CHAPTER 8

Deadlocks

In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads. This situation is called a **deadlock**. We discussed this issue briefly in Chapter 6 as a form of liveness failure. There, we defined deadlock as a situation in which *every process in a set of processes is waiting for an event that can be caused only by another process in the set*.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

8.2 Deadlock in Multithreaded Applications

Prior to examining how deadlock issues can be identified and managed, we first illustrate how deadlock can occur in a multithreaded Pthread program using POSIX mutex locks. The `pthread_mutex init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread mutex lock()` and `pthread mutex unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread mutex lock()` blocks the thread until the owner of the mutex lock invokes `pthread mutex unlock()`.

Two mutex locks are created and initialized in the following code example:

```
pthread mutex t first mutex;
pthread mutex t second mutex;

pthread mutex init(&first mutex,NULL);
pthread mutex init(&second mutex,NULL);
```

Next, two threads—`thread one` and `thread two`—are created, and both these threads have access to both mutex locks. `thread one` and `thread two` run in the functions `do work one()` and `do work two()`, respectively, as shown in Figure 8.1.

In this example, `thread one` attempts to acquire the mutex locks in the order (1) `first mutex`, (2) `second mutex`. At the same time, `thread two` attempts to acquire the mutex locks in the order (1) `second mutex`, (2) `first mutex`. Deadlock is possible if `thread one` acquires `first mutex` while `thread two` acquires `second mutex`.

```
/* thread one runs in this function */
void *do work one(void *param)
{
   pthread mutex lock(&first mutex);
   pthread mutex lock(&second mutex);
   /**
    * Do some work
    */
   pthread mutex unlock(&second mutex);
   pthread mutex unlock(&first mutex);

   pthread exit(0);
}

/* thread two runs in this function */
void *do work two(void *param)
{
   pthread mutex lock(&second mutex);
   pthread mutex lock(&first mutex);
   /**
    * Do some work
    */
   pthread mutex unlock(&first mutex);
   pthread mutex unlock(&second mutex);

   pthread exit(0);
}
```

Figure 8.1 Deadlock example.

Note that, even though deadlock is possible, it will not occur if `thread one` can acquire and release the mutex locks for `first mutex` and `second mutex` before `thread two` attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

8.3 Deadlock Characterization

In the previous section we illustrated how deadlock could occur in multithreaded programming using mutex locks. We now look more closely at conditions that characterize deadlock.

8.3.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
2. **Hold and wait**. A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
4. **Circular wait**. A set $\{T_0, T_1, ..., T_n\}$ of waiting threads must exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, ..., T_{n-1} is waiting for a resource held by T_n, and T_n is waiting for a resource held by T_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 8.5, however, that it is useful to consider each condition separately.

8.4 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.

The first solution is the one used by most operating systems, including Linux and Windows. It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution. Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

Next, we elaborate briefly on the three methods for handling deadlocks. Then, in Section 8.5 through Section 8.8, we present detailed algorithms. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions (Section 8.3.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.5.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the thread should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread. We discuss these schemes in Section 8.6.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 8.7 and Section 8.8.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by threads that cannot run and because more and more threads, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per month), the extra expense of the other methods may not seem worthwhile.

In addition, methods used to recover from other liveness conditions, such as livelock, may be used to recover from deadlock. In some circumstances, a system is suffering from a liveness failure but is not in a deadlocked state. We see this situation, for example, with a real-time thread running at the highest priority (or any thread running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.