Resumen Deadlock

B82957

Marco Ferraro

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**.

“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.”

System Model 8.1

A system consists of a finite number of resources to be distributed among a number of competing threads.

If a system has four CPUs, then the resource type CPU has four instances.

The various synchronization tools discussed in Chapter 6, such as mutex locks and semaphores, are also system resources; and on contemporary computer systems, they are the most common sources of deadlock

Note that throughout this chapter we discuss kernel resources, but threads may use resources from other processes (for example, via interprocess commu□nication), and those resource uses can also result in deadlock. Such deadlocks are not the concern of the kernel and thus not described here.

A thread cannot request two network interfaces if the system has only one.

The request and release of resources may be system calls. Examples are the `request()` and `release()` of a device, `open()` and `close()` of a file, and `allocate()` and `free()` memory system calls.

Request and release can be accomplished through the `wait()` and `signal()` operations on semaphores and through `acquire()` and `release()` of a mutex lock.

A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set.

Deadlock in Multithreaded Applications

```
/* 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);
}
```

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

Livelock 8.2.1

Livelock is another form of liveness failure. It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons. Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, **livelock occurs when a thread continuously attempts an action that fails.**

Livelock is similar to what sometimes happens when two people attempt to pass in a hallway: One moves to his right, the other to her left, still obstructing each other’s progress. Then he moves to his left, and she moves to her right, and so forth. They aren’t blocked, but they aren’t making any progress.

Deadlock Characterization 8.3

Necessary Conditions:

A deadlock situation can arise if the following four conditions hold simultane□ously in a system:

- 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
- 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.
- 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
- Circular wait.

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.

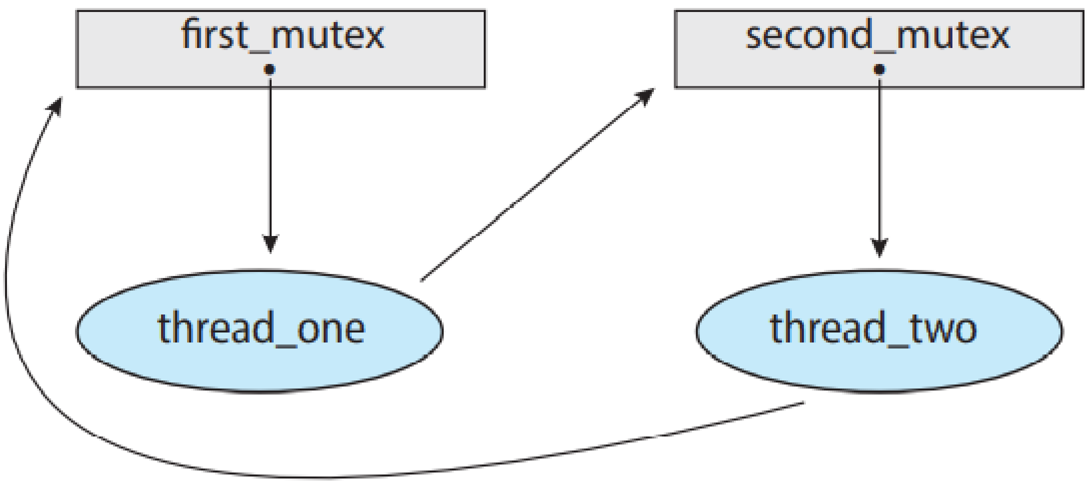

Figure 8.3 Resource-allocation graph for program in Figure 8.1.

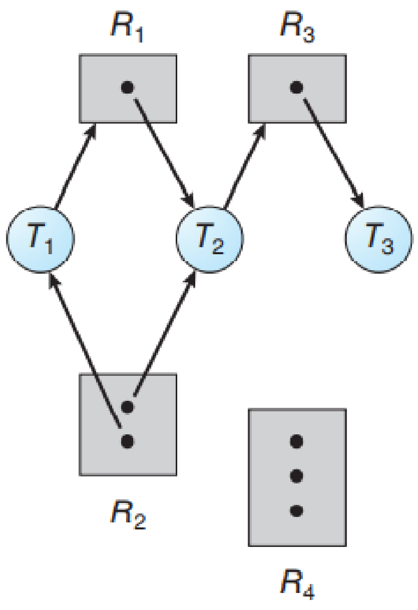

Figure 8.4 Resource-allocation graph.

- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

Resources Instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R

Thread states:

- Thread T1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Thread T2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Thread T3 is holding an instance of R3.

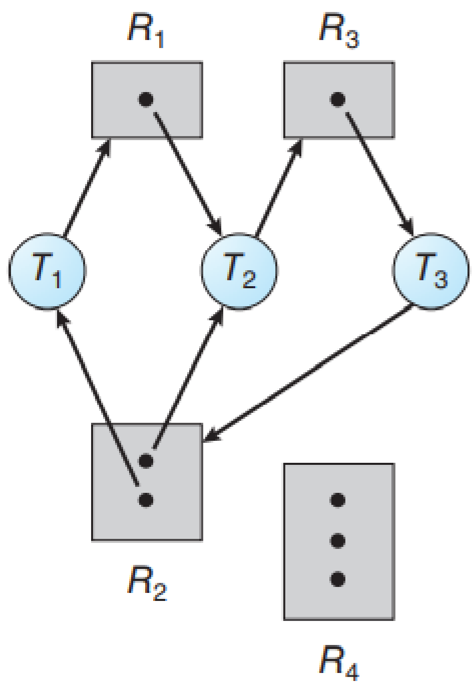

Figure 8.5 Resource-allocation graph with a deadlock.

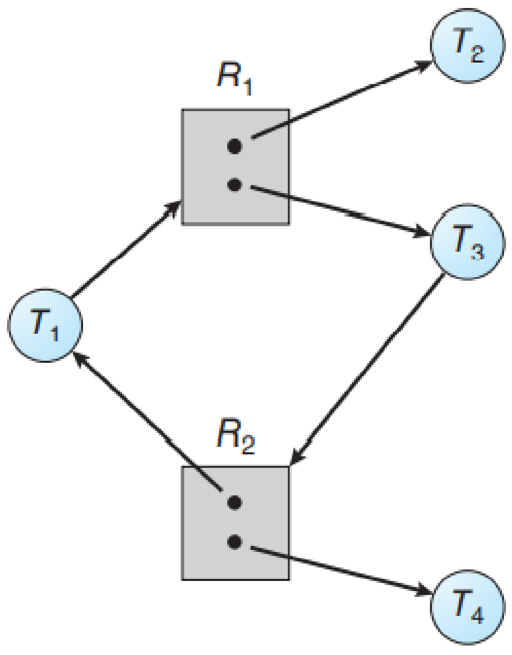

Figure 8.6 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

- 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

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.

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 sys□tem can decide for each request whether or not the thread should wait.