

Operating System

UNIT –III

Part II - Deadlocks

Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state.

Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

Obviously, the number of resources requested may not exceed the total number of resources available in the system.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

- 1. Request.** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- 2. Use.** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- 3. Release.** The process releases the resource.

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 non- sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- 2. Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- 3. No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- 4. Circular Wait:** A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_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.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource allocation 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: $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

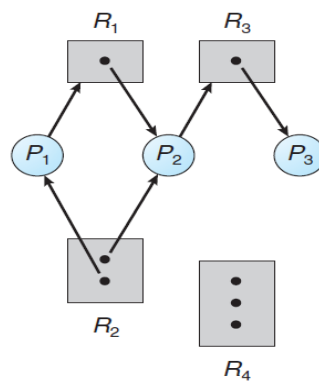
A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph.

When this request can be fulfilled, the request edge is **instantaneously** transformed to an assignment edge.

When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.



Resource-allocation graph

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.

Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

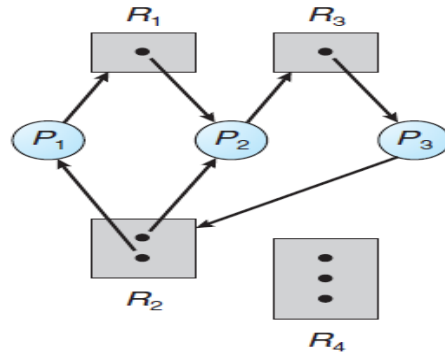
If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P_3 requests an instance of resource type R_2 . Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph (Figure 7.2). At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_1 , P_2 , and P_3 are deadlocked. Process P_2 is waiting for the resource R_3 , which is held by process P_3 . Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2 . In addition, process P_1 is waiting for process P_2 to release resource R_1 .



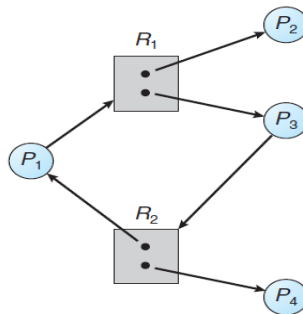
Resource-allocation graph with a deadlock

In the resource allocation graph below we have a cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

However, there is no deadlock. Process P_4 may release its instance of resource type R_2 . That resource can then be allocated to P_3 , breaking the cycle.

Thus, if a resource-allocation graph does not have a cycle, then the system is **not** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.



Resource-allocation graph with a cycle but no deadlock

Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

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

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

With this additional knowledge, the operating system can decide for each request whether or not the process 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 process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise.

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can **prevent** the occurrence of a deadlock.

Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.

If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable. For example, a mutex lock cannot be simultaneously shared by several processes.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them.

Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

The third necessary condition for deadlocks is that there is no preemption of resources that have already been allocated.

To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.

In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources.

If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.

While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur.

Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need.

Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource allocation **state** is defined by the number of available and allocated resources and the maximum demands of the processes.

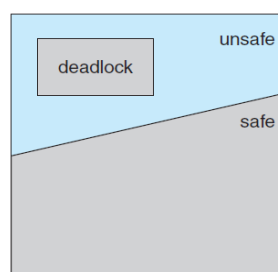
Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**.

A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks; however an unsafe state **may** lead to a deadlock.

As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.



Safe, unsafe, and deadlocked state spaces

For Bankers Algorithm refer to notes given during classwork

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from the deadlock

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

In addition, the number of processes involved in the deadlock cycle may grow.

Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

Another possibility is to let the system **recover** from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait.

The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
2. **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim
2. Rollback
3. Starvation