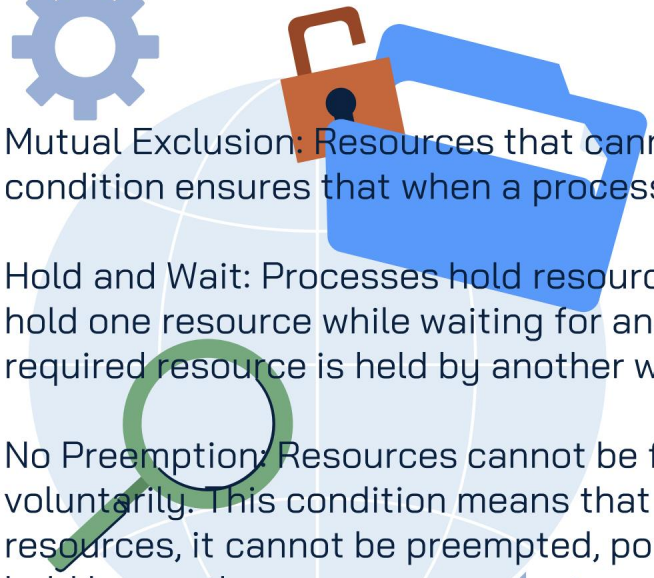
Deadlock Avoidance

By:
Aman Raj

Introduction

Deadlock occurs in a multitasking environment, typically in operating systems, where multiple processes or threads are competing for resources. These resources could be anything from memory segments to input/output devices or even software components. When these processes are unable to proceed because each is waiting for a resource that's held by another, they enter a state of deadlock.

Cause!



Mutual Exclusion: Resources that cannot be used by more than one process at a time. This condition ensures that when a process is using a resource, it has exclusive access to it.

Hold and Wait: Processes hold resources while waiting for others. This means that a process may hold one resource while waiting for another, causing resource wastage and potential deadlock if the required resource is held by another waiting process.

No Preemption: Resources cannot be forcibly taken from a process; they must be released voluntarily. This condition means that if a process is holding a resource and requires additional resources, it cannot be preempted, potentially leading to a deadlock if it's waiting for a resource held by another process.

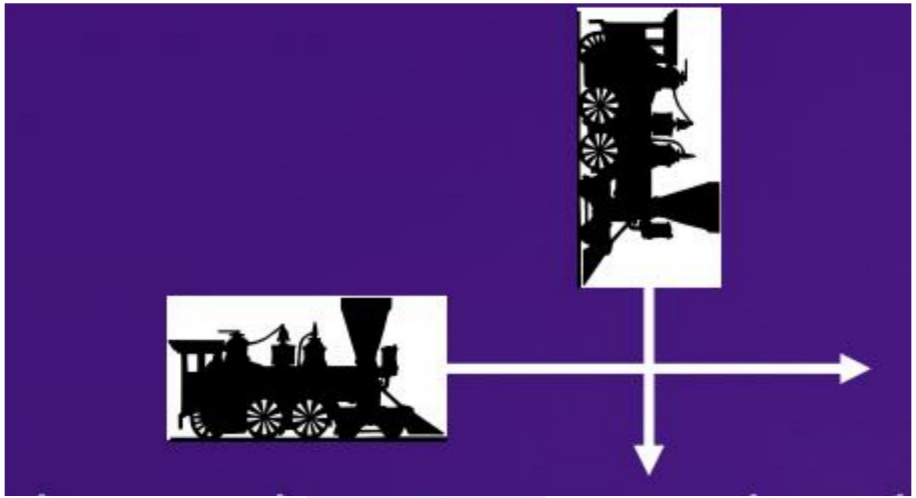
Circular Wait: A circular chain of dependencies exists among processes, where each process holds at least one resource that is requested by the next process in the chain. This condition leads to a deadlock when each process in the chain is waiting for a resource held by another process in the chain, creating a circular dependency.

01

Real-Life Examples

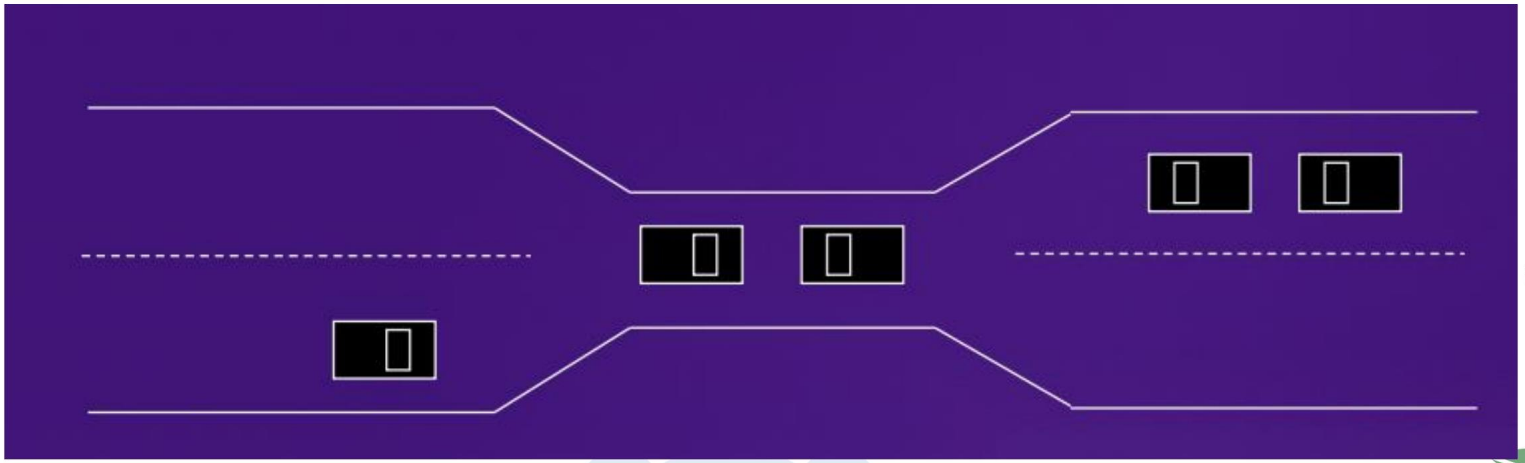
Simple Scenarios

Deadlock in Trains: When two trains are traveling on different tracks and encounter a crossing, they must hold or wait for some time. This situation is analogous to a deadlock.



Simple Scenarios

Imagine two people needing to cross a narrow bridge, each waiting for the other to move first.



Key aspects to keep in mind

- A deadlock occurs when a set of processes are permanently blocked, waiting for resources held by each other.
- Deadlocks can significantly impact the performance and stability of operating systems by preventing processes from completing their tasks.

CONDITIONS

How can we identify that it is a deadlock?

Must not Happen

Mutual exclusion

only one process at a time can use a resource.

Hold and wait

a process holding resource(s) is waiting to acquire additional resources held by other processes.

No preemption

a resource can be released only voluntarily by the process holding it upon its task completion.

Circular wait

there exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by
P1, P1 is waiting for a resource that is held by P2, …, Pn−1 is waiting for a resource that is held by Pn, and Pn is waiting for a
resource that is held by P0.

Deadlock Prevention

Restrain one of the following four conditions:

Mutual exclusion

Not required for sharable resources. (but not work always.)

No preemption

If a process P1 requests a resource R1 that is allocated to some other process P2 waiting for additional resource R2, R1 is allocated to P1.

Hold and wait

Must guarantee that whenever a process requests a resource, it does not hold any other resources.

Circular wait

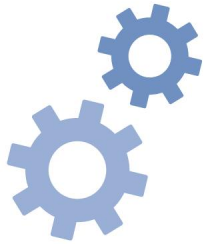
Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Safe State

A system is in a safe state only if there exists a only if there exists a safe sequence.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence of ALL the processes in the systems such that for each Pi , the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj , with $j < i$
- That is:
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pi terminates, Pi +1 can obtain its needed resources, and so on.



Facts

- If a system is in safe state ⇒ *no deadlocks*
- If a system is in unsafe state ⇒ *possibility of deadlock*
- Avoidance ⇒ ensure that a system will *never enter an unsafe state*.

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

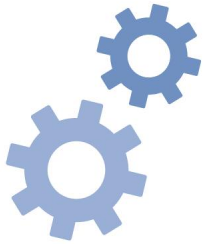
Data Structures for the Banker's Algorithm

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type Rj available
- **Max**: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
- **Allocation**: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
- **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Working of Bankers Algorithm

- 1. Initialization: The system must know the total number of resources of each type and how many of each type are currently available. This information is used to initialize the system.
- 2. Request Handling: When a process requests resources, the system checks if the request can be granted immediately without leading to a deadlock. If the request can be satisfied safely, the resources are allocated to the process; otherwise, the process must wait until sufficient resources are available.
- 3. Safety Algorithm: The Banker's algorithm employs a safety algorithm to determine if the system is in a safe state before granting a resource request. The safety algorithm simulates the allocation of resources to processes in a specific sequence and checks if each process can finish executing without getting stuck in a deadlock. If all processes can complete, the system is in a safe state, and the resource request can be granted.
- 4. Resource Allocation: If the safety algorithm determines that granting a resource request will not lead to a deadlock, the requested resources are allocated to the process, and the process can proceed with its execution.
- 5. Release of Resources: When a process finishes executing, it releases the resources it was allocated back to the system. These released resources become available for allocation to other processes.



Code Explanation

Applications

1. Operating Systems Education: This code could be used in an educational setting to demonstrate how deadlock detection and avoidance algorithms work in operating systems.

2. System Monitoring Tools: It can be extended into a system monitoring tool where administrators can visualize resource allocation and usage in real-time, potentially helping in proactive resource management.

3. Resource Allocation Optimization: By analyzing historical data collected by this system, organizations can optimize their resource allocation strategies to minimize the occurrence of deadlocks and maximize system efficiency.

4. Process Management in Cloud Environments: This code could be adapted for managing processes in cloud environments where multiple virtual machines or containers compete for resources.

5. Database Management Systems: It can be used as a basis for implementing deadlock detection and prevention mechanisms in database management systems to ensure data integrity and system stability.

Conclusion

Deadlock avoidance is a critical aspect of system design and management, particularly in environments with concurrent processes competing for resources. Throughout this presentation, we've explored various strategies and techniques aimed at preventing deadlocks and ensuring the smooth operation of systems. Here are the key takeaways:

1. Understanding Deadlock: Deadlock occurs when two or more processes are indefinitely blocked, each waiting for a resource held by the other, leading to a state of paralysis in the system.

2. Deadlock Avoidance Strategies: We've explored several strategies for preventing deadlocks, such as resource allocation graphs, Banker's algorithm, and various heuristic approaches. These methods aim to allocate resources in a way that ensures deadlock-free execution.

3. Banker's Algorithm: We've delved into the Banker's algorithm as a proactive approach to deadlock avoidance. By carefully managing resource allocation and employing a safety algorithm, the Banker's algorithm ensures that processes can be executed without encountering deadlocks.

4. Implementation and Applications: The provided code exemplifies how deadlock avoidance techniques can be implemented in real-world scenarios using programming languages like Python and integrated with databases and graphical user interfaces. The applications of such implementations range from educational tools to system monitoring and optimization in diverse domains.

5. Continuous Improvement: Deadlock avoidance is an ongoing process that requires continuous monitoring, analysis, and refinement of strategies as systems evolve and requirements change. By staying vigilant and proactive, organizations can effectively mitigate the risks associated with deadlocks and ensure the reliability and efficiency of their systems.

THANK YOU

Made By :
Aman Raj