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 this waiting can never end, because the resources requested are held by other waiting threads. This situation is called a deadlock.

In this chapters we explain how deadlock can occur and what methods OS programmers can use prevent or deal with deadlocks.

Contents

Chapter Objectives

- *Illustrate how deadlock can occur*
- *Define the four necessary conditions that characterize deadlock*
- *Identify a deadlock situation in a resource allocation graph*
- *Evaluate the four different approaches for preventing deadlocks*
- *Apply the deadlock detection algorithm*
- *Evaluate approaches for recovering from deadlock*

8.1 System Model

A system consists of a finite number of **resources**, which are used by a number of competing threads. There are several types of resources (types $R_1, R_2 ... R_m$) such as CPU cycles, memory space, I/O devices etc. Each resource type R_i has W_i instances. Each process utilizes a resource as follows:

- **Request** resource.
- **Use** resource after getting access.
- **Release** the resource after using.

8.2 Deadlock in Multithreaded Applications

A set of threads is in a deadlocked state when every thread in the set is waiting for an event (resource acquisition and release) that can be caused only by another thread in the set. The resources are typically logical (mutex locks, semaphores and files).

8.3 Deadlock Characterization

8.3.1 Necessary Conditions

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

- **Mutual exclusion –** At least one resource must be held in a non-sharable 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 he preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
- **Circular wait** – A set $\{T_0, T_1 \dots T_n\}$ 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 $T_2, \dots T_{n-1}$ is waiting for a resource held by T_n and T_n is waiting for a resource held by T_0.

Deadlock is only possible when all four conditions are present together.

8.3.2 Resource Allocation Graph

Deadlocks can be modeled with **system-resource-allocation graphs**, where a **cycle** indicates deadlock. It is a **directed** graph. This graph consists of a set of vertices **V** and a set of edges **E**.

The set of vertices **V** is partitioned into **two** different types of nodes:

- $T = \{T_1, T_2, ..., T_n\}$ it consists of all the **active threads** in the system
- $R = \{R_1, R_2, ..., R_m\}$ it consists of all **resource types** in the system

The set of edges E has the following types of (directed) edges:

- **Request edge:** $T_i \rightarrow R_j$ a directed edge from thread T_i to resource type R_j
  It signifies that thread T_i has requested an instance of resource type R_j and is currently waiting for that resource.
- **Assignment edge:** $R_j \rightarrow T_i$ a directed edge from resource type R_j to thread T_i
  It signifies that an instance of resource type R_j has been allocated to thread T_i

Pictorially, each thread $\mathbf{T_i}$ is represented as a circle and each resource type $\mathbf{R_j}$ as a rectangle. Some resource type $\mathbf{R_j}$ may have more than one instance, that is shown as a dot within the rectangle.

When thread $\mathbf{T_i}$ requests an instance of resource type $\mathbf{R_j}$, a **request edge** is inserted in the graph. When this request is fulfilled, the request edge is transformed to an **assignment edge**. When the thread no longer needs access, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in the **figure 1** depicts the following situation:

- The threads, $\mathbf{T = \{T_1, T_2, T_3\}}$
- The resources, $\mathbf{R = \{R_1, R_2, R_3, R_4\}}$
- The resource requests and assignments edges:
    - o $\mathbf{E = \{T_1 \text{ -> } R_1,\ T_2 \text{ -> } R_3,\ R_1 \text{ -> } T_2,\ R_2 \text{ -> } T_2,\ R_2 \text{ -> } T_1,\ R_3 \text{ -> } T_3\}}$
- Resource instances:
    - o One instance of $\mathbf{R_1}$
    - o Two instances of $\mathbf{R_2}$
    - o One instance of $\mathbf{R_3}$
    - o Three instance of $\mathbf{R_4}$
- Thread states:
    - o $\mathbf{T_1}$ holds one instance of $\mathbf{R_2}$ and is waiting for an instance of $\mathbf{R_1}$
    - o $\mathbf{T_2}$ holds one instance of $\mathbf{R_1}$, one instance of $\mathbf{R_2}$, and is waiting for an instance of $\mathbf{R_3}$
    - o $\mathbf{T_3}$ is holds one instance of $\mathbf{R_3}$

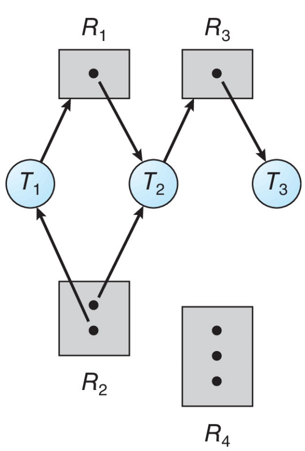

Figure 1 *Resource-allocation graph*

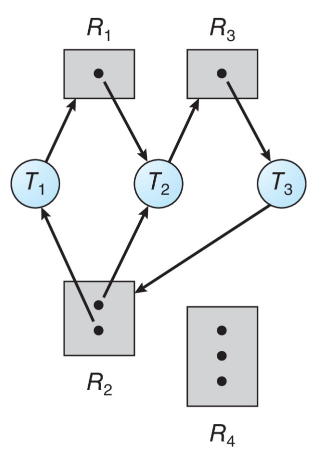

Figure 2 *Resource-allocation graph with a deadlock*

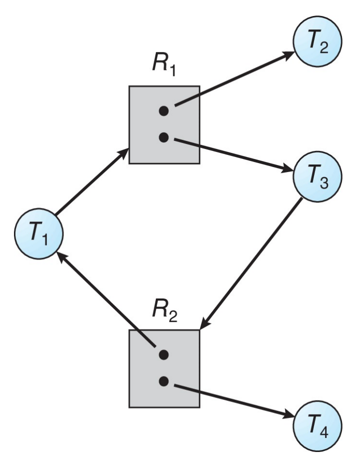

Figure 3 *Resource allocation graph with a cycle but no deadlock*

According to the resource-allocation graph:

- If there is no cycle in the graph, then no thread is in deadlock.
- If the graph contains a cycle:
    - If each resource type has only one instance, then the threads involved are deadlocked.
    - If each resource type has several instances, there is a possibility of deadlock.

In the above scenario (**figure 1**), if the thread $\mathbf{T_3}$ requests an instance of resource type $\mathbf{R_2}$, a request edge $\mathbf{T_3 \text{ -> } R_2}$ is added, the graph now has cycles and the threads $\mathbf{T_1, T_2, T_3}$ are deadlocked.

8.4 Methods for Handling Deadlocks

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

- Ensure that the system will never enter a deadlock state, by using protocols to:
    - Deadlock prevention
    - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system (used by most OS including Linux and Windows, it's up to the application developers to avoid deadlock).

8.5 Deadlock Prevention

Deadlocks can be prevented by ensuring that one of the four necessary conditions (mutual exclusion, hold and wait, no preemption, circular wait) for deadlock cannot occur. Of the four necessary conditions, eliminating the **circular wait** is the only practical approach.

8.6 Deadlock Avoidance

Deadlock can be avoided by using the **Banker's algorithm**, which does not grant resources if doing so would lead the system into an **unsafe** state where deadlock would be possible.

8.6.1 Safe State

When a process requests an available resource, the system must decide if immediate allocation of that resource leaves the system in a **safe state.** A system is in a **safe state** if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of **all** the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by – currently available resources + resources held by all the P_j, with $j < i$. That is:

- If the resource needs of P_i are not immediately available, then P_i can wait until all P_j finish.
- When P_j is finished, P_i can obtain its needed resources, execute, return the allocated resources, and terminate.
- When P_i terminates, P_i+1 can obtain its needed resources, and so on.

If a system is in safe state, there is no deadlocks. If a system is in unsafe state, there is a possibility of occurring deadlock. Deadlock avoidance algorithms ensure that a system never enters an unsafe state.

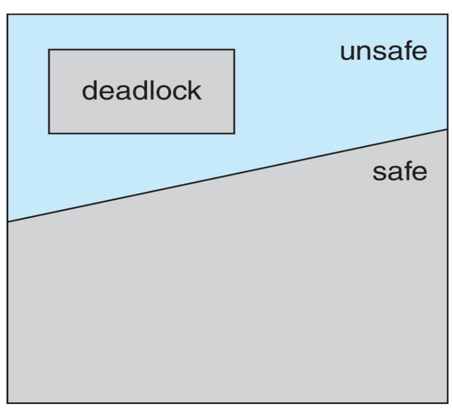

Figure 4 *Safe, unsafe and deadlocked state space*

8.7 Deadlock Detection

A deadlock-detection algorithm can evaluate processes and resources on a running system to determine if a set of processes is in a deadlocked state.

8.7.1 Single Instance of Each Resource Type

If all the resources in the system have only a single instance, then a variant of the **resource-allocation** graph called a **wait-for graph** can be used to detect deadlocks. We obtain this graph from the resource allocation graph by removing the resource nodes and collapsing the appropriate edges.

An edge from T_i to T_j implies that thread T_i is waiting for thread T_j to release a resource that T_i needs. An edge $T_i \rightarrow T_j$ exists in a wait-for-graph if and only if the corresponding resource-allocation graph contains two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ for some resource R_q.

As before, if there is a cycle, there exists a deadlock.

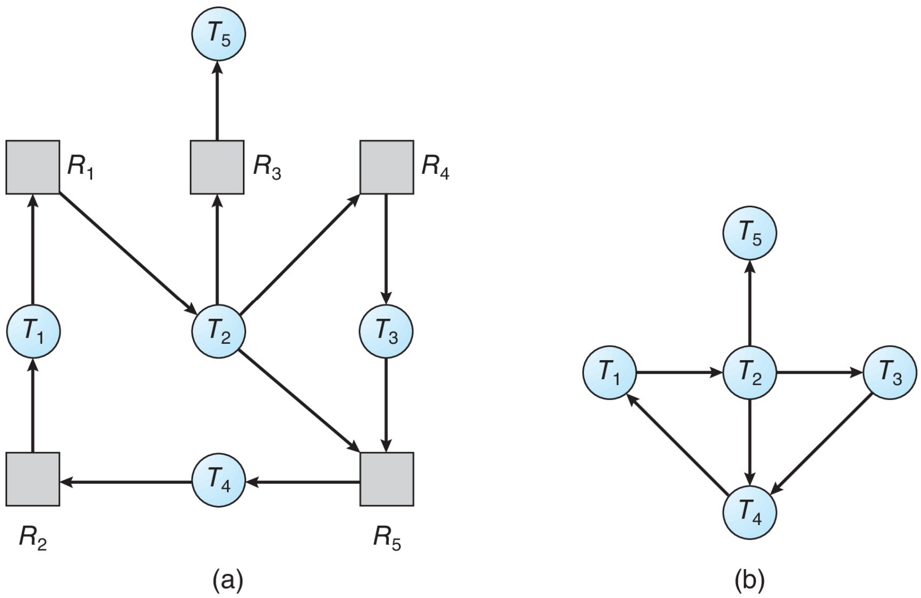

***Figure 5** (a) Resource-allocation graph. (b) Corresponding wait-for graph*

8.8 Recovery from Deadlock

If deadlock does occur, a system can attempt to recover from the deadlock by ether **aborting** one of the processes in the circular wait or **preempting resources** that have been assigned to a deadlocked process.