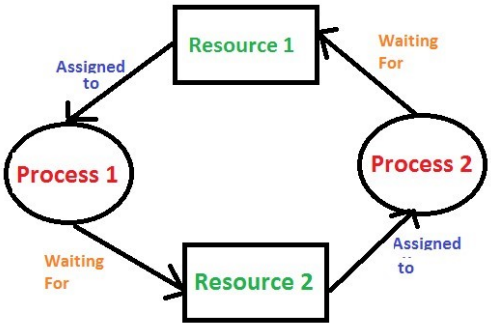
UNIT-III

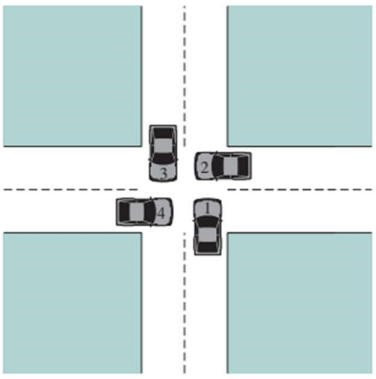
DEADLOCK

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 wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.





System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. If a process requests an instance of a resource type, the allocation of an instance of the type will satisfy the request.

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 requires1 carrying out its designated

task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

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

A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource. A set of processes is in a deadlock state when every process in the set is waiting for an even that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example files, semaphores, and monitors).

DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting.

THE CONDITIONS FOR DEADLOCK

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 model 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.
- **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.
- **No preemption:** Resources cannot be preempted; that is, the process holding it, after that process has completed its task can release a resource only voluntarily.

- **Circular wait:** A set $\{P_0, P_1,\ldots,P_n\}$ of waiting processes must exist such that P0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is 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 $P = \{P_1, P_2,... Pn\}$, the set consisting of the entire active processes in the system, and $R = \{R_1, R_2, ..., R_m\}$, 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->R_j; it signifies that process P_i 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 Pi is denoted by R_j-> P_i; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge P_i -> R_j is called a request edge; a directed edge R_j-> P_i is called an assignment edge.

Pictorially, we represent each process Pi as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square R_j, whereas an assignment edge must also designate one of the dots in the square. When process Pi 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, and as a result the assignment edge is deleted.

The resource-allocation graph depicts the following situation

- The sets P R and E
  - P={p1,p2,p3}
  - R={R1,R2,R3,R4}
  - E={P1-> R1, P2->R3,R1->P2, R2->P2, R2->P1,R3->P3}

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

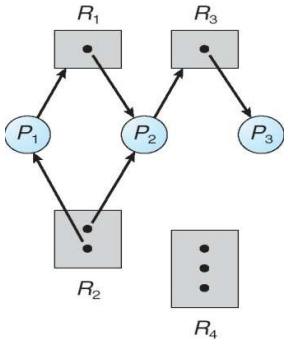


Figure: Resource allocation graph

Process states:

- Process P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.
- ProcessP_2 is holding an instance of R_1 and R_2, and is waiting for an instance of resource type R_3.
- Process P_3 is holding an instance of R_3

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.

For example in above resource allocation graph suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge P_3 -> R_2 is added to the graph. At this point, two minimal cycles exist in the system:

P1-> R1 -> P2 ->R3 ->P3 -> R2 ->P1 P2->R3 -> P3->R2 -> P2

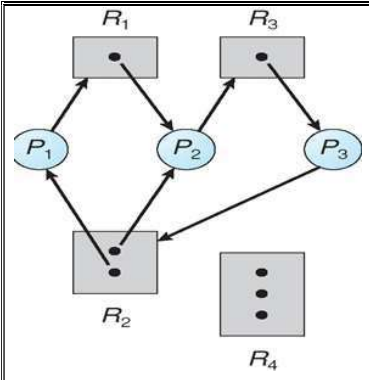


Figure: Resource-allocation graph with a deadlock

Now consider another resource-allocation graph as follows

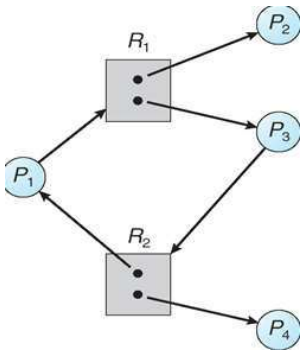


Figure: Resource-allocation graph with a cycle but no deadlock.

In above graph we have cycle. i.e.

P1-> R1 -> P3 -> R2 -> P1

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state.

METHODS FOR HANDLING DEADLOCKS

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

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention is a set of methods for ensuring 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, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we 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, and then a deadlock situation may occur. 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 system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

DEADLOCK PREVENTION

The strategy of deadlock prevention is, simply put, to design a system in such a way that the possibility of deadlock is excluded. We can view deadlock prevention methods as falling into two classes. An indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously. A direct method of deadlock prevention is to prevent the occurrence of a circular wait.

- **Mutual Exclusion**

The mutual-exclusion condition must hold for non sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, 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: Some resources are intrinsically non sharable.

- **Hold and Wait**

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted

simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require. There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

- **No Preemption**

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources. This latter scheme would prevent deadlock only if no two processes possessed the same priority. This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

- **Circular Wait**

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering. To see that this strategy works, let us associate an index with each resource type. Then resource Ri precedes Rj in the ordering if i<j. Now suppose that two processes, A and B, are deadlocked because A has acquired Ri and requested Rj , and B has acquired Rj and requested Ri . This condition is impossible because it implies i<j and j<i.

DEADLOCK AVOIDANCE

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance. In deadlock prevention, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. Deadlock avoidance, on the other hand, allows the

three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests.

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 $<P_1, P_2,...,P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe. A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

Process	Maximum Needs	Current Needs
P1	10	5
P2	4	2
P3	9	2

At time to, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape

drives and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will have all the 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time h, process P_2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are un available, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

Our mistake was in granting the request from process P_2 for 1 more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock. We can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

RESOURCE-ALLOCATION GRAPH ALGORITHM

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge P_i->R_j indicates that process P_i may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process P_i requests resource R_j, the claim edge P_i->R_j is converted to a request edge. Similarly, when a resource R_j is released by Pi, the assignment edge R_j->P_i is reconverted to a claim edge P_i->R_j. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge P_i->R_j to be added to the graph only if all the edges associated with process P_i are claim edges.

Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge P_i->R_j to an assignment edge R_j->P_i does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a

cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of process in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found; then the allocation will put the system in an unsafe state. Therefore, process P_i will have to wait for its requests to be satisfied.

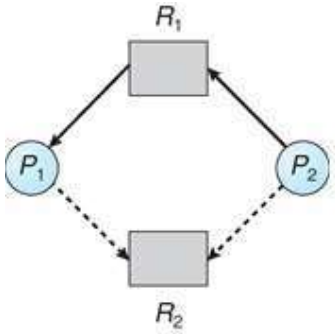


Figure: Resource-allocation graph for deadlock avoidance.

To illustrate this algorithm, we consider the resource-allocation graph of above. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph as shown below.

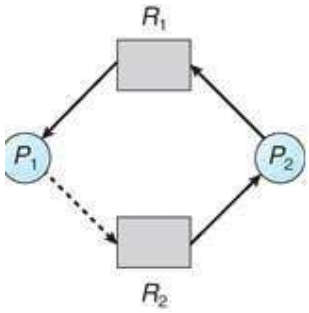


Figure: An unsafe state in resource allocation graph

A cycle indicates that the system is in an unsafe state. If P_l requests R_2, and P_2 requests R_l, then a deadlock will occur.

DEADLOCK DETECTION

- That examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

We elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. A detection-and-recovery scheme requires overhead that includes not only the run-time

costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process Pi to release a resource that Pi needs. An edge $P_i \square P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \square R_q$ and $R_q \square P_i$ for some resource R_q. We present a resource-allocation graph and the corresponding wait-for graph as follows.

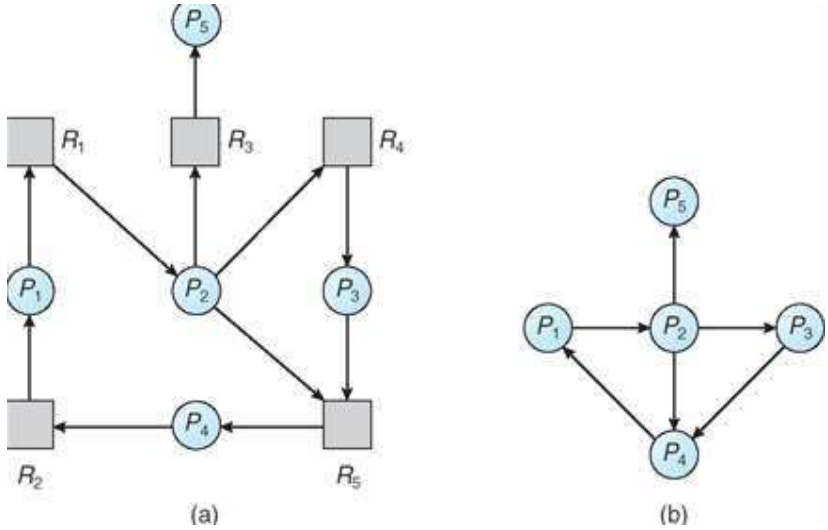


Figure: (a) Resource-allocation graph. (b) Correspo1nding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking CI, deadlock. One

solution is simply to abort one or more processes to break the circular wait. The second option 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. Abort all deadlocked processes: This method clearly will break the dead lock cycle, but at a great expense; these processes may have computed for long time, and the results of these partial computations must be discarded and probably recomputed later. 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.

If the partial termination method is used, then, given a set of deadlock processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. We should abort those processes the termination of which will incur the minimum cost.

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

RESOURCE PREEMPTION

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources for 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:** As in process termination, we must determine the order of pre emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. **Rollback:** when we preempt a resource from a process, we must roll back the process to some safe state, and restart it from that state. Since, in general, it is difficult to determine what a safe state is; the simplest solution is total roll back. Abort the process and then restart it. However, it is more effective to rollback the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.
3. In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.