Operating System Module – 3 Deadlock

Dr. Naveenkumar Jayakumar

PRP – 217 4

Deadlock

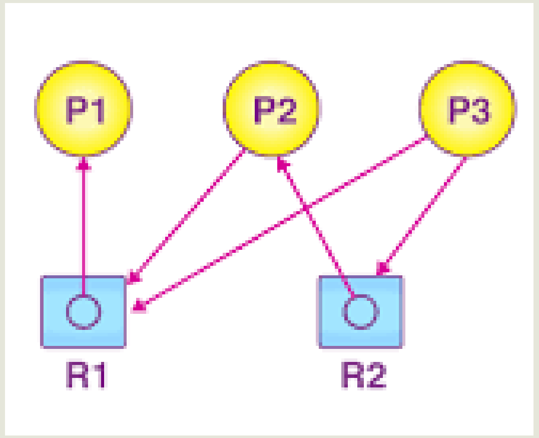
- A deadlock in an operating system is a situation where **two or more processes are permanently blocked** because **each process is waiting for a resource that is held by another process** in the same set.

How Process Consume Resource

Request	• A process requests a resource. If the resource is available, the system grants it. If not, the process waits.
Use	• The process utilizes the resource it has acquired.
Release	• The process releases the resource, making it available for other processes.

Deadlock – System Model

- Deadlocks are commonly modeled using a system **resource-allocation graph**. This **directed graph** consists of two types of nodes:
  - Processes (P): Represented by circles.
  - Resources (R): Represented by rectangles.
  - There are two types of directed edges in the graph:
    - Request Edge: A directed edge from a process Pi to a resource type Rj (Pi → Rj) indicates that process Pi has requested an instance of resource type Rj and is currently waiting for it.
    - Assignment Edge: A directed edge from a resource type Rj to a process Pi (Rj → Pi) indicates that an instance of resource type Rj has been allocated to process Pi.



Deadlock – System Model

- Within each resource type node Rj, there are dots representing each instance of that resource type.
- An assignment edge from Rj to Pi originates from one of these dots, signifying that one instance of Rj is assigned to Pi.
- A system is in a deadlock state if and only if the resource-allocation graph contains a cycle.
- If the graph contains no cycles, then no process is deadlocked.
- If a cycle exists, a deadlock may occur, depending on whether there are multiple instances of a resource type.
- If each resource type has exactly one instance, then a cycle implies a deadlock.
- If there are multiple instances, a cycle indicates the possibility of a deadlock, but not necessarily a guaranteed deadlock.

Deadlock – Conditions

- A deadlock in an operating system occurs when four specific conditions, often called the Coffman conditions, are simultaneously present.

Mutual Exclusion	• This condition states that at least one resource must be non-shareable, meaning only one process can use it at any given time. • If another process requests that resource, it must wait until the resource is released.
Hold and Wait	• A process is holding at least one resource and is simultaneously waiting to acquire additional resources that are currently held by other processes. • For example, Process 1 holds Resource A and is waiting for Resource B, which is held by Process 2.
No Preemption	• Resources cannot be forcibly taken away from a process. • A resource can only be released voluntarily by the process that is holding it, typically after the process has completed its task.
Circular Wait	• This condition exists when a set of processes are waiting for each other in a circular fashion. For instance, Process P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, and so on, until the last process Pn is waiting for a resource held by P1. This forms a closed chain where each process needs a resource held by another process in the cycle.

Deadlock – Management

- It refers to the set of strategies an operating system uses to handle deadlocks.
- These strategies aim to either **prevent deadlocks** from occurring, or to **detect them** and **recover the system** if they do occur.

Deadlock – Management

- Deadlock Prevention
  - This is the most stringent method, which focuses on ensuring that deadlocks are structurally impossible.
  - It works by making sure that **at least one of the four necessary conditions for a deadlock (mutual exclusion, hold and wait, no preemption, or circular wait) can never happen**.
  - By preventing one of these conditions, the system can guarantee that a deadlock will never occur.

Deadlock – Management - Avoidance

- **Deadlock Avoidance**
  - In this strategy, the operating system uses an algorithm to avoid entering a state that could lead to a deadlock.
  - It requires the system to have advance information, such as the **maximum resources each process will need**, to decide if a resource allocation request is safe.
  - The system must remain in a safe state, where there is at least one execution sequence for all processes that does not result in a deadlock.
  - A well-known example of a deadlock avoidance algorithm is the **Banker's Algorithm**.

Deadlock – Management - Detection

- **Deadlock Detection**
  - This approach allows deadlocks to happen, periodically checks for them, and then takes action to resolve them.
  - **Detection:** The system uses an algorithm, such as **analyzing a resource-allocation graph or a wait-for graph**, to determine if a deadlock has occurred.

Deadlock – Management - Recovery

- **Recovery** Once a deadlock is detected, the system can use one of several methods to recover:
  - **Process Termination**
    - Aborting one or more of the processes involved in the deadlock cycle.
  - **Resource Preemption**
    - Forcibly taking a resource from one process and giving it to another.
  - **Rollback**
    - Returning the deadlocked processes to a previous safe state that was saved at a checkpoint.

Deadlock – Management - Ignoring

- **Ignoring Deadlocks**
  - Also known as the **Ostrich Algorithm**, this approach involves simply ignoring the problem.
  - The operating system assumes that deadlocks will occur so infrequently that the performance overhead of implementing prevention, avoidance, or detection is not worthwhile.
  - Many general-purpose operating systems, including variants of UNIX and Windows, use this approach, as it is often cheaper to reboot the system than to implement complex deadlock-handling mechanisms.

Deadlock – Management – Prevention - Strategy

- To prevent deadlocks, the operating system must ensure that at least one of the four necessary conditions—Mutual Exclusion, Hold and Wait, No Preemption, or Circular Wait—can never hold true.

Deadlock - Prevention – Strategy – Eliminating Mutual Exclusion

- Making Resources Shareable The most direct way to eliminate mutual exclusion is to allow multiple processes to access a resource simultaneously.
- This works for resources that can be shared without conflict.
- **For example, a file can be simultaneously accessed by multiple processes if they are all only reading from it**. Since no process has to wait for another to finish, mutual exclusion is not enforced, and that resource cannot be part of a deadlock. However, if any process needs to write to the file, exclusive access is required.

Deadlock - Prevention – Strategy – Eliminating Mutual Exclusion

- **Spooling** For devices that are inherently non-shareable, like printers, a technique called spooling (**Simultaneous Peripheral Operations Online**) can be used.
- Instead of giving a process exclusive control of the printer, its print job is sent to a queue in a special disk area or memory buffer called a spool.
- A separate system process, often called a printer daemon, is the only process that has direct access to the printer.
- It reads the jobs from the queue and prints them one by one, typically in a first-come, first-served order.

Deadlock - Prevention – Strategy – Eliminating Hold & Wait

- **Request all resources at the start** A process must request and be allocated all the resources it will need before its execution begins.
- This approach ensures that the process will never be in a state of holding some resources while waiting for others.
- The main disadvantage of this method is that it can lead to low resource utilization, as resources may be allocated long before they are actually needed. It is also often impractical because it's difficult for a process to know all its required resources in advance.

Deadlock - Prevention – Strategy – Eliminating Hold & Wait

- **Release resources before new requests**
  - This protocol requires a process to release all the resources it currently holds before it can request any additional ones.
  - After releasing its resources, the process can then attempt to re-acquire its old resources along with the new ones it needs.
  - While this prevents the hold and wait condition, it is often inefficient.
  - It can also lead to a problem known as starvation, where a process that needs a popular resource may have to wait indefinitely because the resource is always allocated to some other

Deadlock - Prevention – Strategy – Eliminating No Preemption

- **Preempt on Request Failure**
  - If a process holding some resources requests another resource that cannot be immediately allocated, the operating system can preempt all the resources currently held by that process.
  - These preempted resources are then added to the list of resources for which the process is waiting.
  - The process is only restarted when it can regain its old resources along with the newly requested ones.

- **Preempt from Waiting Processes**
  - If a process requests a resource that is currently held by another process, and that other process is itself blocked and waiting for a different resource, the system can preempt the resource from the waiting process.
  - The resource is then allocated to the requesting process.

Deadlock - Prevention – Strategy – Eliminating Circular Wait

- **Assign a unique number to each resource type**:
  - Every resource in the system is assigned a unique integer value or priority.

- **Enforce an ordered resource allocation policy**:
  - A process is only allowed to request resources in a strictly increasing order of their assigned numbers. For example, if a process is holding resource R(i), it can only request a resource R(j) if the number assigned to R(j) is higher than the number assigned to R(i).

- **Release higher-numbered resources if a lower one is needed**: If a process holds a resource and needs to request another resource with a lower assigned number, it must first release the higher-numbered resource it currently holds.

Deadlock - Detection – Strategy

- **Single Instance of Each Resource Type**:
  - If every resource type has only one instance, deadlock detection is straightforward.
  - The system uses a Resource Allocation Graph (RAG) to model the state.
  - A deadlock exists if and only if the graph contains a cycle. The presence of a cycle is a sufficient condition for a deadlock in this scenario.

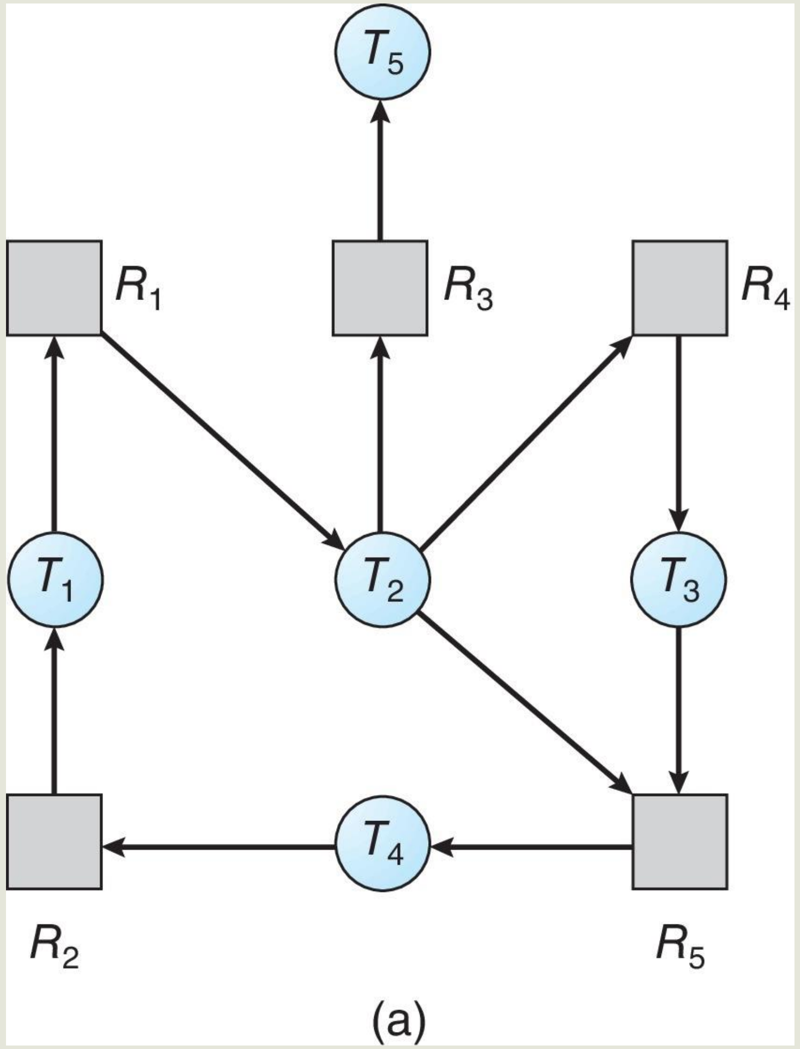
- **Multiple Instances of Resource Types:**
  - When resource types have multiple instances, the detection method is more complex.
  - A cycle in the Resource Allocation Graph is a necessary but not sufficient condition for a deadlock.
  - A cycle indicates that a deadlock might exist, but it is not guaranteed.
  - Therefore, the system must use more advanced algorithms to determine if a deadlock has actually occurred.

Deadlock - Detection – Algorithm

- **Wait-For Graph**
  - It shows only the dependencies between processes.
  - The nodes in the graph represent the processes in the system.
  - A directed edge from process Pi to Pj **(Pi → Pj)** exists if process Pi is waiting for a resource that is currently held by process Pj.
  - A deadlock is present in the system if and only if the wait-for graph contains a cycle.
  - The system must periodically invoke an algorithm that searches for cycles in this graph to detect deadlocks.

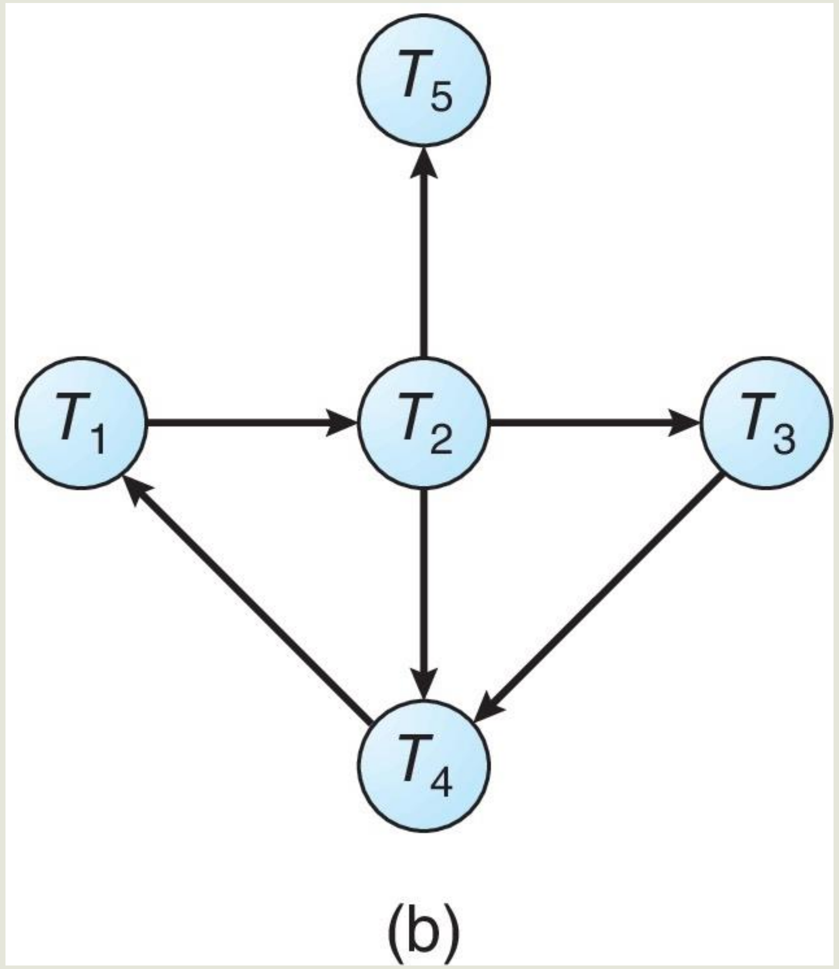
Deadlock - Detection – Algorithm

- **Wait-For Graph**
- Example 1 For the given resource allocation Graph create wait for graph



Deadlock - Detection – Algorithm

- **Wait-For Graph**
- Example 1 wait for graph



Deadlock - Detection – Algorithm

- **Detection Algorithm using Resource Matrices**
  - This algorithm, which is a variant of the Banker's Algorithm, is used for systems with multiple instances of resources.

  - It uses several data structures to analyze the state of the system:
    - **Available:** A vector indicating the number of available resources of each type.
    - **Allocation:** A matrix defining the number of resources of each type currently allocated to each process.
    - **Request:** A matrix indicating the current resource requests of each process.

- The algorithm works by **checking if there is a sequence of processes that can complete their execution with the available resources**.
- If no such sequence exists, the system is in a deadlock state.
- This requires O(m x n^2) operations, where *m* is the number of resource types and *n* is the number of processes.

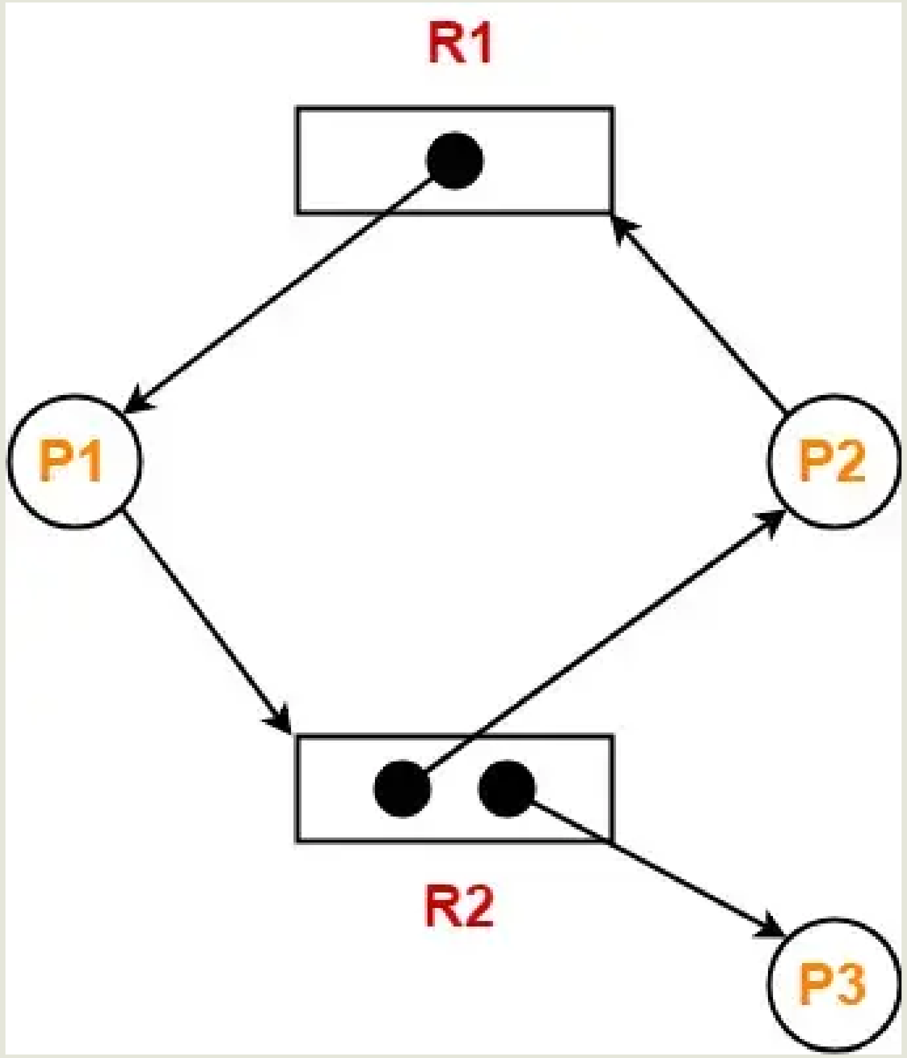
Deadlock - Detection – Algorithm

- **Multi Instance of Each Resource Type**
- Example 1 - Consider the given scenario where,
  - 1 instance of R1 is held by P1
  - P1 is requesting for R2
  - 1 instance of R2 is held by P2
  - P2 is requesting for 1 Instance R1
  - 1 Instance of R2 is held by P3.

Draw the resource allocation graph for the scenario. Identify whether the scenario leads to deadlock otherwise find a safe sequence.

Deadlock - Detection – Algorithm

- Example 1
- The given resource allocation graph is multi-instance with a cycle contained in it.
- The system may or may not be in a deadlock state.

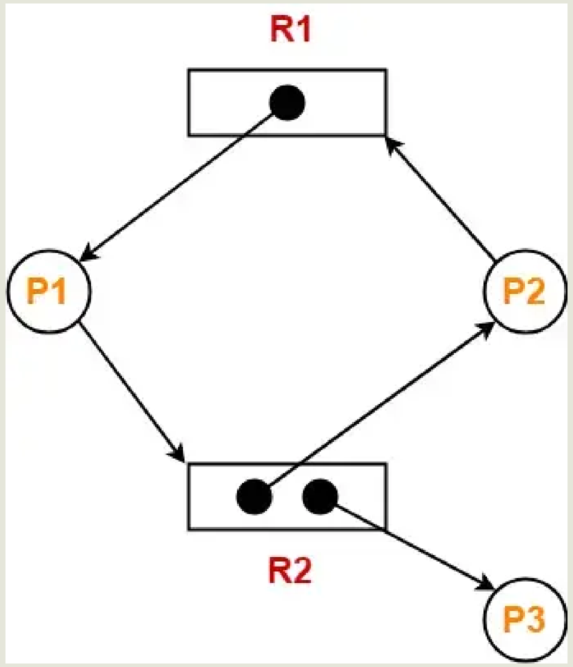


Deadlock - Detection – Algorithm

- Example 1 - **Detection Algorithm using Resource Matrices**
- Construct a table

Process	Resource Allocation		Resource Need	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- **Available Resource – R1=0, R2=0**



Deadlock - Detection – Algorithm

- Example 1 - **Detection Algorithm using Resource Matrices**
- P3 executes and releases its resource then the table changes

Process	Resource Allocation		Resource Need	
	R1	R1	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	0	0	0

- **Available Resource – R1=0, R2=1**

Deadlock - Detection – Algorithm

- Example 1 - **Detection Algorithm using Resource Matrices**
- With available resources only P1 can be satisfied.
- P1 is allocated with R1 resource, it completes execution and releases its resources.

Process	Resource Allocation		Resource Need	
	R1	R2	R1	R2
P1	0	0	0	0
P2	0	1	1	0
P3	0	0	0	0

- **Available Resource – R1=1, R2=1**

Deadlock - Detection – Algorithm

- Example 1 - **Detection Algorithm using Resource Matrices**
- Now with available P2 can be satisfied.
- P2 is allocated with R1 resource, it completes execution and releases its resources.

Process	Resource Allocation		Resource Need	
	R1	R2	R1	R2
P1	0	0	0	0
P2	0	0	0	0
P3	0	0	0	0

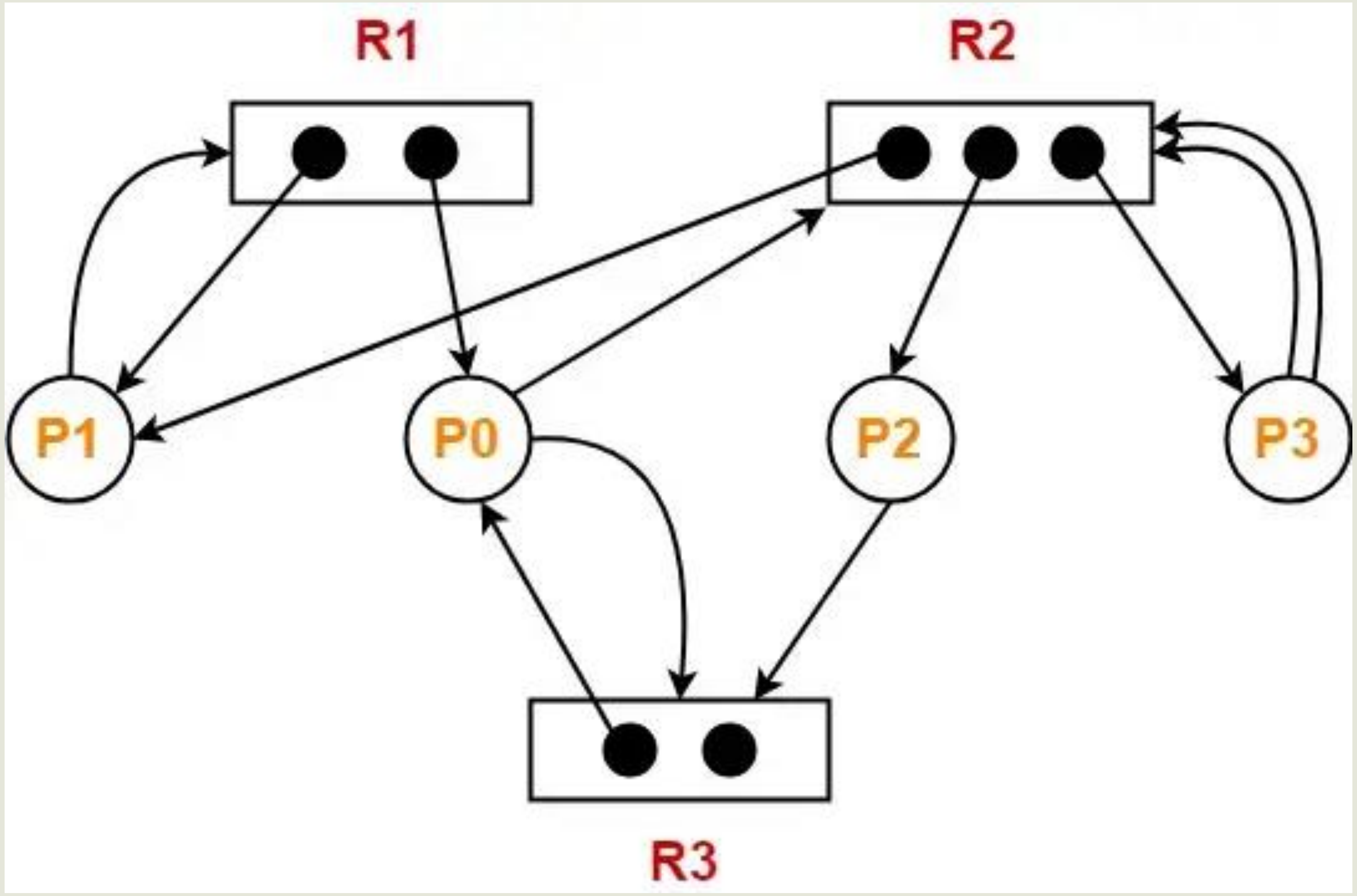
- **Available Resource – R1=1, R2=2, There exists a safe sequence P3, P1, P2 in which all the processes can be executed.**

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Consider the following scenario and identify whether the system in deadlock, otherwise find a safe sequence.
  - There are R1- 2 instances, R2 – 3 instances, R3 – 2 Instances
  - P1 requests R1
  - 1 instance of R1 is allocated to P1
  - 1 instance of R1 is allocated to P0
  - P0 requests R2
  - 1 instance of R2 is allocated to P1
  - 1 instance of R2 is allocated to P2
  - 1 instance of R2 is allocated to P3
  - P3 requests 2 instances of R2
  - 1 instance of R3 is allocated to P0
  - P0 requests 1 instance of R3
  - P2 Requests 1 instance of R3

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**



The given resource allocation graph is multi-instance with a cycle contained in it. So, the system may or may not be in a deadlock state

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Constructing table - Available = 0 0 1

Process	Resource Allocation			Resource Need		
	R1	R2	R3	R1	R2	R3
P0	1	0	1	0	1	1
P1	1	1	0	1	0	0
P2	0	1	0	0	0	1
P3	0	1	0	0	2	0

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Available = 0 0 1 => 0 1 1
- only the requirement of the process P2 can be satisfied. So, process P2 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.

Process	Resource Allocation			Resource Need		
	R1	R2	R3	R1	R2	R3
P0	1	0	1	0	1	1
P1	1	1	0	1	0	0
P2	0	1	0	0	0	1
P3	0	1	0	0	2	0

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Available = 0 1 1 => 1 1 2
- only the requirement of the process P0 can be satisfied. So, process P0 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.

Process	Resource Allocation			Resource Need		
	R1	R2	R3	R1	R2	R3
P0	0	0	0	0	0	0
P1	1	1	0	1	0	0
P2	0	0	0	0	0	0
P3	0	1	0	0	2	0

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Available = 1 1 2 => 2 2 2
- only the requirement of the process P1 can be satisfied. So, process P1 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it

Process	**Resource Allocation**			**Resource Need**		
	R1	R2	R3	R1	R2	R3
P0	0	0	0	0	0	0
P1	0	0	0	0	0	0
P2	0	0	0	0	0	0
P3	0	1	0	0	2	0

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Available = 2 2 2 => 2 3 2
- only the requirement of the process P3 can be satisfied. So, process P3 is allocated the requested resources. It completes its execution and then free up the instances of resources held by it.

Process	**Resource Allocation**			**Resource Need**		
	R1	R2	R3	R1	R2	R3
P0	0	0	0	0	0	0
P1	0	0	0	0	0	0
P2	0	0	0	0	0	0
P3	0	0	0	0	0	0

Deadlock - Detection – Algorithm

- Example 2 - **Detection Algorithm using Resource Matrices**
- Available = 2 2 2 => 2 3 2
- There exists a safe sequence P2, P0, P1, P3 in which all the processes can be executed. So, the system is in a safe state.

Process	**Resource Allocation**			**Resource Need**		
	R1	R2	R3	R1	R2	R3
P0	0	0	0	0	0	0
P1	0	0	0	0	0	0
P2	0	0	0	0	0	0
P3	0	0	0	0	0	0

Deadlock - Avoidance – Algorithm

- Deadlock Avoidance in operating systems is a way to **make sure** that **when resources are given to processes**, the **system never ends up in a situation where processes get stuck waiting for each other** forever.
- Operating system checks every time a process requests a resource to see if giving out that resource might cause a deadlock now or later. If there's even a small chance of deadlock, the OS waits and doesn't give out the resource right away, keeping the system safe.

- **Working Concept:**
  - Each process must tell the OS its maximum resource needs ahead of time.
  - Before granting a resource request, the OS simulates whether, after giving out resources, the system can still let every process finish if it asks for its maximum.
  - If it's always possible for all processes to finish (a safe state), the OS gives the resource. If not, it waits.

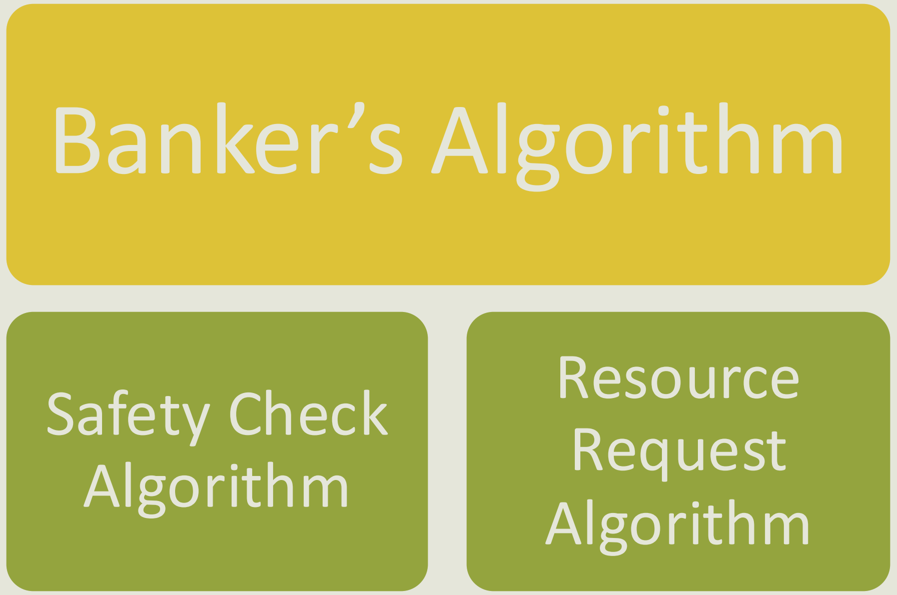
Deadlock - Avoidance – Algorithm

- **SAFE STATE**
  - A safe state in an operating system is a condition where the system can allocate resources to every process (up to their maximum needs) in some order—called a safe sequence—so that all processes can finish without leading to deadlock.

  - The OS checks if resources can be given out in such a way that, for every process, either it can finish now or wait for others to finish and then get its resources
  - There must exist at least one sequence where each process can finish, get its job done, release its resources, and allow other waiting processes to proceed.
  - If no such sequence exists, the system is in an unsafe state, which could lead to deadlock, but not always.

Deadlock - Avoidance – Banker’s Algorithm

- The Banker's Algorithm is a **resource allocation and deadlock avoidance algorithm** in operating systems that ensures the **system always remains in a safe state** when granting resources to multiple processes.
- Banker's Algorithm checks whether it is safe to grant resource requests made by processes, considering both current allocations and future maximum demands.
- The OS acts like a cautious banker who only allocates resources if doing so will never put the system at risk of deadlock.



Deadlock - Avoidance – Banker’s Algorithm

- **Key Data Structures**
  - **Available**: 1D array—each element shows the amount of each resource type currently available.
  - **Max**: 2D array—maximum number of resources each process could ever request.
  - **Allocation**: 2D array—resources currently allocated to each process.
  - **Need**: 2D array—resources each process still needs to finish (Need = Max - Allocation)

Deadlock - Avoidance – Banker’s Algorithm

Steps of the Banker's Algorithm

Safety Check:

Before allocating requested resources, the OS simulates the allocation to see if it can find a sequence (safe sequence) where all processes can finish.

Resource Request Decision:

If granting the request keeps the system in a safe state, the OS grants it.

If not, the OS makes the process wait until it is safe to proceed.

Safe Sequence:

If a possible order exists where every process can complete with available and future-released resources, the allocation is safe.

Deadlock - Avoidance – Banker's Algorithm

- Consider the current system where resources R1, R2 and R3 respectively are consumed by Processes P0 to P4. The total number of Instances of each resource in the system is given as R1=10, R2=5 and R3=7. Apply Banker's algorithm to check whether the system is in safe state and find the safe sequence.

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Deadlock - Avoidance – Banker's Algorithm

- Step 1 - Calculate the current Available resources
  - R1 = 10 – 7(2+3+2) = 3
  - R2 = 5 – 2(1+1) = 3
  - R3 = 7 – 5(2+1+2) = 2

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Deadlock - Avoidance – Banker's Algorithm

- Step 2 - Calculate the **Resource Need** of every Process
  - **Need = Max Resource – Resource Allocation**

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE			NEED RESOURCES		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Deadlock - Avoidance – Banker's Algorithm

- Step 3 - Compare All Available Resources with All Need Resources, if All Need is less than or equal to All Available then we can allocate else check with next process

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE			**NEED RESOURCES**			**Is R1, R2 & R3 can be allocated?**
	R1	R2	R3	R1	R2	R3	R1	R2	R3	Available - 3 3 2
P0	0	1	0	7	5	3	7	4	3	No
P1	2	0	0	3	2	2	1	2	2	Yes
P2	3	0	2	9	0	2	6	0	0	
P3	2	1	1	2	2	2	0	1	1	
P4	0	0	2	4	3	3	4	3	1	

Deadlock - Avoidance – Banker's Algorithm

- Step 4 – Process Sequence Noting:
  - P1 ->

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE			NEED			Allocation Possible	(After Allocation & Process termination) Resource Available Status		
	R1	R2	R3	R1	R2	R3	R1	R2	R3		R1	R2	R3
P0	0	1	0	7	5	3	7	4	3	No			
P1	2	0	0	3	2	2	1	2	2	Yes	5	3	2
P2	3	0	2	9	0	2	6	0	0	No			
P3	2	1	1	2	2	2	0	1	1				
P4	0	0	2	4	3	3	4	3	1				

Deadlock - Avoidance – Banker's Algorithm

- Step 4 – Process Sequence Noting:
  - P1 → P3 →

Process	**RESOURCE ALLOCATION**			**MAXIMUM RESOURCE**		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE			**NEED**			**Allocation Possible**	**(After Allocation & Process termination) Resource Available Status**		
	R1	R2	R3	R1	R2	R3	**R1**	**R2**	**R3**		**R1**	**R2**	**R3**
P0	0	1	0	7	5	3	7	4	3	No			
P1	2	0	0	3	2	2	1	2	2	Yes	**5**	**3**	**2**
P2	3	0	2	9	0	2	6	0	0	No			
P3	2	1	1	2	2	2	0	1	1	Yes	**7**	**4**	**3**
P4	0	0	2	4	3	3	4	3	1				

Deadlock - Avoidance – Banker's Algorithm

- Step 4 – Process Sequence Noting:
  - P1 → P3 → P4 → P0 → P2

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE		
	R1	R2	R3	R1	R2	R3
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

Process	RESOURCE ALLOCATION			MAXIMUM RESOURCE			NEED			Allocation Possible	(After Allocation & Process termination) Resource Available Status		
	R1	R2	R3	R1	R2	R3	R1	R2	R3		R1	R2	R3
P0	0	1	0	7	5	3	7	4	3	Yes	7	5	5
P1	2	0	0	3	2	2	1	2	2	Yes	5	3	2
P2	3	0	2	9	0	2	6	0	0	Yes	10	5	7
P3	2	1	1	2	2	2	0	1	1	Yes	7	4	3
P4	0	0	2	4	3	3	4	3	1	Yes	7	4	5

Deadlock - Avoidance – Banker's Algorithm

Resource Request Algorithm

```
If Request <= Need
   {
          If Request <= Available
          {
                 Allocation = Allocation + request
                 Available = Available – Request
                 Need = Need – Request
          }
   }
```

Safety Check Algorithm

```
Work = Available
Finish[i] = FALSE for i=0 … n-1
Find such i that
Finish[i] = FALSE && Need <= Work
Work = Work + Allocation
Finish[i] = TRUE
If ( Finish[i] = TRUE for all i = 0 … n-1)
Then system is in Safe state.
```

Deadlock - Avoidance – Banker’s Algorithm

Example 2: Consider the process table with number of processes that contains:

- Allocation field (for showing the number of resources of type: R1, R2 and R3 allocated to each process in the table),
- Max field (for showing the maximum number of resources of type: A, B, and C that can be allocated to each process).
- Available resources are: R1=3, R2=2, R1=1
- Find whether the Deadlock can be avoided using banker’s Algorithm?

Process	Allocation			MAX		
	R1	R2	R3	R1	R2	R3
P0	1	1	2	5	4	4
P1	2	1	2	4	3	3
P2	3	0	1	9	1	3
P3	0	2	0	8	6	4
P4	1	1	2	2	2	3

Deadlock - Recovery – Banker’s Algorithm

- **Process Termination**
- Abort all deadlocked threads
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the thread
  - How long has the thread computed, and how much longer to completion
  - Resources that the thread has used
  - Resources that the thread needs to complete
  - How many threads will need to be terminated
  - Is the thread interactive or batch?

Deadlock - Recovery – Banker’s Algorithm

- **Resource Preemption**
  - Selecting a victim – minimize cost
  - Rollback – return to some safe state, restart the thread for that state
  - Starvation – same thread may always be picked as victim, include number of rollback in cost factor

Deadlock - Summary

Deadlock	A situation in OS where two or more processes are permanently waiting for each other to release resources, causing all to be stuck indefinitely.
Deadlock Conditions	Four necessary conditions for deadlock: Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait.
Deadlock Prevention	Strategies ensuring that at least one of the deadlock conditions never holds, thus avoiding deadlock. Common algorithms include Wait-Die, Wound-Wait.
Deadlock Avoidance	The OS dynamically checks resource allocation to avoid unsafe states, ensuring system remains in a safe state. The Banker’s Algorithm is the most well-known method.
Deadlock Detection	Allows deadlocks to occur but detects them using algorithms like Wait-For Graph and takes corrective actions to recover.

Deadlock - Summary

Banker's Algorithm	A deadlock avoidance algorithm that checks the "safe state" by simulating allocation requests, granting them only if it keeps the system safe.
Wait-Die Algorithm	A deadlock prevention method where older processes wait for younger ones, but younger processes requesting older process resources get rolled back (die).
Wound-Wait Algorithm	A deadlock prevention approach where older processes preempt (wound) younger ones holding needed resources; younger ones wait if resource is held by older process.