Operating System

Module-4 Notes

by pyqfort.com

Contents Covered:

- Intro to Deadlock

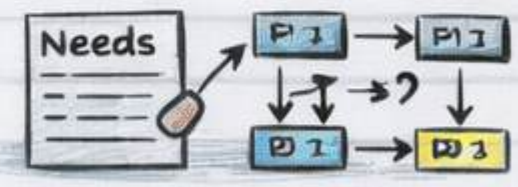
- Conditions for Deadlock

- Pre-emphion Condition
- Preventing Circular Wait

- Deadlock Avoidance
- Banker's Algorithim

- Safety Test Algorithim

- Resource Request-Handling Algorithim



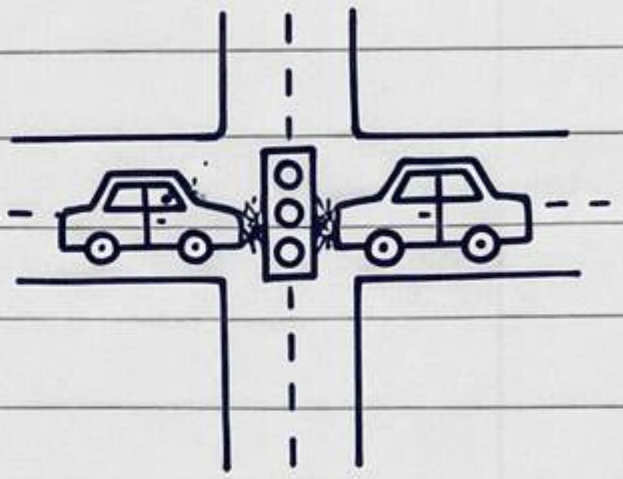
- Deadlock Detection & Recovery

Module 4: Deadlocks

Book Definition: "A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for a resource that another process in the set holds, and none will ever release it (circular wait + no progress)."

Definition:

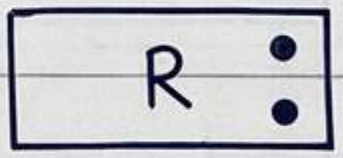
A situation in a system where a set of CONCURRENT PROCESSES request resources in a CONFLICTING manner, leading to an INDEFINITE DELAY in resource allocation.



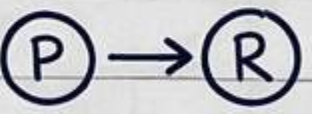
Representation (Resource Allocation Graph):

- Processes are CIRCULAR NODES (e.g., P).

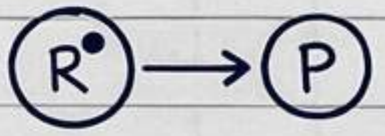
- Resources are RECTANGULAR NODES (e.g., R). Dots inside represent INSTANCES.



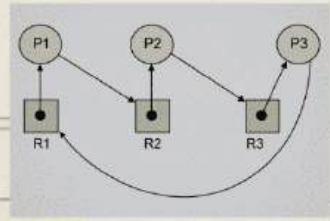
- REQUEST EDGE: Arrow from Process to Resource (process wants resource).



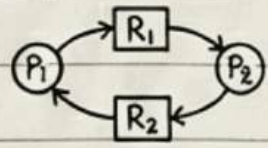
- ASSIGNMENT EDGE: Arrow from Resource to Process (resource allocated to process).



CONDITIONS FOR DEADLOCK

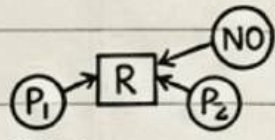


Necessary and Sufficient Condition: A cycle must be present in the Resource Allocation Graph (RAG).

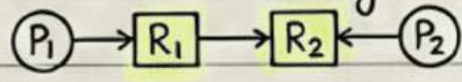


The Four Necessary Conditions (must all hold simultaneously):

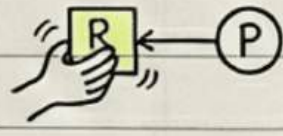
1. Mutual Exclusion: Resources cannot be shared; only one process can use a resource at a time.



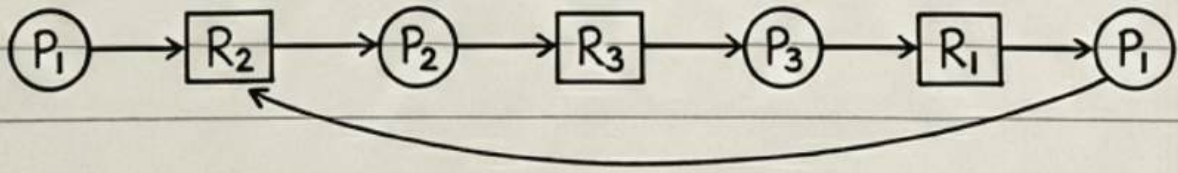
2. Hold and Wait: A process holding at least one resource is waiting for another resource held by some other process.



3. No Preemption: A resource cannot be forcibly taken away; it must be released voluntarily



4. Circular Wait: A set of processes are waiting for each other in a circular chain.



Circular Wait is the most important necessary condition and is a result of the first three.

Breaking the circular wait condition will prevent the deadlock.

DEADLOCK PREVENTION

Philosophy: 'Prevention is better than cure'.
Goal: Make any one of the four necessary conditions false to prevent deadlock.
Advantage: Avoids the cost of detecting and resolving deadlocks.

1. Preventing Mutual Exclusion Condition:
   Impossible to prevent for non-sharable resources, which require mutually-exclusive access.

   Mutual exclusion is an inherent nature of a resource. The idea is to recognize and use sharable resources (e.g., read-only files) as much as possible.

2. Preventing Hold and Wait Condition:

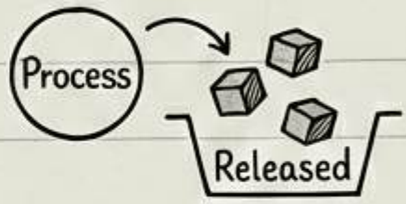
   Two protocols can be used.
   Protocol 1: Request all resources at one time, in advance. No process executes until it gets everything.
   Disadvantages: Process may be idle for a long time. Resource utilization is reduced (e.g., printer held from start but used only at end). May cause starvation.

   Protocol 2: A requesting process should not be holding any resource. To get a new resource, it must first release the resource it is holding.

Preventing No Pre-emption Condition

Necessary to break this condition; deadlock can be prevented by pre-empting the resources.

Method: If a process holding resources wants more that are not free, it must not wait. Instead, it should pre-empt all its resources to avoid deadlock.

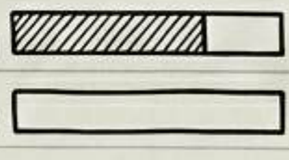


Later, when resources are free, it can request all of them and start execution.

Disadvantage: Pre-empting can cause a process to lose all its work (e.g., pre-empting a printer).

Guidelines for use:

1. Based on execution: If process A is near completion and B just started, pre-empt B's resource for A.



2. Based on priority: Higher priority process can pre-empt resource from a lower priority one

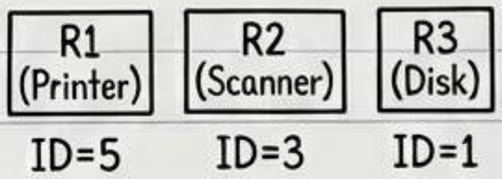
This method is costlier as processes lose work and must restart, so it's used in rare cases only.

PREVENTING CIRCULAR WAIT

Circular wait is a consequence of the other three conditions, so it can be prevented if any of them are.

Resource Ordering / Ranking:

Every resource type is given a unique integer number (ID) as identification ($F: R \rightarrow I$).

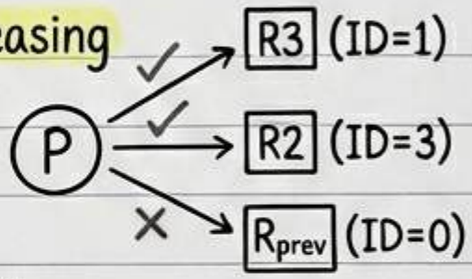


Protocol:

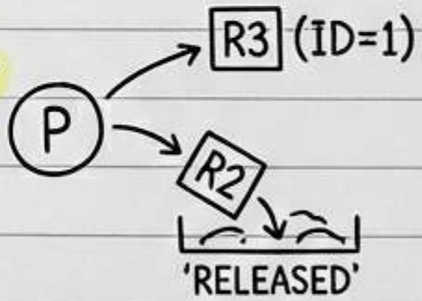
Processes must request resources in an increasing order of their IDs.

If a process holds a resource, it can only request a resource with a greater ID.

This specific ordering will not allow the circular wait condition to arise.



If a process wants a resource with a lower ID than it holds, it must release all its resources first.



Challenges & Guidelines:

- Difficult to implement practically; releasing non-pre-emptable resources can cause a process to lose its work and lead to degraded performance.

- Ordering should be based on use (e.g., printer has higher ID than CD drive if used at the end).

Start ⟶ CD (ID=2) ⟶ ⋯ ⟶ Printer (ID=5) ⟶ End

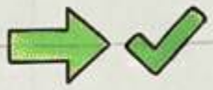
DEADLOCK AVOIDANCE

An alternative to prevention, which can lead to low efficiency and low device utilization.

Mechanism: Checks in advance if a resource request will lead to a deadlock condition.

States:

- Safe state: The system state that does not lead to a deadlock.



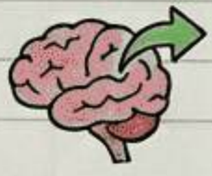
- Unsafe state: The system state that will lead to a deadlock.

Algorithm: An algorithm runs dynamically to check if allocating a resource changes the system to an unsafe state. If it does, the request must wait.

Feasibility: A request is feasible only if total allocated resources do not exceed the total available.

Advantage: Better than prevention as it doesn't constrain resources, avoiding performance degradation.

Requirement: Requires advanced knowledge of maximum demands, available resources, and future requests.



DEADLOCK AVOIDANCE (Single Instance Resources)

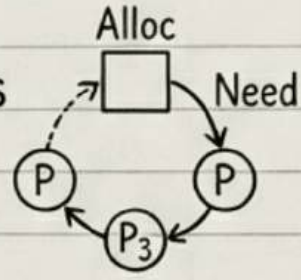
Used when every resource type has a single instance.

Mechanism: Uses the Resource Allocation Graph (RAG) with with a new edge called a claim edge.

Claim Edge: A dotted line from a process to a resource, indicating a potential future request. It's not yet an active request.

Process: When a process requests a resource, a corresponding claim edge is drawn.

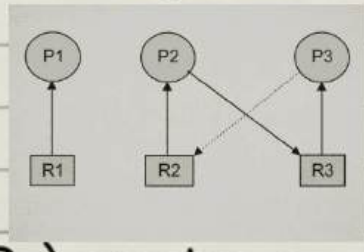
Before granting, the system performs a cycle check.



If no cycle is formed, the claim edge is converted to a request edge (solid line), and the request is granted.

If a cycle is created, the request is rejected to avoid deadlock.

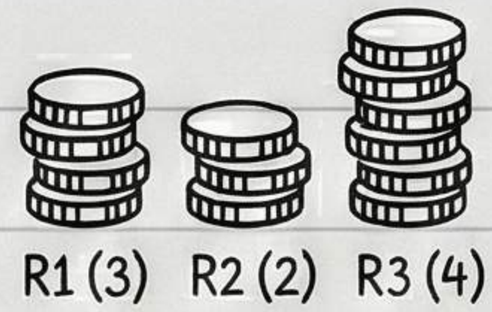
Example: Request from P_3 for R_2.



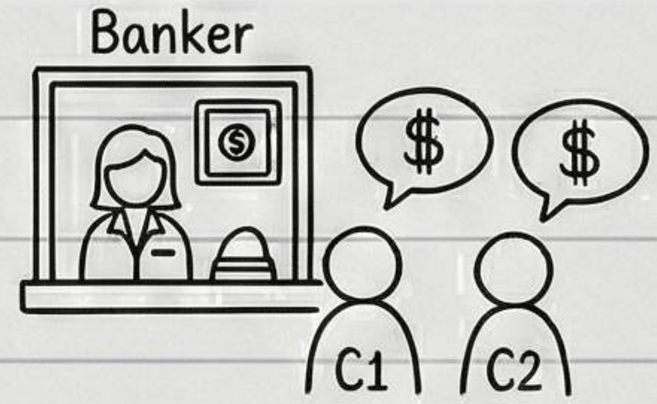
Drawing a claim edge (P_3...>R_2) creates a cycle ($P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$). Thus, the request is denied.

BANKER'S ALGORITHM

Designed for systems with multiple instances of resource types, where RAG cycle check is insufficient.

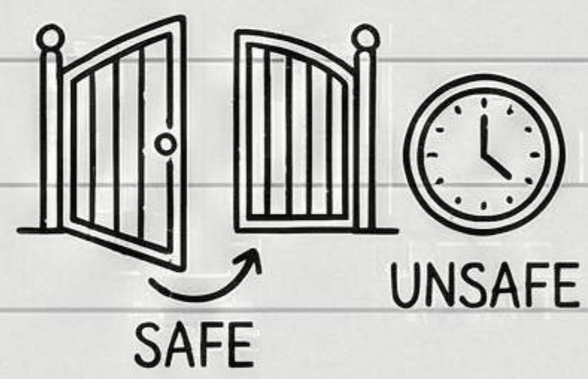


Analogy: Like a banker lending cash to customers. Banker can't lend more than a customer's request or the total available cash.



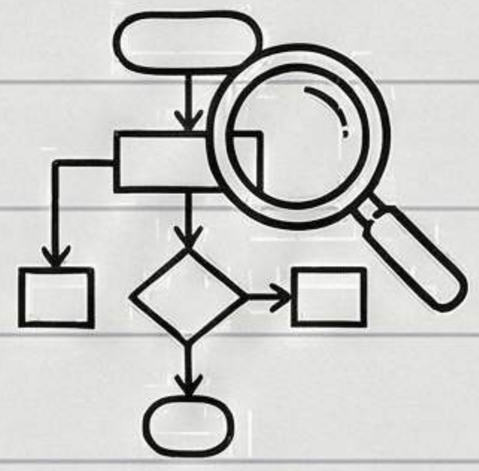
Mechanism: Maintains data structures to check if a resource request maintains a safe state.

If the request leads to a safe state, it is granted; otherwise, the process must wait.



Algorithm has two parts:

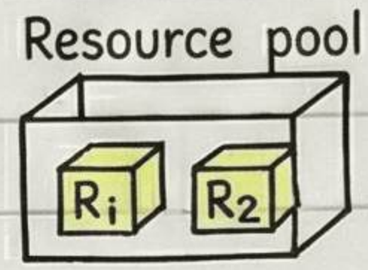
1. Safety Test Algorithm: Checks current state's safety.
2. Resource Request-Handling Algorithm: Checks if a new request affects safety.



This way, the algorithm avoids deadlock by denying unsafe requests.

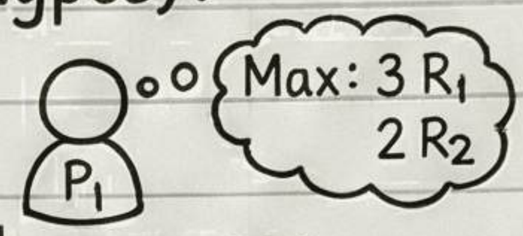
BANKER'S ALGORITHM DATA STRUCTURES

1. **Total Resources (Total_Res) : - vector**



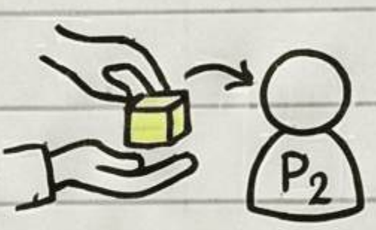
Stores total number of resources in a system.
Let Total_Res[i] = j
j instances of resource type R_i are in the system.
It's a **vector** of length r (number of resource types).

2. **Maximum Demand (Max) : - matrix**



Stores maximum demand of resources for each process.
Let Max[i,j] = k
Process P_i has a total demand of k instances of resource type R_j.
It's a **p × r matrix** (p = number of processes).

3. **Current Allocation (Alloc) : - matrix**



Indicates current allocation status of all resource types.
Process P_i is allocated k instances of resource type R_j.
It's a **p × r matrix.**

4. **Available Resources (Av) : - vector**

Stores current available instances of each resource type.
Let Av[i] = j. j instances of R_i are available.
It's a **vector.**

$$Av[i] = Total\_Res[i] - \sum_{nall} (\text{all processes})\ Alloc[i].$$

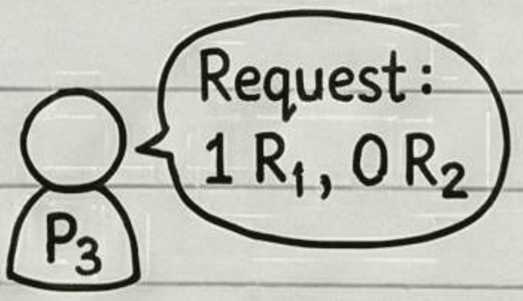
5. **Current Need (Need) : - matrix**

Max - Alloc = Need
□ □□ □□□

Indicates remaining resource need of each process.
Let Need[i,j] = k. Process P_i requires k more instances of R_j.
It's a **p × r matrix.**

$$Need[i,j] = Max[i,j] - Alloc[i,j].$$

6. **Request (Req) : - vector**



Stores the resource request for process P_i.
Let $Req_i[j] = k$. Process P_i has requested k instances of resource type R_j.
It's a **vector.**

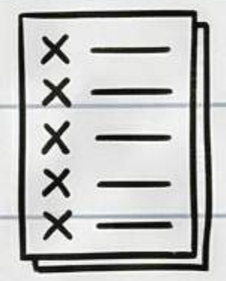
SAFETY TEST ALGORITHM

Used to check the current state's safety.

Initialization:
Let Current_Avail and Marked be two vectors of length n and p, respectively. Safe String is an array to store process IDs.

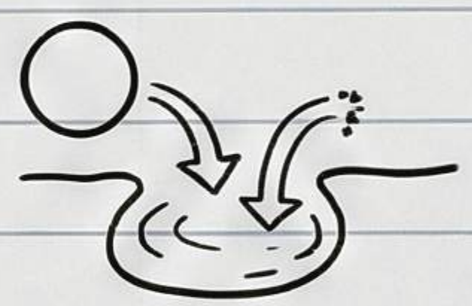
The algorithm:

1. Current_Avail = Av
2. Initialize Marked[i] = false for all i = 1 to p.



3. Find a process P_i such that:
   $Need_i \leq Current\_Avail$ and Marked[i] = false

4. If (found) {
   Current_Avail = Current_Avail + $Alloc_i$
   Marked[i] = true
   Save process number in SafeString[]
   go to step 3.
   }



5. If (Marked[i] == true) for all processes, then the system is in safe state.
   Print SafeString.

P1, P2, P3...

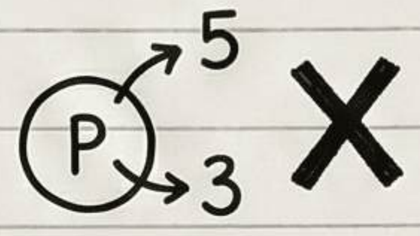
6. Otherwise, the system is not in safe state, and is in deadlock.

Resource Request-Handling Algorithm

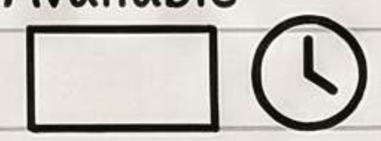
Let Req_i be the vector for resource request of process P_i.

The algorithm checks if a request can be granted safely:

1. If ($Req_i > Need_i$): The request is not feasible (asking for more than needed) and is rejected.



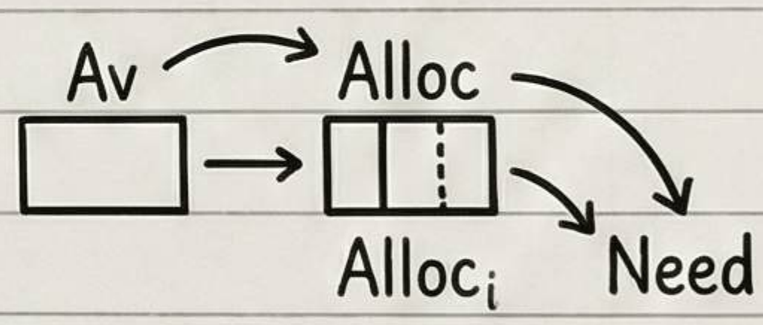
2. elseif ($Req_i > Av$): Resources are not available, so the process must wait.



3. Otherwise (Request is valid and available):

   a. Provisionally allocate resources and update state:

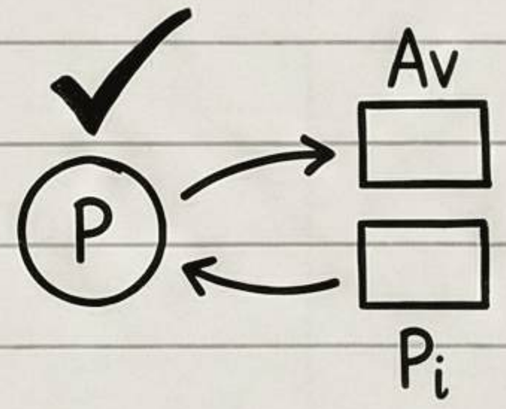
$$Av = Av - Req_i$$
$$Alloc_i = Alloc_i + Req_i$$
$$Need_i = Need_i - Req_i$$



   b. Execute Safety Test Algorithm assuming state change.

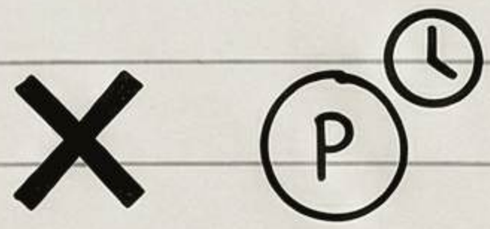
   c. if (state is safe):

      Change the state in actual and allocate the resources.



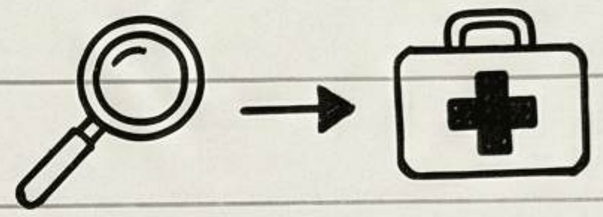
   d. Otherwise (unsafe):

      Keep the state unchanged and do not allocate. Process waits.



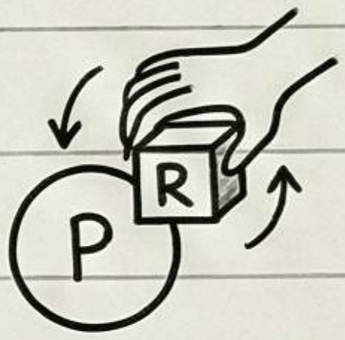
DEADLOCK DETECTION AND RECOVERY

This is the third approach to handle deadlocks. The goal is to detect the deadlock at an appropriate time and then recover from it. The detection algorithm is beneficial only if the system is able to recover. If there is no recovery, the system cannot proceed and is useless.

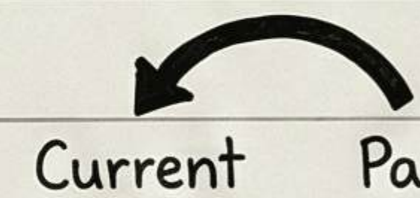


Recovery Methods:

- Resource pre-emption: Forcibly taking a resource from a process.



- Rollback: Reverting a process to a previous safe state and restarting.
- Abortion of the process: Ending one or more processes involved in the deadlock.



The information acquired from the detection algorithm helps in recovering from the deadlock.

Note: The last method is to just ignore the deadlock (often called the Ostrich Algorithm).