OS (Deadlock)

★★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.

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.

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

(Sequence of events required to use a resource)

1. Request: Process must request for a resource. If the request cannot be immediately granted (for example, 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 line printer, the process can print on the printer).

3. Release: The process releases the resource.

★★Resources

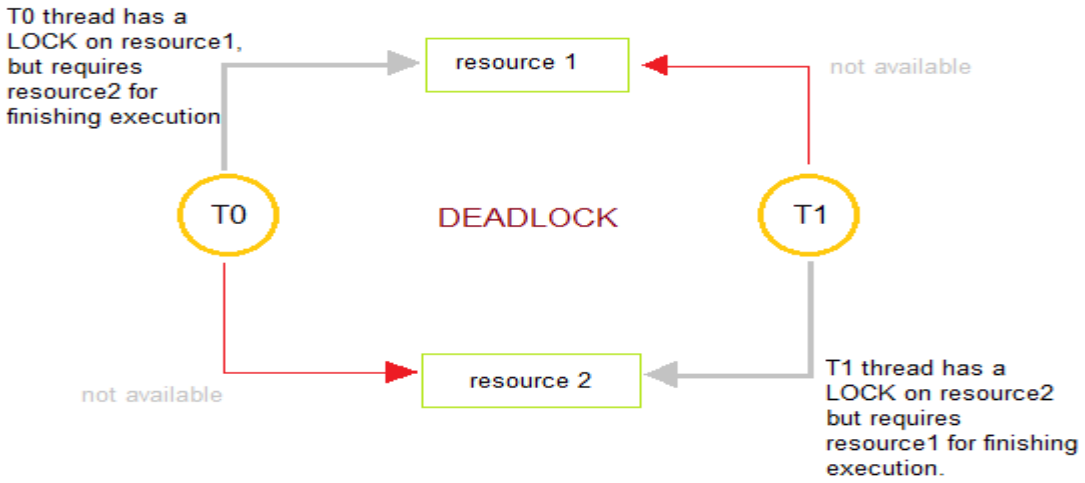
A **resource** can be hardware device (e.g., a tape drive memory) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. Resources come in the following types:

- **Preemptible Resources:** Resources the OS can remove from a process (before it is completed) and give to another process.
- **Non-preemptible Resources:** Resources a process must hold from when they are allocated to when they are released.
- **Shared Resource:** A resource which may be used by many processes simultaneously.
- **Dedicated Resource:** A resource that belongs to one process only.

★★Deadlock Definition

A set of process is in a **deadlock state** when every process in the set is waiting for an event that can be caused by another process in the set. **Example:** A deadlock occurs when two people start crossing the river from opposite sides and meet in the middle.

In other words deadlock is a state where one or more processes are blocked, all waiting on some event that will never occur.



★★Deadlock Modeling [Resource-Allocation Graph]

Deadlocks can be described more precisely in terms of a directed graph called a **system resource allocation graph**. This consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices is partitioned into two types P = {p1, p2, ..., pn}, the set consisting of all the processes in the system, and R = {r1, r2, ..., rm}, the set consisting of all resource types in the system.

An edge (p_i, r_j) is called a request edge, while an edge (rj, pi) is called and assignment edge. Pictorially, we represent each process, pi as a circle and each resource type, rj 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. The resource allocation graph in Fig. 4.3 depicts the following situation.

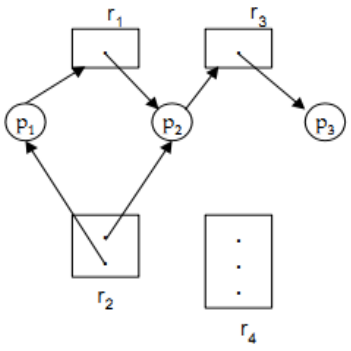


Fig. 4.3 : Resource allocation graph.

●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 r1.
- ✓ Two instances of resource type r2.
- ✓ One instances of resource type r3.
- ✓ Two instances of resources type r4.

●Process states:

- ✓ Process p1 is holding an instance of resource type r2 and is waiting for an instance of resource type r1.
- ✓ Process p2 is holding an instance of r1 and r2 and is waiting for an instance of resource type r3.
- ✓ Process p3 is holding an instance of r3.

A **cycle** in the graph is both a necessary and sufficient condition for the existence of deadlock. To illustrate this concept, let us return to the resource allocation graph depicted in (Fig. 4.3). Suppose that process p3 requests an instance of resource type r2. Since no resource instance is available, a request edge (p3, r2) is added to the graph (Fig. 4.4). At this point two minimal cycles exist in the system.

$$p_1 \rightarrow r_1 \rightarrow p_2 \rightarrow r_3 \rightarrow p_3 \rightarrow r_2 \rightarrow p_1$$

$$p_2 \rightarrow r_3 \rightarrow p_3 \rightarrow r_2 \rightarrow p_2$$

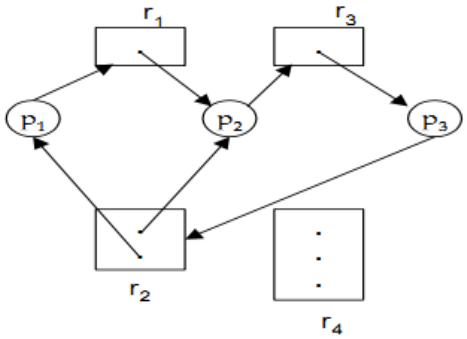


Fig. 4.4 : Resource allocation graph with a deadlock.

So, Processes, p1, p2 and p3 are deadlocked. Process p2 is waiting for the resource r3, which is held by process p3. Process p3 is waiting for either process p1 or p2 to release r2. Meanwhile, process p2 waiting on process p3. In is Process p1 is waiting for process p2 to release resource r1.

★★Necessary Conditions for Deadlock [Deadlock Characterization]

Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks. Coffman (1971) showed that a deadlock situation can arise if and only if the following four conditions hold simultaneously in a system.

1. **Mutual Exclusion Condition:** There exist shared resources that are used in a mutually exclusive manner. 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 Condition:** Processes hold onto resources they already have while waiting for the allocation of other resources.

3. **No preemption condition:** Resources cannot be removed from a process until that process releases.

4. **Circular wait:** A circular chain of processes exists in which each process holds resources wanted by the next process in the chain.

★★Methods for Handling Deadlocks

Principally, 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 deadlock state.

2. We can allow the system to enter a deadlock state, detect it, and recover.

3. 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 a **deadlock prevention or** a **deadlock avoidance** scheme. **Deadlock prevention** is a set of methods for ensuring that at least one of the necessary conditions (*previous section) cannot hold.

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.

We will discuss these schemes in further sections.

★★Deadlock Prevention

We know that for the occurrence of a deadlock, each of the four necessary conditions must hold. By preventing on of the 4 necessary conditions, we can prevent the occurrence of a deadlock.

1. **Mutual Exclusion Condition:** A process never needs to wait for a sharable resource. So, it is not possible to prevent deadlocks by denying the mutual-exclusion condition.

2. **Hold and Wait Condition**: If we can prevent processes that hold resources from waiting for more resources we can eliminate deadlock. One way to achieve this goal is to require all processes to request all their resources before staring execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.
   Advantage: It is easy to code.

3. **No Preemption Condition**: If we can ensure that, no preemption does not hold, we can eliminate deadlocks.

4. **Circular wait Conditions:** By preventing the circular wait from occurring, we can eliminate deadlock. For this, resources are uniquely numbered and processes can only request resources in linear ascending order.

★★Deadlock Avoidance

Deadlock-prevention algorithms, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. 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. This approach defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a deadlock condition can never exist. We discuss **Banker's Algorithm to avoid Deadlock.**

☆☆**Safe State and Unsafe State**

A **safe state** is one in which it can be determined that the system can allocate resources to each process (up to its maximum) in some order (for all pending requests) and still avoid deadlock. A safe state is not a deadlock state. A **deadlock state** is an **unsafe state**. Not all unsafe states are deadlocks but an unsafe state may lead to a deadlock.

The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish, whereas from an unsafe state, no such guarantee can be given.

It will be more clear to illustrate safe state and unsafe state by an example.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P0, P1, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4, and process P2 may need up to 9 tape drives.

Suppose that, at time t0, process P0 is holding 5 tape drives, process P1 is holding 2, and process P3 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t0, the system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition, since process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P0 can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated 1 more tape drive. The system is no longer in a safe state, as there are 3 free drives and all are blocked.

Our mistake was in granting the request from process P2 for 1 more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

★★ Banker's Algorithm in Operating System [Deadlock Avoidance Algorithm]

The scheduling algorithm for **avoiding deadlock** is known as Dijkstra's banker's algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

● Data Structures used to implement the Banker's Algorithm

Some data structures that are used to implement the banker's algorithm are:

1. Available: It is an **array** of length m. It represents the number of **available resources** of each type. If Available[j] = k, then there are k instances available, of resource type Rj.

2. Max: It is an n x m matrix which represents the **maximum** number of instances of each resource that a process can **request**. If Max[i][j] = k, then the process Pi can request atmost k instances of resource type Rj.

3. Allocation: It is an n x m matrix which represents the number of resources of each type currently **allocated** to each process. If Allocation[i][j] = k, then process Pi is currently allocated k instances of resource type Rj.

4. Need: It is a two-dimensional array. It is an n x m matrix which indicates the **remaining resource needs** of each process. If Need[i][j] = k, then process Pi may need k more instances of resource type Rj to complete its task.

Need[i][j] = Max[i][j] - Allocation [i][j]

Banker's algorithm comprises of two algorithms:

1. Safety algorithm
2. Resource request algorithm

☆☆**Safety Algorithm**

A safety algorithm is an algorithm used to find whether or not a system is in its safe state. The algorithm is as follows:

(1) Let Work and Finish be vectors of length m and n, respectively. Initially,

Work = Available

Finish[i] =false for i = 0, 1, ... , n - 1.

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

(2) Find an index i such that both

Finish[i] ==false

$Need_i <= Work$

If there is no such i present, then proceed to **step 4**.

It means, we need to find an unfinished process whose **needs** can be satisfied by the available resources. If no such process exists, just go to step 4.

(3) Perform the following:

$Work = Work + Allocation_i$

Finish[i] = true

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

(4) If Finish[i] == true for all i, then the system is in a safe state.

That means if all processes are finished, then the system is in safe state.

This algorithm may require an order of ***mxn² operations*** *in order to determine whether a state is safe or not.*

☆☆**Resource Request Algorithm**

Now the next algorithm is a resource-request algorithm and it is mainly used to determine whether requests can be safely granted or not.

Let $Request_i$ be the request vector for the process P_i. If $Request_i[j] == k$, then process Pi wants k instance of Resource type Rj. When a request for resources is made by the process Pi, the following are the actions that will be taken:

1. If Requesti <= Needi, then go to step 2; else raise an error condition, since the process has exceeded its maximum claim.

2. If Requesti <= Availablei then go to step 3; else Pi must have to wait as resources are not available.

3. Now we will assume that resources are assigned to process Pi and thus perform the following steps:

Available = Available - Requesti;

Allocationi = Allocationi +Requesti;

Needi = Needi - Requesti;

If the resulting resource allocation state comes out to be safe, then the transaction is completed and, process Pi is allocated its resources. But in this case, if the new state is unsafe, then Pi waits for Requesti, and the old resource-allocation state is restored.

Reference: https://www.studytonight.com/operating-system/bankers-algorithm

☆☆**An Illustrative Example**

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the **matrix *Need*** is defined to be *Max* − *Allocation* and is as follows:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <*P*1, *P*3, *P*4, *P*2, *P*0> satisfies the safety criteria.

Suppose now that process *P*1 requests one additional instance of resource type *A* and two instances of resource type *C,* so *Request*1 = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request*1 ≤ *Available*—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.

Extra - [You should be able to see, however, that when the system is in this state, a request for (3,3,0) by *P*4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by *P*0 cannot be granted, even though the resources are available, since the resulting state is unsafe.]

☆☆**Another Example: Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.**

An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has **12 resources** allocated among processes *P*0, *P*1, and *P*2. The resources are allocated according to the following policy:

	Max	Current	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Currently there are two resources available. This system is in an unsafe state as process *P*1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes *P*0 and *P*2 can complete.

However, it is possible that a process may release resources before requesting any further. For example, process *P*2 could release a resource, thereby increasing the total number of resources to five. This allows process *P*0 to complete, which would free a total of nine resources, thereby allowing process *P*2 to complete as well.

★★ Deadlock Detection Algorithm in Operating System

If a system does not employ either a deadlock-prevention or deadlock-avoidance algorithm, then there are chances of occurrence of a deadlock.

In this case, the system may provide **two things**:

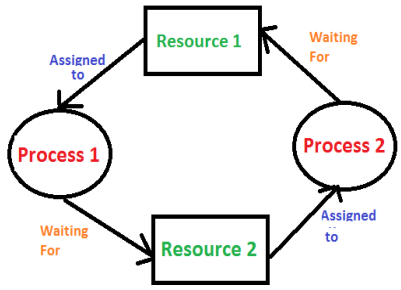
- An algorithm is used to **examine the state of the system** in order to determine **whether a deadlock has occurred**. (**Deadlock Detection**)
- An algorithm that is used to **recover from the deadlock. (Deadlock Recovery)**

☆☆Deadlock Detection Algorithm

***** In most websites, it says that Bankers Algorithm can act as Deadlock Detection Algorithm, see https://tutorialspoint.dev/computer-science/operating-systems/operating-system-deadlock-detection-algorithm**

***** As Alternative, we can prefer following discussion**

1. If resources have a single instance – In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is confirmed.

2. If there are multiple instances of resources – Detection of the cycle is necessary but not sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

Reference: https://www.geeksforgeeks.org/deadlock-detection-recovery/

☆☆**Deadlock Recovery**

Detection is only one part of the solution of deadlock problem. ***After detecting deadlock with the help of detection algorithm***, what should be done next. T***here should be some way in the system needed to recover from deadlock*** and get the system going again. There are two ways for breaking a deadlock:

(1) Recovery through killing process.

(2) Recover through resource preemption.

(1) Recovery through killing process

For eliminating the deadlock by killing a process the following two methods can be used:

- **Kill all Deadlock Processes:** By killing all deadlocked processes will break the deadlock cycle.
- **Kill one process at a time until the deadlock cycle is eliminated:** After each process is killed, the other processes will be able to continue and a deadlock detection algorithm must be involved to determine whether any processes are still deadlocked.

(2) Recover through resource preemption

The following issues are related to preemption: Selection of a Victim and Rollback

Selection of a Victim

We have to determine the resources and processes to be preempted and determine the order of preemption cost.

Case 1: A has a higher priority than B. So, B will have to back up.

Case 2: A needs only 2 more stepping stones to cross the river (i.e. A has already used 98 stepping stones). So, B have to back up.

Case 3: A and B deadlock in the middle of the river. There are ten people behind A. So it would be more reasonable to require B have to back up. Otherwise, eleven people will have to back up.

Rollback

The solution of a rollback is to abort the process and then restart it. Let us go through river crossing example. If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. So, We have to roll the process back to some safe state and restart it from that state.

An effective solution of rollback is to place several additional stepping stones in the river, so that one of the people involved in the deadlock may step aside to break the deadlock.