we can simply terminate all threads in the resource-dependency cycle. It may also be possible to construct an analysis algorithm that suggests which thread to kill (such as the one holding the most needed resources), but this would add complexity and running time to the program.

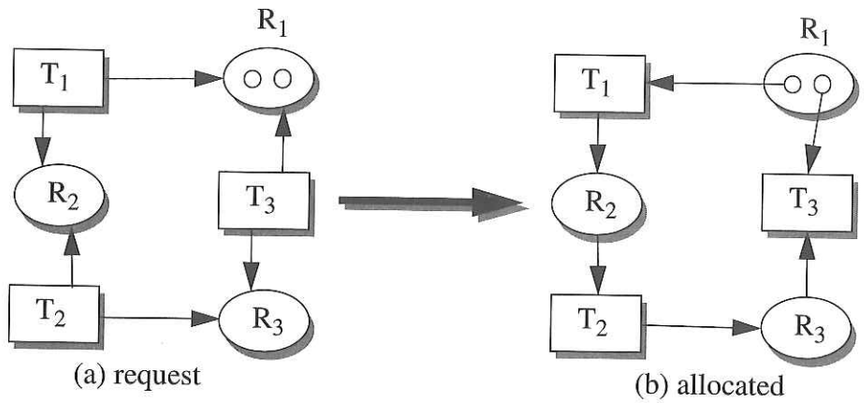

Figure 6–7. Resource Allocation: Granting Resources to Threads.

In general, terminating threads is an effective technique if they can be restarted without much inconvenience (e.g., a print job). In terminating a thread and restarting threads, however, we lose the time and work already done to that point.

2. **Preemption**. In a few cases, we may be able to preempt certain deadlocked threads and give their resources to others. For example, in a particular system we ported to the multithreaded environment [Pham and Garg, 1992], there was a thread that held a mutex lock on heap space to write shared data, and allocated heap memory in the writing process. Sometimes memory was low and needed to be garbage-collected. The garbage collector (GC), however, needed to lock the heap so that it could move the data safely. So the writing thread held a lock on heap space and waited for the GC, while the GC couldn't go on because it waited for the mutex lock on heap to be released by our thread. We had a deadlock. Knowing this behavior pattern, we preempted the writing thread by making it release the heap mutex, call the garbage collector, and terminate by calling itself with its own argument. This sequence of actions effectively preempted the execution of the writing thread in favor of the garbage collector (Figure 6–8).
3. **Roll back from log**: This is the most expensive technique, since it requires threads to have a log of *checkpoints*. When deadlock is detected, we roll back the system state to the checkpoint where the resource causing the deadlock was allocated, in order to try a different allocation of resources. This requires rolling back the state of all threads in the system to that point, often difficult and costly. If we insist upon preventing deadlock at all

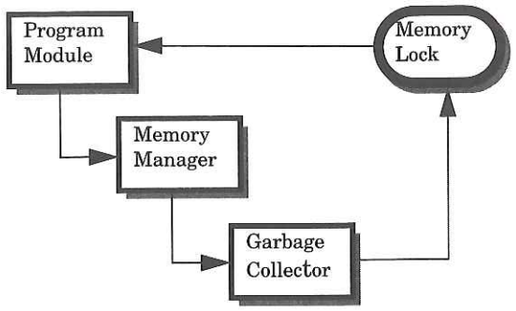

Figure 6–8. Deadlock from Nested Procedure Calls.

cost, we are better off attacking this problem using the deadlock prevention technique, eliminating any possibility of deadlock from the onset.

6.8 Deadlock Avoidance

A less extreme practice than deadlock prevention is deadlock avoidance. Here, the possibility of deadlock still looms but when it is imminent we sidestep it with careful resource allocation, paying the full cost of thread serialization if necessary. By being less insistent about complete deadlock elimination, this technique allows resources to be allocated more flexibly, leading to improved utilization and performance.

Deadlock avoidance techniques are based on the idea that if we know how resource and processing units remain available in the system, we can determine whether granting a particular request will bring us to a state where deadlock is imminent. If so, we deny the request at that time. In effect, we implement an algorithm that can help us to carefully allocate resources and make the right choice every time about whether to grant or deny a resource request.

6.8.1 Safe and Unsafe States

Before exploring deadlock avoidance techniques in greater detail, we must discuss the concept of *safe* and *unsafe* system states. A *safe state* is a state of computation in which there can be no deadlock resulting from the execution of threads and from granting them the requested resources. In other words, with the resources available in the system, there is a way to allocate resources to requesting threads so that they will all be satisfied, eventually complete their execution, and return resources to the system. An *unsafe state*, on the other hand, is one that eventually leads to deadlock.

This concept of safety can be illustrated in graphical form. In Figure 6–9,

we model the progress of two threads, T_1 and T_2, sharing two resources, R_1 and R_2. The X and Y axes indicate the progress of threads T_1 and T_2, respectively. The shaded regions indicate resource requirements of the two threads. The step-like trajectory denotes their joint progress. In uniprocessor systems, only one thread at a time actually runs on the processor, so the progress trajectory is in either the X or Y direction. In multiprocessor machines, parts of the trajectory can be diagonal if the threads make progress in parallel across several processors.

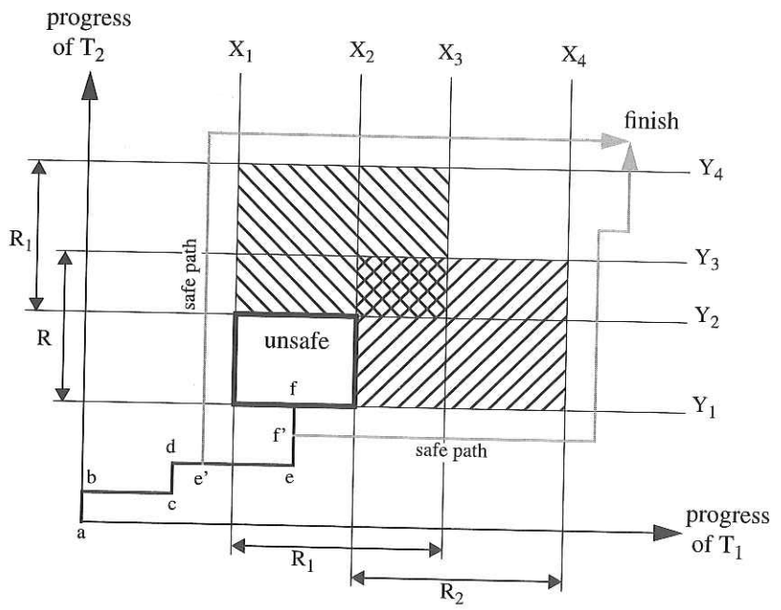

Figure 6–9. Safe and Unsafe States of System Progress.

In this diagram, every point in the trajectory indicates the joint progress of the two threads. In our example, progress can be made in either the positive X or Y direction, depending on the scheduling. When the trajectory crosses the X_1 line, resource R_1 is granted to T_1. At X_2, T_1 gets R_2. At X_3 and X_4, T_1 releases R_1 and R_2, respectively. Similarly, one can describe the progress of thread T_2 in the vertical direction at lines Y_1 to Y_4.

In our example, the progress trajectory first indicates progress made by T_2 (a–b), then T_1 (b–c), and so on, up to point *f*, where we find ourselves at the border of an unsafe state. At this point, if we allow progress to be made in the Y direction, we'll face imminent deadlock (at the shaded region). The reason for deadlock is obvious: thread T_1 has the resource R_1 and will need R_2 in the future. If we allow T_2 to own R_2 by crossing the line Y_1 at point *f*, the two threads will eventually deadlock with T_1 waiting for T_2 to release R_2 and T_2 waiting for T_1 to

release R_1. This is exactly what happens at the X_2 and Y_2 borders of the unsafe regions. Note that it is impossible for a progress trajectory to enter the shaded regions, since that denotes resources needed by both threads. Our mutual exclusion rules ensure that two threads cannot own the same resources at the same time.

So, graphically and intuitively, if we can monitor the progress of thread execution and force progress in either the X or Y direction to avoid the unsafe region, we can avoid deadlock. One way to force progress in a particular direction is by denying resources to a particular thread, thereby making it wait while progress is made in the other direction past the unsafe region.

Using the two-threaded system in Figure 6–9 as an example, at points *a*, *b*, *c*, and *d*, we have complete freedom in granting resources R_1 and R_2 to either thread. As we give R_1 to T_1 moving from *d* to *e*, however, our options become limited because of potential deadlock. At point *e*, our safe choice is to move forward in the X direction past the unsafe region. At *e*, we can still allow T_2 to make progress, but only as far as point *f* without crossing line Y_1 into the unsafe region. At *f*, our safe resource allocation option is limited to denying R_2 to T_2 in favor of T_1, even if it means forcing R_2 to block when T_1 may not need R_2 just yet. In general, a deadlock avoidance algorithm uses its knowledge of resource availability in a system to direct allocation among threads and steer the system's progress away from unsafe states in order to avoid deadlocks.

6.8.2 Banker's Algorithm

Among various deadlock avoidance techniques, Dijkstra's *banker algorithm* is perhaps the most famous. The algorithm is known by this name because it mimics the way in which a banker would grant lines of credit to a group of clients, whose combined financial need is often larger than the banker's liquid assets. The banker must therefore manage his resources so that he can satisfy his customers' borrowing needs. If he is not careful in lending, he may eventually run into a situation where he runs out of money and still has unsatisfied loan applications. The customers then cannot repay their loans since they were not able to borrow enough to finish their investment tasks. Thus, we have a deadlock in which borrowers cannot borrow and the lender cannot lend.

For example, in Figure 6–10 the banker has three customers, Robert, Susan, and Terry, who are granted lines of credit of 5, 8, and 7 units, respectively. (A unit can be a certain number of dollars, but this is not important in our discussion.) The total needed is 20 units, but suppose our banker only has 12 units available to lend. Let us assume that if the banker can grant a customer the full loan requested, the customer is satisfied, uses the money, and repays it at some future time. (In our ideal world, there is no default or interest.) The initial state of this situation is described in Figure 6–10a.

Over time, the customers take out their loans, and this could eventually produce a situation where Robert, Susan, and Terry took out loans of 2, 4, and 3 credit units, respectively, and the banker has only 3 units of capital left to lend (Figure 6–10b). He must carefully examine each incoming loan request to avoid

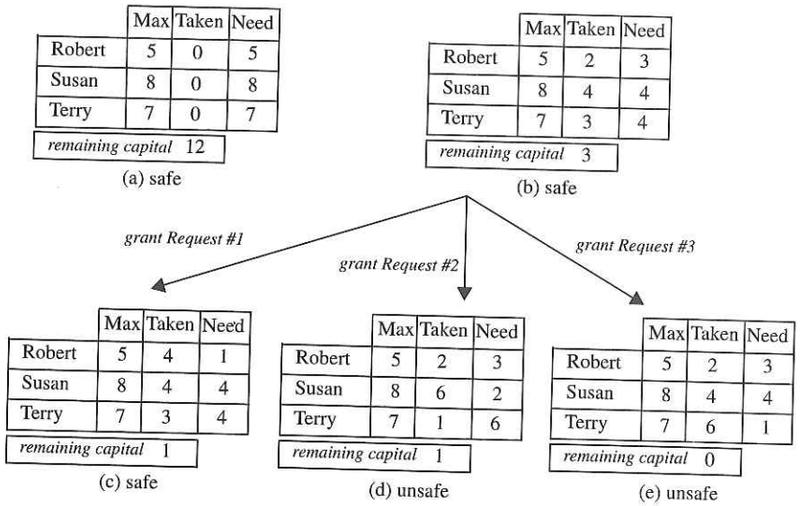

Figure 6–10. Banker's Algorithm: Safe and Unsafe Resource Allocations.

running out of money and being unable to satisfy the borrowing need of at least one customer.

Suppose the banker now receives three loan requests from his customers: Robert asks for 2 units (Request #1), Susan asks for 2 (Request #2), and Terry asks for 3 (Request #3). Our banker must examine each request and check to see if granting it will cause him to be in an unsafe state. The banker knows he is currently in a safe state, since he has 3 units left in his vault, which could be used to satisfy Robert's total need, after which he will repay all 5 resources units when finished. Granting Robert's request will leave our banker with 1 resource unit, still able to satisfy the minimum borrowing need of 1 (from Robert) in the future (Figure 6–10c). This is a safe state. On the other hand, granting 2 units to Susan will leave our banker with 1 resource unit, unable to complete any one loan in the future because the minimum need is 2 units (Figure 6–10d). Similarly, granting 3 units to Terry will deplete the resources available, but there will still be three unsatisfied customers (Figure 6–10e). Thus, the only safe move is to grant Robert's loan request.

This strategy applies directly to computer resources, where customers correspond to threads and credit lines correspond to the resource requirements of each thread, such as disks, shared data, I/O devices, and so on. The banker becomes an algorithm and resource allocation process that we must implement to avoid deadlock.

6.8.3 Banker's Algorithm for Multiple Resources

The banker's algorithm can easily be generalized to handle several kinds of resources. Let us extend our example to illustrate this by allowing the lender

and borrowers to handle loan units of several currency denominations: $5, $10, $20, $50, and $100. Again, a unit can be a fixed number of bills, say 100, but this does not affect our discussion.

Using Figure 6–11a as an illustration, suppose the banker has [5,7,6,9,8] units of the respective currency denominations, and the customers qualify for the lines of credit Robert [3,2,3,5,2], Susan [2,5,4,5,4], and Terry [3,2,2,3,5]. Again, total customer need exceeds the banker's resources, so he must manage the transactions carefully to avoid deadlock.

Over time, Robert, Susan, and Terry take out the loans [0,0,2,3,2], [1,1,3,2,1], and [1,2,0,2,3], respectively. The system comes to a state described by Figure 6–11b.

Suppose the banker now receives three loan requests: (#1) Robert asks for [1,2,0,2,0], (#2) Susan asks for [1,2,1,2,2], and (#3) Terry wants [2,0,1,1,1]. To be safe from deadlock, the banker must compute the state he will be in if he grants each request, and he must approve it only if he will be in a safe state. Granting Robert's request will leave the banker with [2,2,1,0,2] resource units, enough to satisfy at least one customer (Figure 6–11c). Similarly, granting Susan's request will put the banker in a safe state, because his remaining resources will be large enough to fulfill the minimum need of at least customer, namely Robert (Figure 6–11d). On the other hand, granting Terry's request will eventually cause deadlock because the banker's remaining resources will not be enough for any of his customer's needs (Figure 6–11e). Thus, the banker can safely grant either Robert's or Susan's loan request. Again, the strategy applies directly to multiple types of computer resources, such as file descriptors, disk arrays, and so forth.

6.8.4 Limitations of Bankers' Algorithm

The *banker's algorithm* is simple and elegant, but there are some serious weaknesses that prevent its use in practice. As you might have observed in our running example, the algorithm requires that we know a priori:

- The number of resources
- The number of resource users
- The maximum resource need of each user

In a modern computing system, users and resources change dynamically, so these requirements are restrictive and impractical. It is also very hard to require each thread or process to specify in advance the resources it will need. Finally, there is no simple way to determine how long a thread must wait until the system can safely grant its request. This may be unacceptable for real-time systems.

Despite its weaknesses, the algorithm remains theoretically interesting and important because it is occasionally practical and provides a standard against which to compare other algorithms.

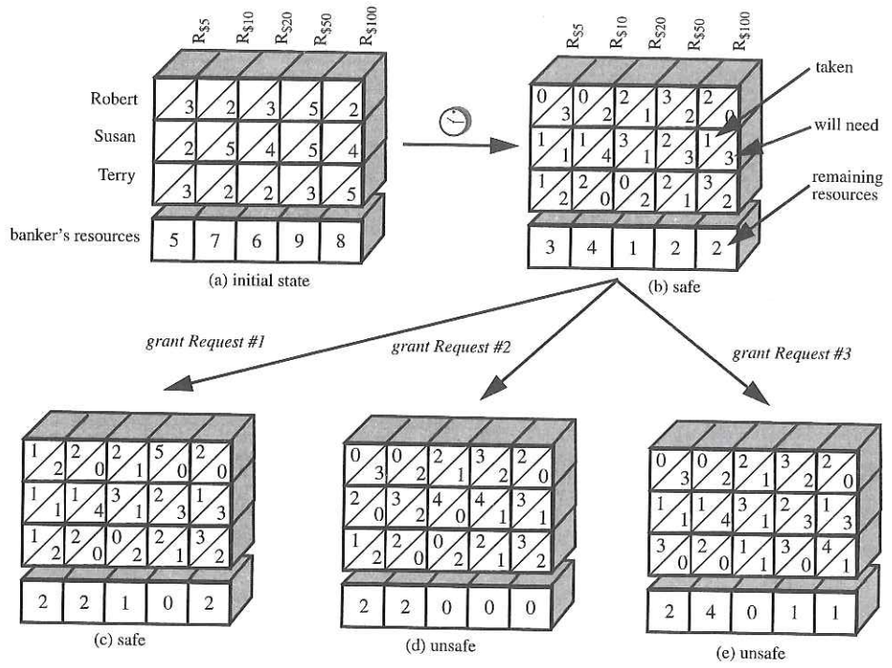

Figure 6–11. Banker's Algorithm for Multiple Resources.

6.9 Summary

This chapter illustrated several examples of deadlock, listed the conditions under which deadlock occurs, and discussed various deadlock handling techniques. Deadlock is a situation in which two or more threads wait for conditions that can only be established by the others. Four conditions must be present for deadlock to occur: mutual exclusion, hold and wait, no preemption, and circular wait.

There are several ways to handle deadlock. One is to ignore it altogether. For deadlock detection there are algorithms to search for deadlocked threads, allowing us to preempt, roll back, or simply terminate the deadlocked threads. Resource allocation graphs and the graph reduction technique are useful to identify a potential deadlock situation among a group of processes or threads. Deadlock prevention can be achieved by precluding the conditions for deadlock at every point in the program. The deadlock avoidance technique seeks to improve resource utilization by not ruling out all possibilities of deadlock, but rather monitor and control a system's resource allocation to steer it away from a potentially deadlocked (unsafe) state. Finally, the concept of state safety and Dijkstra's

banker's algorithm for single and multiple resources were illustrated by two examples.

6.10 Bibliographic Notes

The concept of system deadlock was studied actively during the late 1960s and throughout the 1970s to early 1980s. The four conditions for deadlock were defined by Coffman et al. [1971]. The techniques for deadlock prevention originated from the pioneering work of Havender [1968]. The concept of state safety and the banker's algorithm were introduced by Dijkstra [1968]. Haberman [1969] provided proofs on state safety and state transitions for deadlock avoidance. Modeling of resource requests and allocations using graphs was done by Holt [1972]. Much work was done on deadlock detection techniques by Murphy [1968], Newton [1979], and others. The two operating system textbooks, [Deitel, 1990] and [Tanenbaum, 1992], have excellent chapters and references on deadlock. The presentation of Dijkstra's *banker's algorithm* for a single resource was modeled after a very clear explanation in [Tanenbaum, 1992].

6.11 Exercises

1. What are the four conditions for deadlock? Describe them.
2. What are the four ways in which we can handle a system deadlock?
3. Trace the program steps of the five philosophers in Section 6.4 simultaneously picking up forks and show how they can deadlock.
4. In the example of Section 6.6.1, why can't we use the function WaitForMultipleObject to get all the resources at once? What can happen if you use this function?
5. As in question 3, trace the program steps of the five philosophers in Section 6.6.3. Show why they cannot possibly deadlock.
6. Deadlock prevention strategies can effectively eliminate the possibility of system deadlock. But at what price? Describe the inefficiencies that result from implementing each of the three deadlock prevention strategies.
7. In Figure 6–9, what does the "unsafe" area signify? Why don't we want the execution path to reach point f?
8. What is the Banker's algorithm? How does it work? And, what are its strengths and weaknesses?

6.12 References

1. Coffman, Jr., E. G., Elphick, M. J., and Shoshani, A. (1971). System Deadlocks. *Computing Surveys*, 3(2):67–78.
2. Deitel, H. M. (1990). *An Introduction to Operating Systems*. Addison-Wesley, Reading, MA.
3. Dijkstra, E. W. (1968). Cooperating sequential processes. In Genuys, F., editor, *Programming Languages*. Academic Press, New York.

4. Haberman, A. N. (1969). Prevention of System Deadlocks. *Communications of the ACM*, 12(7):373–377.
5. Havender, J. W. (1968). Avoiding Deadlock in Multitasking Systems. *IBM Systems Journal*, 7(2):74–84.
6. Holt, R. C. (1972). Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196.
7. Murphy, J. E. (1968). Resource Allocation with Interlock Detection in a Multitask System. In *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 33, pp. 1169–1176.
8. Newton, G. (1979). Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography. *ACM Operating Systems Review*, 13(2):33–44.
9. Pham, T. Q. and Garg, P. K. (1992). On Migrating a Distributed Application to a Shared-Memory Multi-Threaded Environment. In *Proceedings of the Summer USENIX*, San Antonio, TX.
10. Tanenbaum, A. (1992). *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ.