Deadlock

(UNIT-III :: Part-2)

Syllabus:

1. System Model
2. Deadlock Characterization
3. Deadlock Prevention, Detection and
4. Avoidance
5. Recovery from Deadlock

System Model

- A system consists of **a finite number of resources** to be distributed among a number of competing processes
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances
- Resource types **R1,R2,R3….Rn**
  - CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
- Each Resource type **Ri** has **Wi** instances
  - If a system has **two CPUs**, then the resource type CPU has two instances.
  - The resource type printer may have five instances.

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.

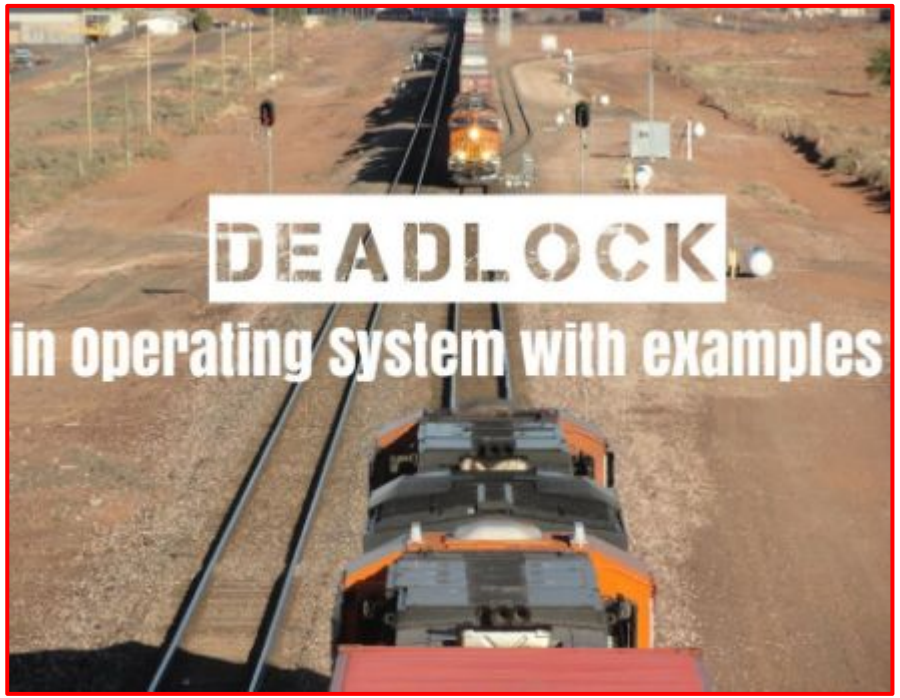
2. Use:

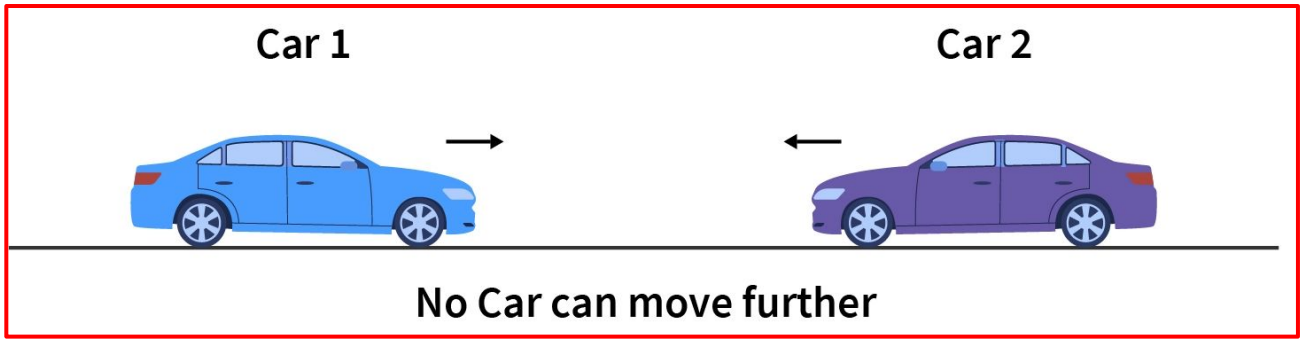
The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

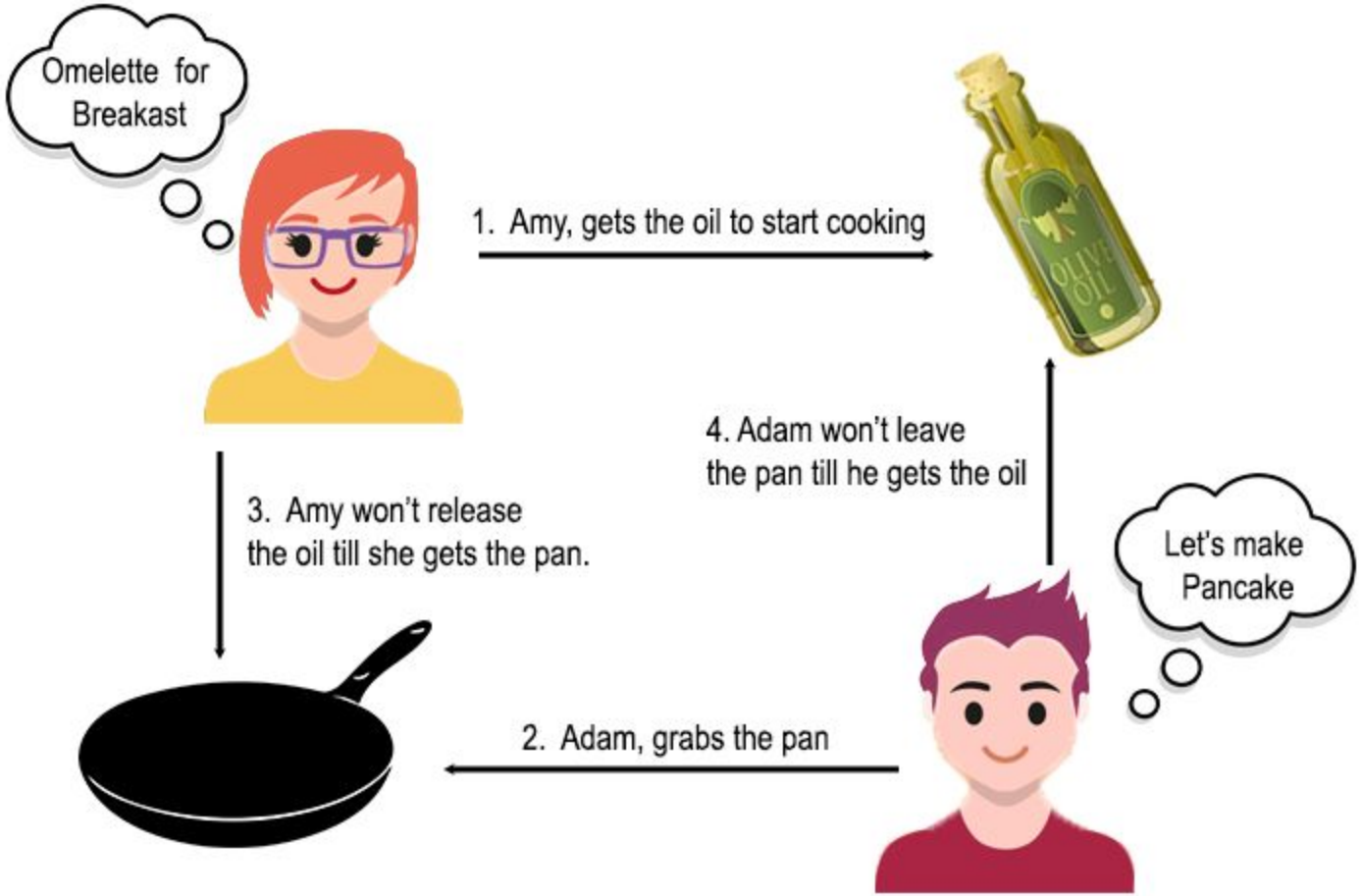
3. Release:

The process releases the resource.

What is Deadlock?


DEADLOCK
in Operating System with examples


Car 1
Car 2
No Car can move further


Omelette for Breakast
1. Amy, gets the oil to start cooking
OLIVE OIL
4. Adam won't leave the pan till he gets the oil
3. Amy won't release the oil till she gets the pan.
Let's make Pancake
2. Adam, grabs the pan

- In operating systems, a deadlock occurs

when two or more processes are blocked and unable to proceed because they are waiting for each other to release resources that they need in order to continue execution.

- Deadlocks can cause a system to become **unresponsive** and may require intervention to resolve.
- A real-time example : Two trains are moving towards each other on a single track. In order for the trains to pass each other safely, they need to use a shared resource, which is the single track.
- *Train A is waiting for Train B to pass so that it can proceed, but Train B is also waiting for Train A to pass so that it can proceed. Neither train can proceed until the other train releases the shared resource, which is the single track, but neither train will release the shared resource until it has completed its task.*

- Consider a system with two processes, P1 and P2, that need to access two shared resources: R1 and R2.
- Assume that P1 acquires R1 and then tries to acquire R2, while at the same time, P2 has acquired R2 and is trying to acquire R1. If both processes reach this point at the same time, a deadlock can occur.
- P1 is waiting for P2 to release R1 so that it can acquire it, while P2 is waiting for P1 to release R2 so that it can acquire it.
- Neither process can proceed without the other process releasing the shared resource it needs, which leads to a deadlock.

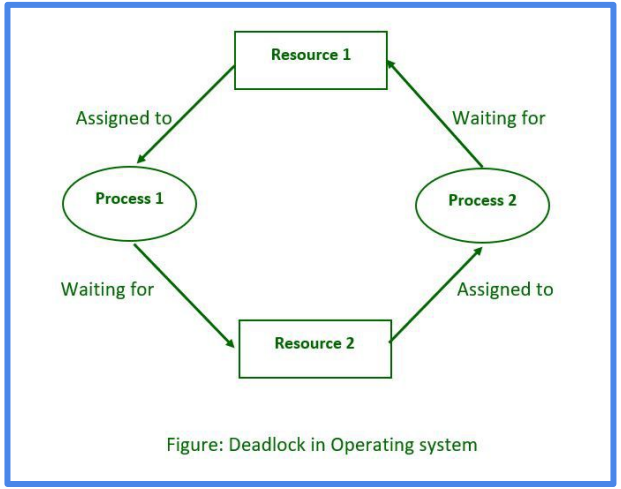
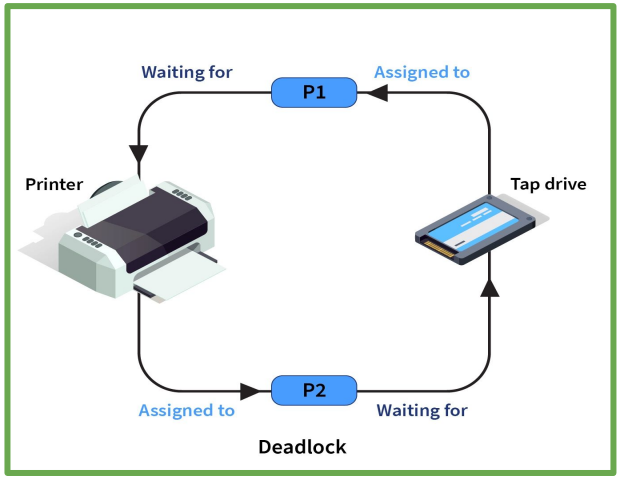


Figure: Deadlock in Operating system



Deadlock Principle

★ A deadlock is **a permanent blocking of a set of processes.**

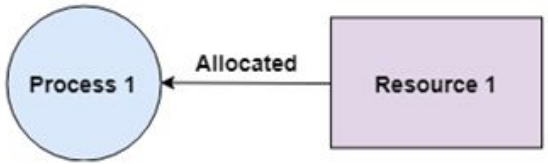
★ a deadlock can happen while threads/processes are competing for system resources or communicating with each other.

Deadlock Characterization

Deadlock can arise if the following four conditions hold simultaneously

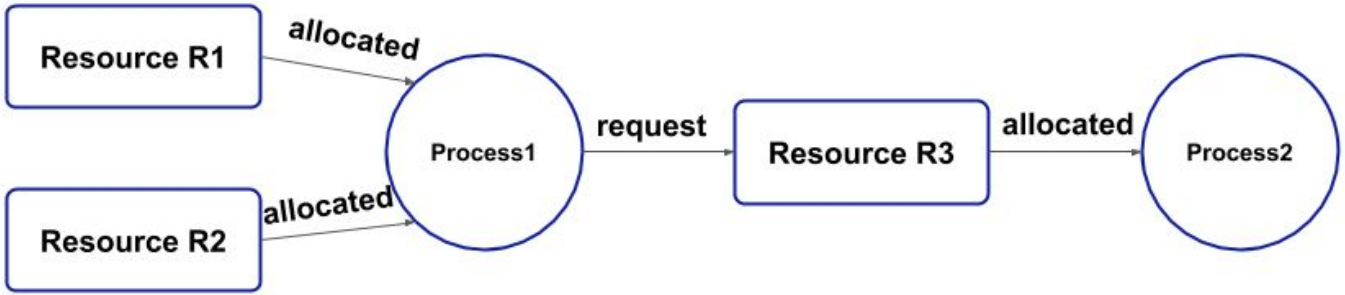
1	Mutual Exclusion
2	Hold and Wait
3	No Preemption
4	Circular Wait

1.**Mutual Exclusion :** Two or more resources are **non-shareable** (Only one process can use at a time) .



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.

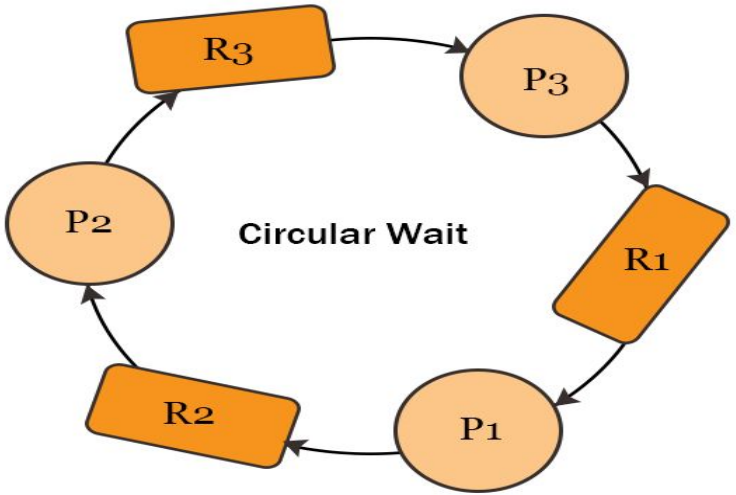


3. **No Preemption :**

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 {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

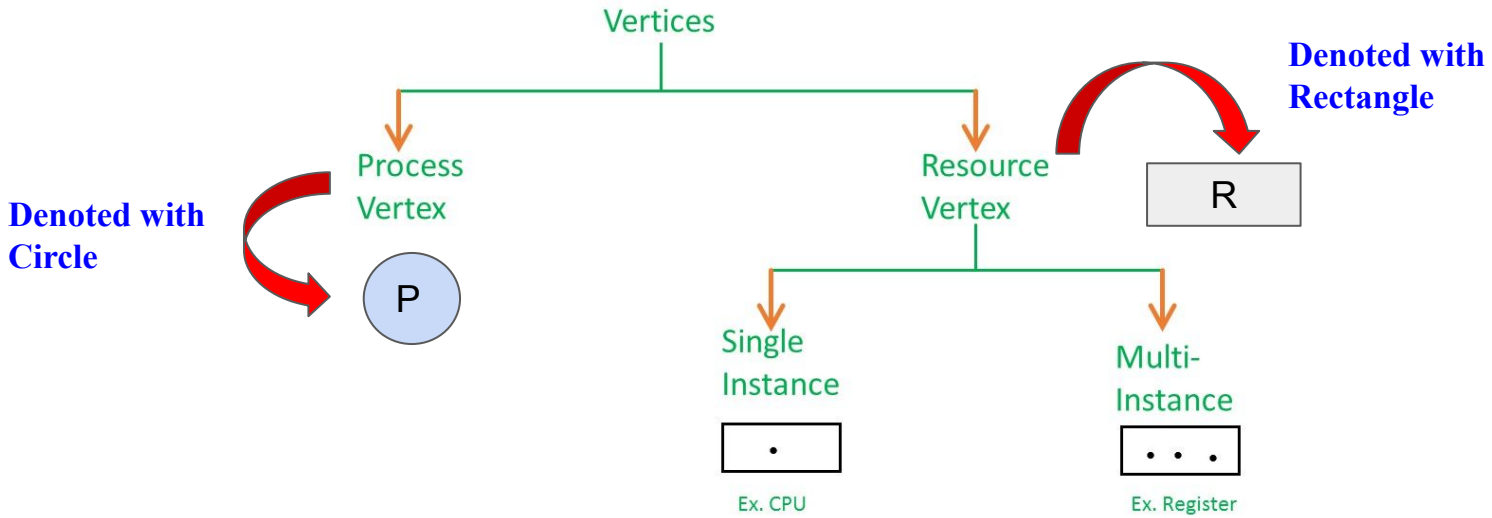


P1 is waiting for P2 to release R2 , P2 is waiting for P3 to release R3 and P3 is waiting for P1 to release R1

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 = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system.
  - R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.
- A directed edge from process Pi to resource type Rj is denoted by Pi → Rj
  - It signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
- A directed edge from resource type Rj to process Pi is denoted by Rj → Pi .
  - It signifies that an instance of resource type Rj has been allocated to process Pi .
- A directed edge Pi → Rj is called a request edge.
- A directed edge Rj → Pi is called an assignment edge.

- Pictorially, we represent each process Pi as **a circle** and each resource type Rj as **a rectangle.**
- Since resource type Rj may have **more than one instance**, we represent each such instance as a dot within the rectangle.

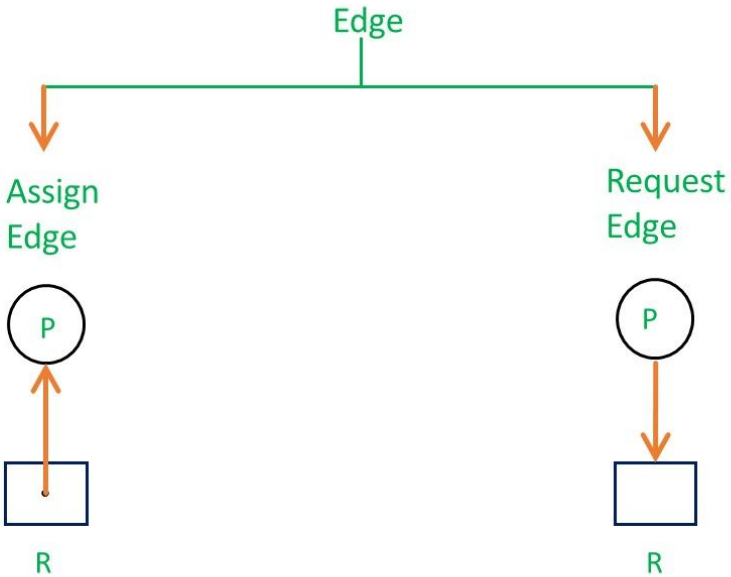


- Now coming to the edges of RAG.There are two types of edges in RAG –

  1. Assign Edge – If you already assign a resource to a process then it is called Assign edge.

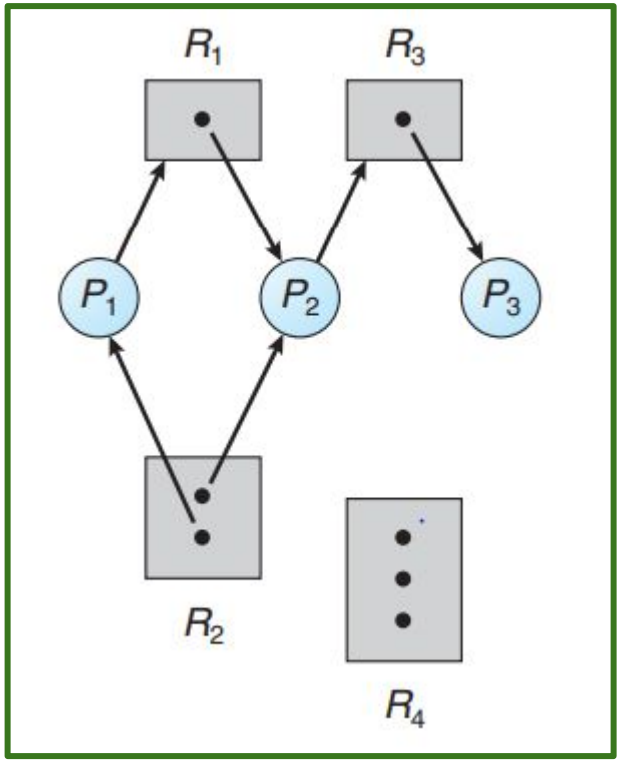
  2. Request Edge – It means in future the process might want some resource to complete the execution, that is called request edge.

.

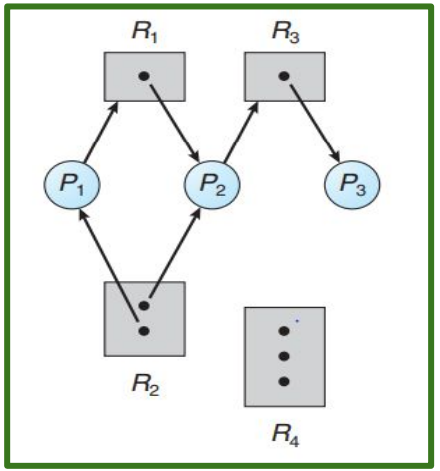


- The resource-allocation graph shown in Figure depicts the following situation.
  - The sets P, R, and E:
  - P = {P1, P2, P3}
  - R = {R1, R2, R3, R4}
  - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource-allocation graph



- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource 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 an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.

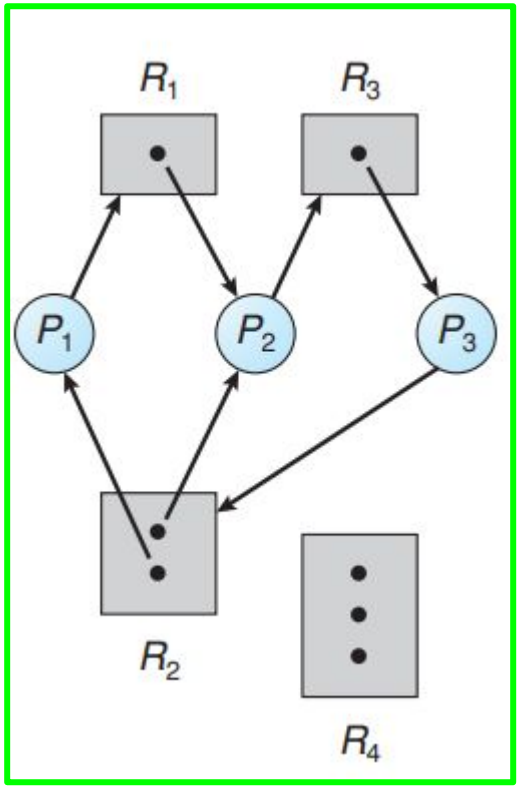
BASIC FACTS

- If the graph contains no cycles ⟹ **No deadlock.**
- If the graph does contain a cycle
  - If each resource type has exactly one instance,

    a deadlock has occurred.

  - If several instances per resource type,

    Possibility of a deadlock



Is this Resource-allocation graph hss deadlock state?

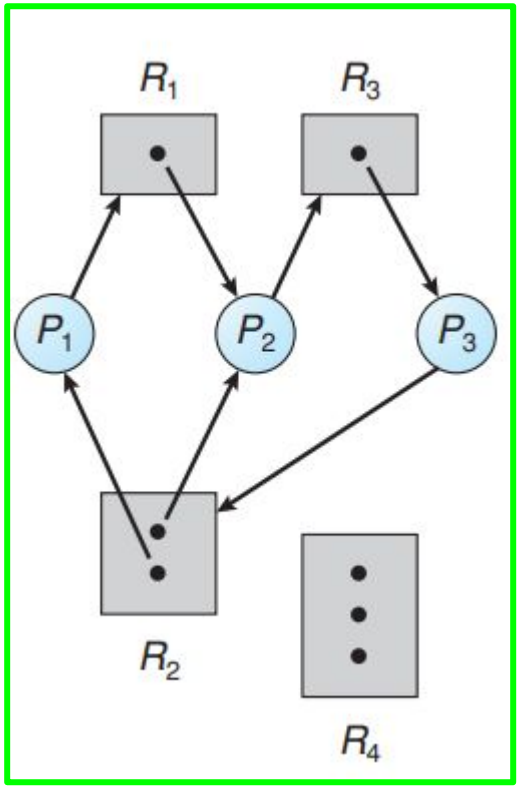
BASIC FACTS

- If the graph contains no cycles ⟹ **No deadlock.**
- If the graph does contain a cycle
  - If each resource type has exactly one instance,

    a deadlock has occurred.

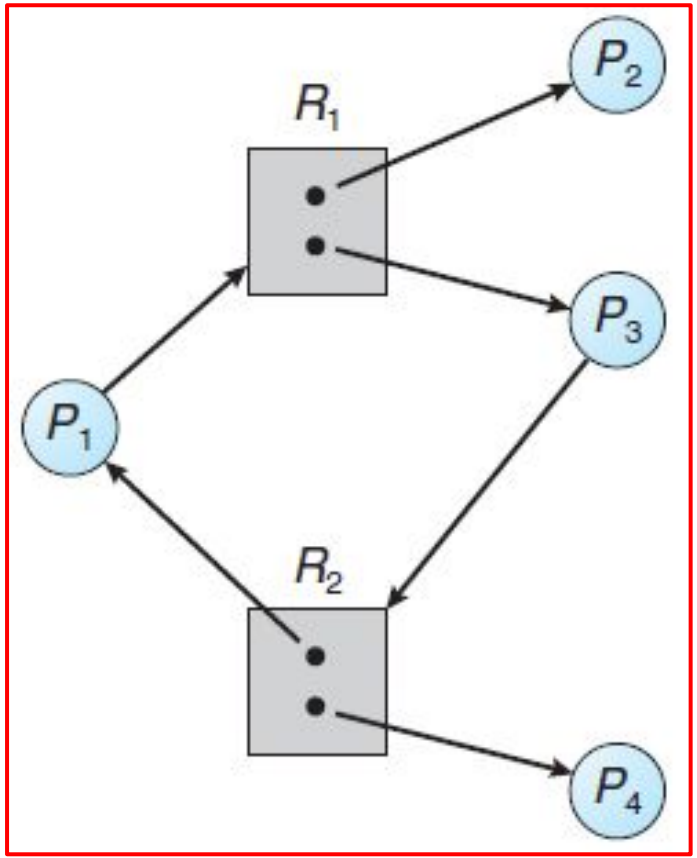
  - If several instances per resource type,

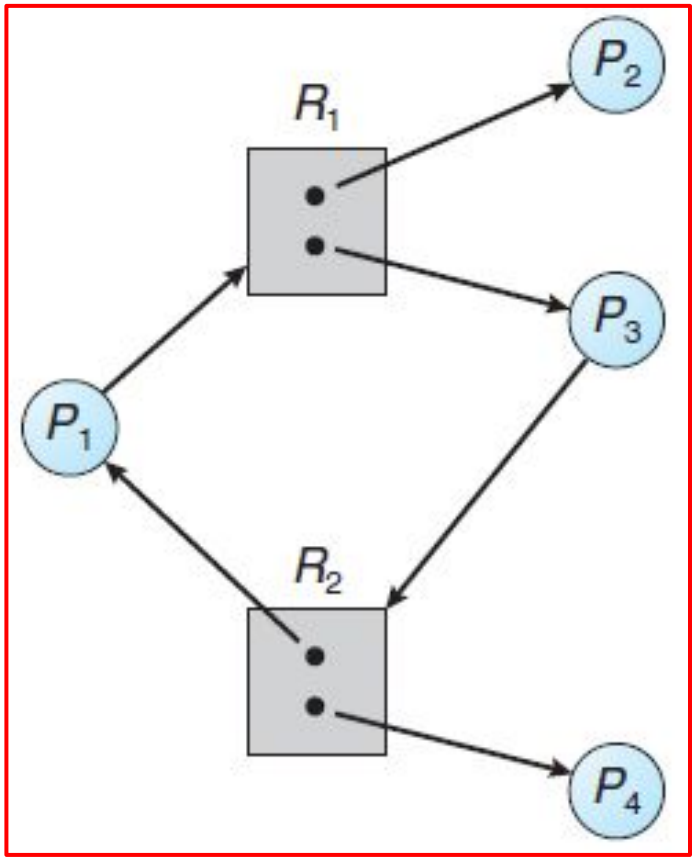
    Possibility of a deadlock



Resource-allocation graph with a deadlock

Is this Resource-allocation graph has deadlock state?





Resource-allocation graph with a cycle but no deadlock

In this example,we also have a cycle:

P1 → R1 → P3 → R2 → P1

- In this example the graph has Cycle , However, there is **no deadlock.**
- ***Observe that process P4 may release its instance of resource type R2.***
- ***That resource can then be allocated to P3, breaking the cycle.***

In summary,

- 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.**

Methods for Handling Deadlocks

There are **four approaches** to dealing with deadlocks.

1. Deadlock Prevention

2. Deadlock Avoidance (**Banker's Algorithm**)

3. Deadlock Detection

4. Deadlock Recovery

Difference between Deadlock Prevention and Deadlock Avoidance ?

Deadlock Prevention

- ➔ Deadlock prevention involves designing the system in such a way that deadlocks cannot occur at all.
- ➔ This can be achieved by ensuring that **at least one of the four necessary conditions for deadlock** (mutual exclusion, hold and wait, no preemption, and circular wait) **is not allowed to occur**.
- ➔ **For example**, if processes are not allowed to hold onto resources while waiting for additional resources, then the hold and wait condition is prevented, and deadlocks cannot occur.

Deadlock Avoidance

- ➔ Deadlock avoidance, on the other hand, allows deadlocks to occur but provides a mechanism to detect and resolve them.
- ➔ This technique requires **a system to have more information about the processes and their resource needs**, and the system must use this information to make decisions about when to allocate resources to processes to avoid the possibility of a deadlock.

Deadlock Prevention

- Eliminate one (or more) of:
  - Mutual exclusion
  - Hold and wait
  - No preemption (i.e. have preemption)
  - Circular wait

Deadlock Avoidance (Banker's Algorithm)

- Deadlock avoidance is a technique used in operating systems **to prevent the occurrence of deadlocks**, which is a situation where two or more processes are blocked, waiting for each other to release resources they need to complete their execution.
- Deadlock avoidance involves designing the system in such a way that **it can detect and prevent situations that can lead to deadlocks.**
- **In deadlock avoidance, the necessary conditions are untouched.**
- **Instead, extra information about resources is used by the OS to do better forward planning of process/resource allocation**
    - Indirectly avoids circular wait

Deadlock Avoidance can be solved by two different algorithms:

★ **Resource allocation Graph Algorithm**

★ **Banker's Algorithm**

- **Safety Algorithm**
- **Resource-Request Algorithm**

- **A Resource Allocation Algorithm:**
  - Ensures that all resources required by a process are **available** before it is allowed to proceed with its execution.
- **Banker's Algorithm**:
  - which is used to ensure that resources are allocated to processes in a **safe sequence** that avoids the possibility of deadlock.
  - The Banker's algorithm is based on the concept of **a safety sequence**, which is a sequence of processes that can complete their execution without causing a deadlock.

Safe State & Unsafe State

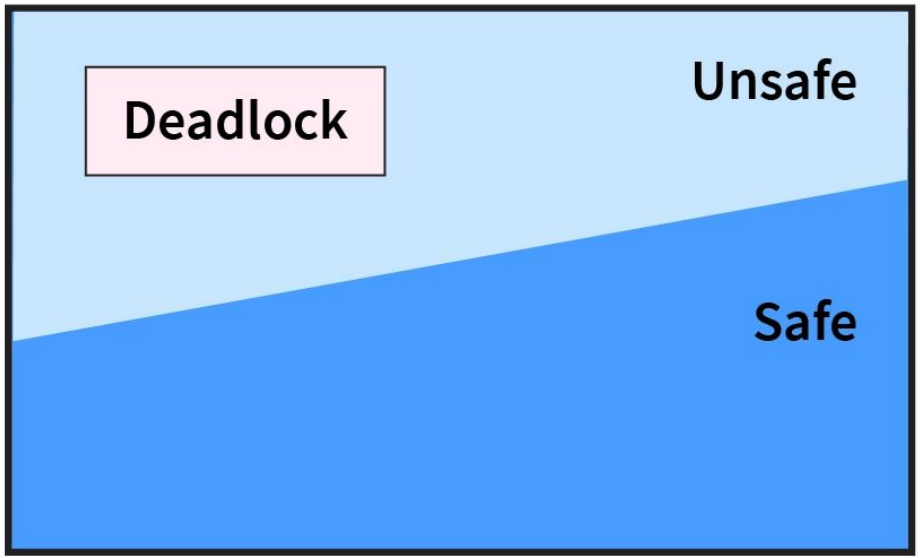
- Operating System avoids Deadlock by knowing the maximum resources requirements of the processes initially, and also Operating System knows the free resources available at that time.
- Operating System tries to allocate the resources according to the process requirements and checks if the allocation can lead to **a safe state or an unsafe state**.

Safe State:

- A safe state is a state in which the system can allocate resources to each process in a way that avoids deadlock.
- **A safe state cannot lead to deadlock.**

Unsafe State:

- If Operating System is not able to prevent Processes from requesting resources which can also lead to Deadlock, then the System is said to be in an Unsafe State.
- **An unsafe state may lead to deadlock.**


Deadlock
Unsafe
Safe

Let's understand the working of **Deadlock Avoidance** with the help of an intuitive example

Process	Maximum Required	current Available	Need
P1	9	5	4
P2	5	2	3
P3	3	1	2

Total Resources =10

Let's consider three processes P1, P2, P3. Some more information on which the processes tells the Operating System are :

- **P1 process** needs a maximum of 9 resources to complete its execution. P1 is currently allocated with 5 Resources and needs 4 more to complete its execution.

- **P2 process** needs a maximum of 5 resources and is currently allocated with 2 resources. So it needs 3 more resources to complete its execution.
- **P3 process** needs a maximum of 3 resources and is currently allocated with 1 resource. So it needs 2 more resources to complete its execution.
- Operating System knows that only **2 resources out of the total available resources are currently free.**
- But only 2 resources are free now. **Can P1, P2, and P3 satisfy their requirements?**

- As only 2 resources are free for now, then only P3 can satisfy its need for 2 resources.
- If P3 takes 2 resources and completes its execution, then P3 can release its 3 (1+2) resources.
- Now the three free resources which P3 released can satisfy the need of P2.
- Now, P2 after taking the three free resources, can complete its execution and then release 5 (2+3) resources. Now five resources are free.
- P1 can now take 4 out of the 5 free resources and complete its execution.
- So, with 2 free resources available initially, all the processes were able to complete their execution leading to Safe State.
- The order of execution of the processes was P3, P2, P1.

Q:: What if initially there was only 1 free resource available?

A: None of the processes would be able to complete its execution. Thus leading to **an unsafe state.**

Example 2 : **The OS is in a *safe* state or Unsafe state ?**

Max no. of resources: 12 tape drives

	Max needs	Current Allocation	Need
P_0	10	5	
P_1	4	2	
P_2	9	2	

Currently, there are 3 free tape drives.

Q :: What is a safe sequence of Process execution ?
Ans::

Example 2 : **The OS is in a *safe* state or Unsafe state ?**

Max no. of resources: 12 tape drives

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

Currently, there are ___ free tape drives.

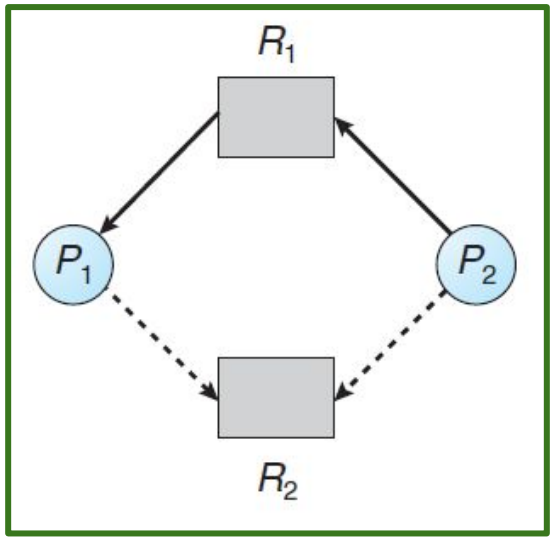
Avoidance Algorithms

- **Single instance of a resource type**
  - **Use Resource-Allocation Graph Algorithm**
- **Multiple instances of a resource type**
  - Use **Banker's Algorithm**

Resource-Allocation Graph Algorithm

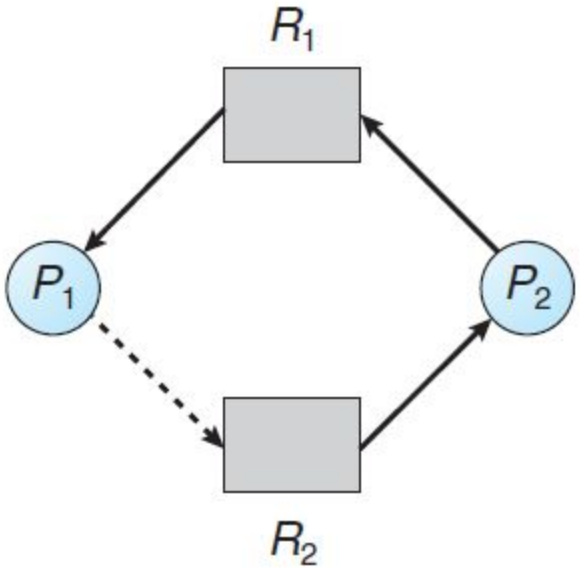
- ➔ The Resource-Allocation-Graph (RAG) algorithm is a deadlock avoidance algorithm used in operating systems. It is used to detect and avoid deadlock situations in a system.
- ➔ If we have a resource-allocation system with only **one instance of each resource type**, we can use a variant of the resource-allocation graph for deadlock avoidance
- ➔ In addition to the **request and assignment edges** already described, we introduce a new type of edge, called **a claim edge**.
- ➔ A claim edge Pi- - - >Rj indicates that process Pi may request resource Rj at some time in the future.
- ➔ This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- ➔ When process Pi requests resource Rj, the claim edge Pi- - - >Rj is converted to a request edge.
- ➔ Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Pi- - - >Rj.

- 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 Pi → Rj to be added to the graph only if all the edges associated with process Pi are claim edges.
-



Resource-allocation graph for deadlock avoidance.

- ➔ Now suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.
- ➔ We check for safety by using a cycle-detection algorithm.
- ➔ An algorithm for detecting a cycle in this graph requires an order of nn^2perations, where n is the number of processes 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.
- ➔ In that case, process Pi will have to wait for its requests to be satisfied
- ➔ Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph (Figure 7.8). A cycle, as mentioned, indicates that the system is in an unsafe state.
- ➔ If P1 requests R2, and P2 requests R1, then a deadlock will occur.



An unsafe state in a resource-allocation graph.

Banker’s Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The deadlock avoidance algorithm (banker's algorithm) is less efficient than the resource-allocation graph scheme.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- It is based on the principle of **avoiding unsafe states** in a system by keeping track of the available resources and the resources required by each process.

Safety Algorithm:

1. Let **Work and Finish** be vectors of length m and n, respectively.

   Initialize **Work = Available** and **Finish[i] = False** for **i = 0, 1, ..., n − 1.**

2. Find an index i such that both

   a. **Finish[i] == False**

   b. **Needi ≤ Work**

3. **Work = Work + Allocation** $_i$

   Finish[i] = True

   Go to step 2.

4. If **Finish[i] == True** for all i, then the system is in **a safe state**

To illustrate the use of the banker's algorithm :

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

P0:

Need ≤ Work

7 4 3 ≤ 3 3 2 , False

P0 must wait.

P1:

Need ≤ Work

1 2 2 ≤ 3 3 2 , True

Work = Work + Allocation

= 3 3 2 + 2 0 0

= 5 3 2

P1 has finished its execution

P2:

Need ≤ Work

6 0 0 ≤ 5 3 2 , False

P2 must wait.

P3:

Need ≤ Work

0 1 1 ≤ 5 3 2 , True

Work = Work + Allocation

= 5 3 2 + 2 1 1

= 7 4 3

P3 has finished its execution

P4: Need ≤ Work

4 3 1 ≤ 7 4 3 , True

Work = Work + Allocation

= 7 4 3 + 0 0 2

= 7 4 5

P4 has finished its execution

P0: Need ≤ Work

7 4 3 ≤ 7 4 5 , True

Work = Work + Allocation

= 7 4 5 + 0 1 0

= 7 5 5

P0 has finished its execution

P2: Need ≤ Work

6 0 0 ≤ 7 5 5 , True

Work = Work + Allocation

= 7 5 5 + 3 0 2

= 10 5 7

P2 has finished its execution

Safe Sequence = P1, P3,P4,P0,P2

Resource-Request Algorithm

Resource-Request Algorithm:

1. If **Request ≤ Need** , go to step 2
2. If **Request ≤ Available**, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
    - ***Allocation[i] = Allocation[i] + Request [i]***
    - ***Available = Available - Request[i]***
    - ***Need[i] = Need[i] - Request [i]***

    If the resulting resource-allocation state is **safe**, the transaction is completed, and process Pi is allocated its resources.
    If the new state is **unsafe**, then Pi must wait for Requesti , and the old resource-allocation state is **restored.**

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Suppose process P1 requests one additional instance of resources = (1,0,2)

Eg: Let P1 request for (1, 0, 2)

Step 1 : **Request ≤ Need**

1 0 2 ≤ 1 2 2 True, goto step 2

Step 2 : **Request ≤ Available**

1 0 2 ≤ 3 3 2 True,goto step 3

Step 3 : **Allocation = Allocation + Request**

= 2 0 0 + 1 0 2 = 3 0 2

Available = Available - Request

= 3 3 2 - 1 0 2 = 2 3 0

Need = Need - Request

= 1 2 2 - 1 0 2 = 0 2 0

Q :: We must determine whether this new system state is safe ?

	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	

P0:

Need ≤ Work

7 4 3 ≤ 2 3 0 , False

P0 has to wait.

P1:

Need ≤ Work

0 2 0 ≤ 2 3 0 , True

Work = Work + Allocation

= 2 3 0 + 3 0 2

= 5 3 2

P1 has finished its execution

P2:

Need ≤ Work

6 0 0 ≤ 5 3 2 , False

P2 has to wait.

P3:

Need ≤ Work

0 1 1 ≤ 5 3 2 , True

Work = Work + Allocation

= 5 3 2 + 2 1 1

= 7 4 3

P3 has finished its execution

P4: Need ≤ Work

4 3 1 ≤ 7 4 3 , True

Work = Work + Allocation

= 7 4 3 + 0 0 2

= 7 4 5

P4 has finished its execution

P0: Need ≤ Work

7 4 3 ≤ 7 4 5 , True

Work = Work + Allocation

= 7 4 5 + 0 1 0

= 7 5 5

P0 has finished its execution

P2: Need ≤ Work

6 0 0 ≤ 7 5 5 , True

Work = Work + Allocation

= 7 5 5 + 3 0 2

= 10 5 7

P2 has finished its execution

Safe Sequence = P1, P3,P4,P0,P2

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:
  a. An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  b. An algorithm to recover from the deadlock
- There are two approaches for deadlock detection
  a. **Single Instances of Each Resource Type**
  b. **Multiple Instances of Each Resource Type**

Single Instances 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 resource nodes and collapsing the appropriate edges.**
- **An edge from Pi to Pj** in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge **Pi → Pj** exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges **Pi → Rq and Rq → Pj for some resource Rq** .
- InFigure (Next Slide) ,wepresent a resource-allocation graph and the corresponding wait-for graph.
- As before, 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 **invoke an algorithm** that searches for a cycle in the graph.

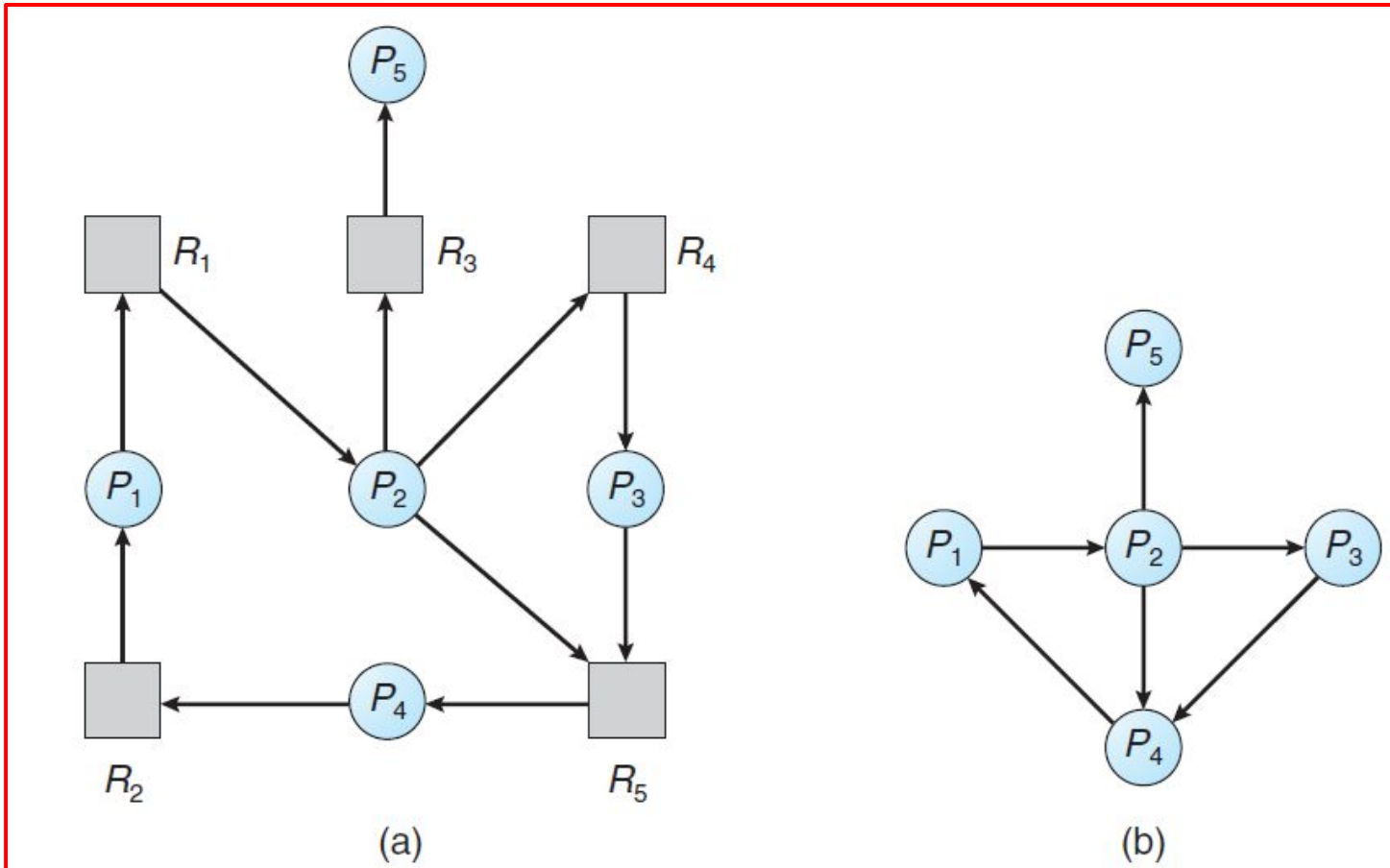


Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Multiple Instances of Each Resource Type

Deadlock Detection Algorithm[Several Instances]:

1. Let **Work and Finish** be vectors of length m and n, respectively.

   Initialize **Work = Available** and **Finish[i] = False** for **i = 0, 1, ..., n − 1.**

2. Find an index i such that both

   a. **Finish[i] == False**

   b. **Request ≤ Work**

3. **Work = Work + Allocation**

   Finish[i] = True

   Go to step 2.

4. If **Finish[i] == False**, then process **Pi is deadlocked**, otherwise the system is in safe state.

we consider a system with five processes P0 through P4 and three resource types A, B, and C. **Resource type A has seven instances, resource type B has two instances, and resource type C has six instances.**

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

P0:

Request ≤ Work

0 0 0 ≤ 0 0 0 , True

Work = Work + Allocation

= 0 0 0 + 0 1 0

= 0 1 0

Finish[0] = True

P0 has finished its execution

P1:

Request ≤ Work

2 0 2 ≤ 0 1 0 , False

P1 has to wait

P2: Request ≤ Work

0 0 0 ≤ 0 1 0 , True

Work = Work + Allocation

= 0 1 0 + 3 0 3

= 3 1 3

Finish[2] = True

P2 has finished its execution

P3:

Request ≤ Work

1 0 0 ≤ 3 1 3 , True

Work = Work + Allocation

= 3 1 3 + 2 1 1

= 5 2 4

Finish[3] = True

P3 has finished its execution

P4: Request ≤ Work

0 0 2 ≤ 5 2 4 , True

Work = Work + Allocation

= 5 2 4 + 0 0 2

= 5 2 6

Finish[4] = True

P4 has finished its execution

P1: Request ≤ Work

2 0 2 ≤ 5 2 6 , True

Work = Work + Allocation

= 5 2 6 + 2 0 0

= 7 2 6

Finish[1] = True

P1 has finished its execution

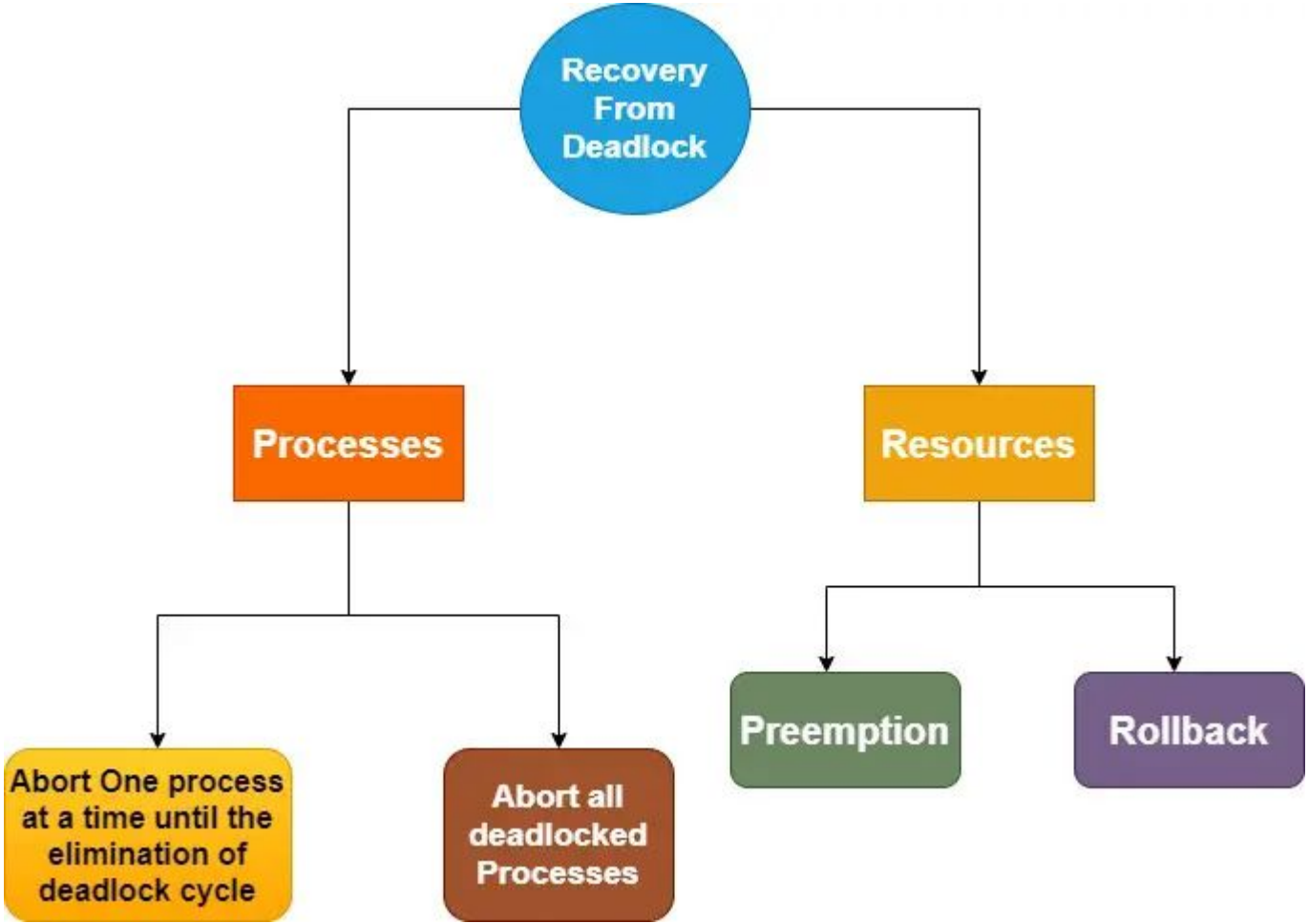
" No deadlock Detected"

Safe Sequence = P0,P2,P3,P4,P1

Deadlock Recovery

➔ 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.


Recovery From Deadlock
Processes
Resources
Abort One process at a time until the elimination of deadlock cycle
Abort all deadlocked Processes
Preemption
Rollback

Process Termination:

To eliminate deadlocks by aborting a process, we use one of two methods

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.
- This approach is not suggestable but can be used if the problem becomes very serious.

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

- Preempt some resources from deadlocked 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.**

Selecting a victim:

★ This strategy involves selecting one of the processes involved in the deadlock and aborting it, allowing the remaining processes to proceed.

★ The selection criteria can be based on a variety of factors such as process priority, resource usage, or time spent waiting.

Rollback:

- Rollback involves **undoing** the work done by a process up to a certain point in time to allow other processes to proceed.
- This can involve rolling back database transactions or undoing changes made by a program.

Starvation:

- Starvation occurs when a process is continually denied access to a resource it needs to proceed.
- This technique involves forcing one of the processes involved in the deadlock to release the resource(s) it is holding.
- This can be done by increasing the priority of other processes that are waiting for the same resource(s) or by introducing a timeout that will cause the process to release the resource(s) after a certain amount of time.