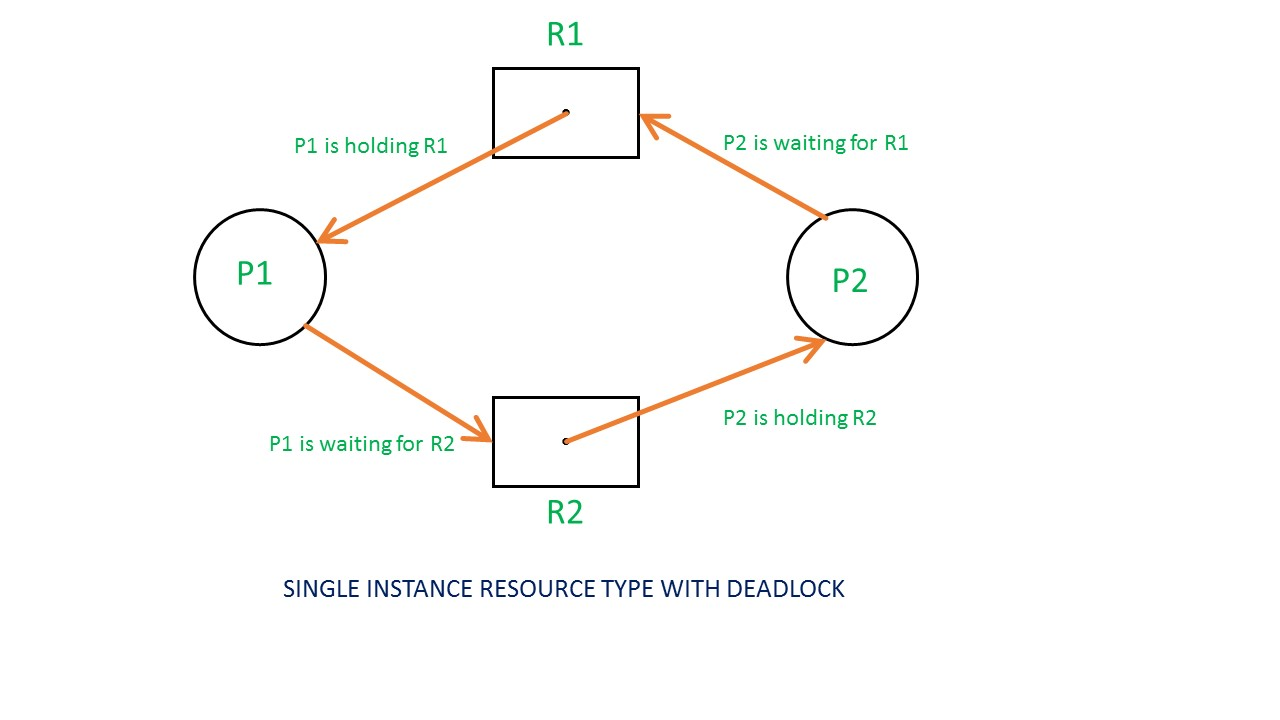
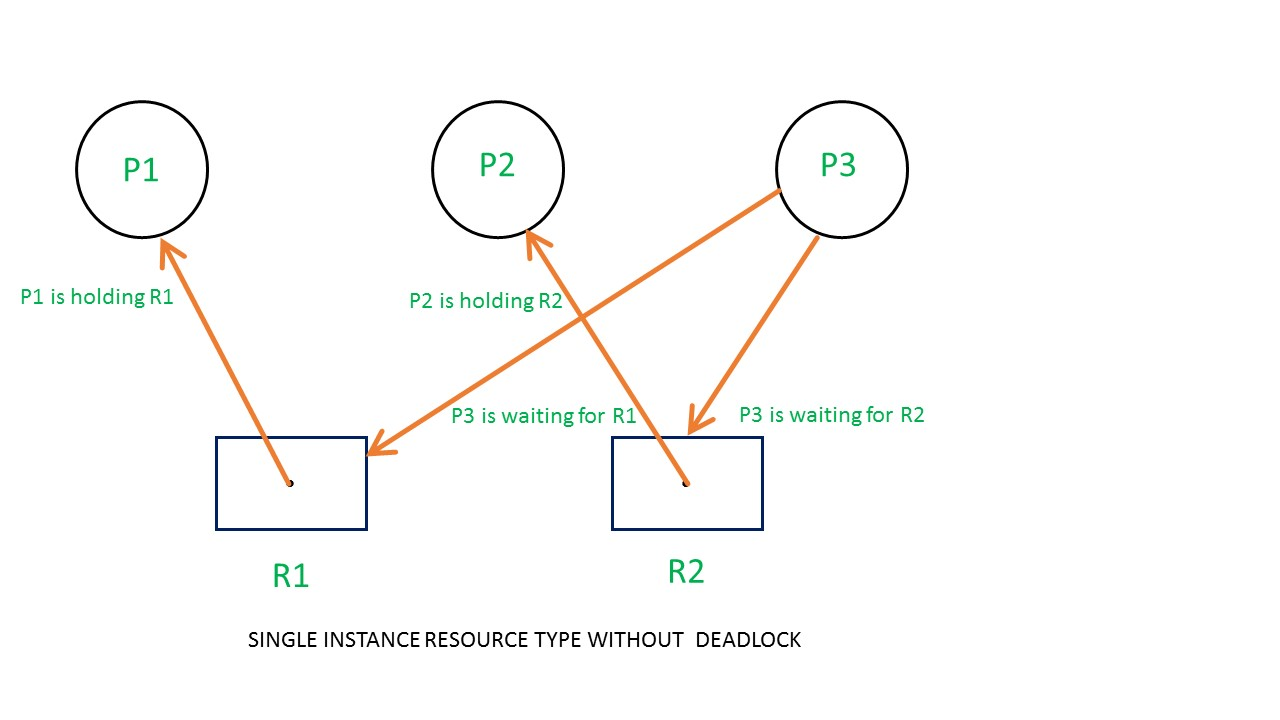
1. **Slow Learner Activities**

Activity 1 (Single Instances RAG)



If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

**Activity 2:**



Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

**Activity 3: Traffic Gridlock at an Intersection**

**Story:**

Imagine a four-way intersection without traffic lights. Cars from all four directions enter the intersection at the same time, and each one blocks the next car's way. No car can move forward or reverse, and the entire intersection is stuck.

Identifying Deadlock Conditions:

1. Mutual Exclusion: Each car needs exclusive access to a section of the intersection.
2. Hold and Wait: Each car holds part of the intersection and waits for the next section to clear.
3. No Preemption: No car can be forced to back out or give up its position.
4. Circular Wait: Car A waits for Car B’s space, Car B for Car C’s, and so on — forming a circle.

**Activity 4: Office Printers**

**Story:**

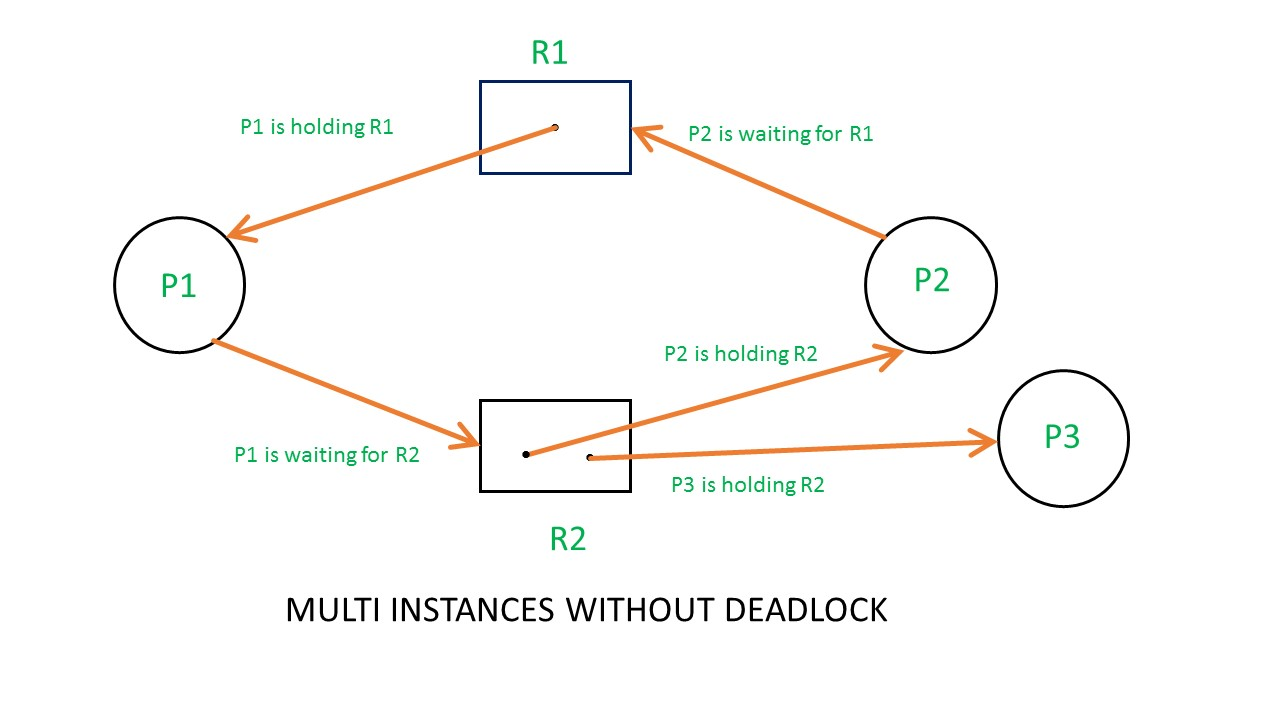
Two employees, Alice and Bob, are printing and scanning documents. Alice is using the printer and needs the scanner, while Bob is using the scanner and needs the printer. Both are waiting for the other to release the resource.

Identifying Deadlock Conditions:

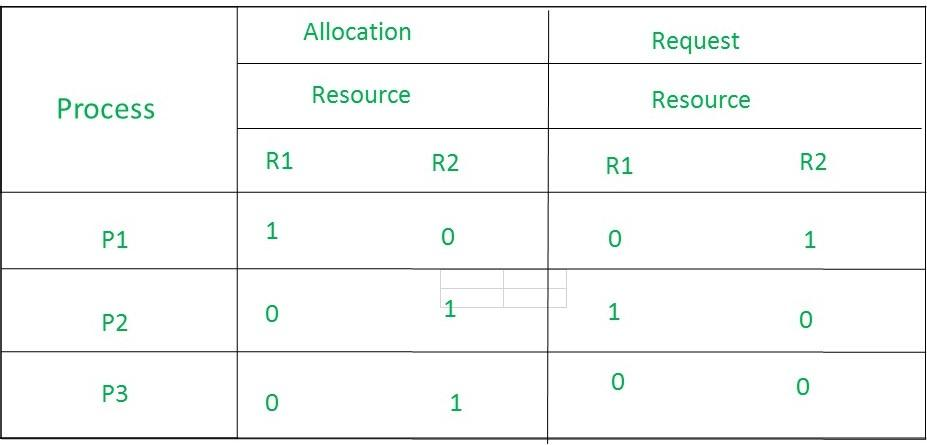
1. Mutual Exclusion: Printer and scanner can only be used by one person at a time.
2. Hold and Wait: Alice holds the printer and waits for the scanner; Bob holds the scanner and waits for the printer.
3. No Preemption: Neither Alice nor Bob can be forced to release their device.
4. Circular Wait: Alice → scanner, Bob → printer — creating a loop of dependencies.

**2. Moderate Learner**

1. **Activity 1**

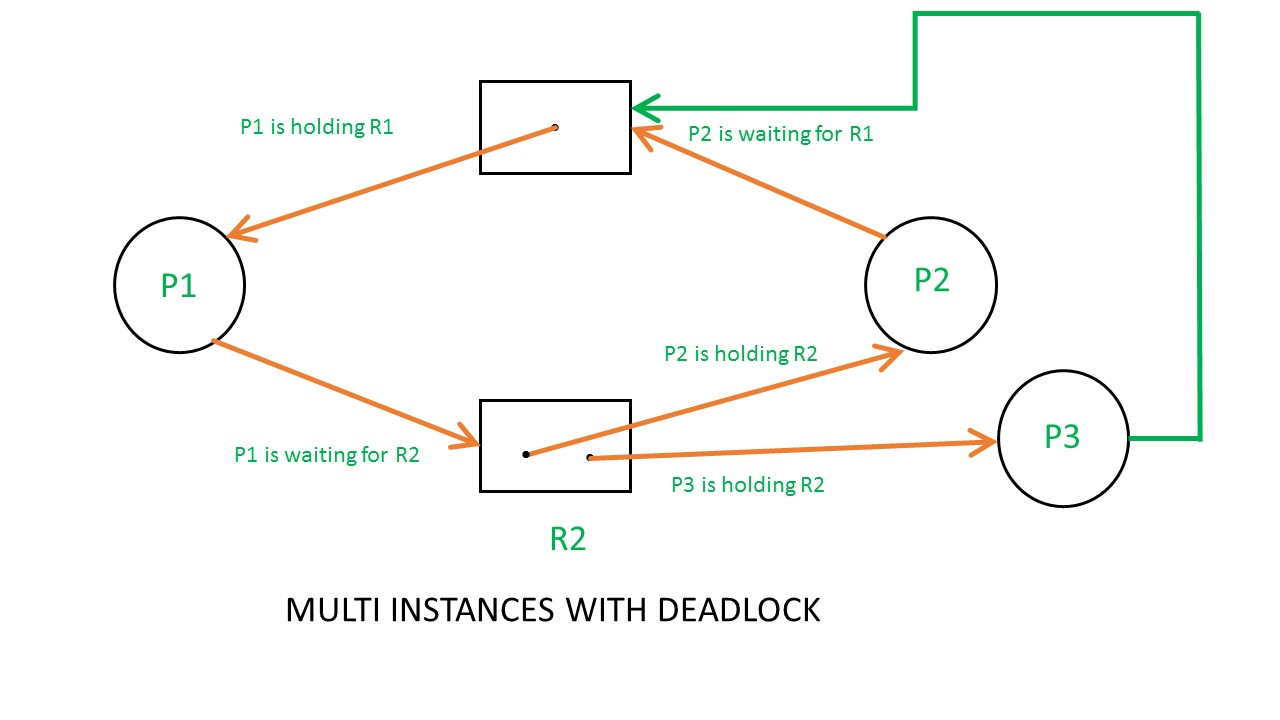


From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

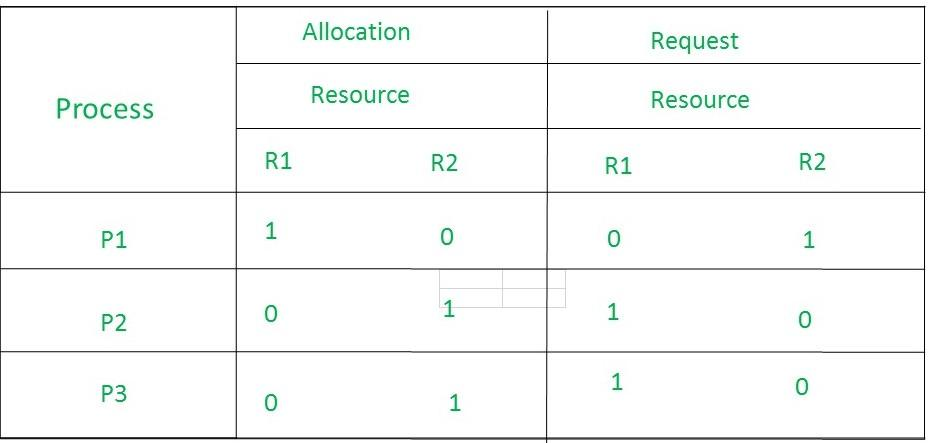


* The total number of processes are three: P1, P2 & P3 and the total number of resources are two: R1 & R2.
* Allocation matrix  
  For constructing the allocation matrix, just go to the resources and see to which process it is allocated.  
  R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.
* Request matrix  
  In order to find out the request matrix, you have to go to the process and see the outgoing edges.  
  P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.  
  So now available resource is = (0, 0).
* Checking deadlock (safe or not)

1. **Activity 2**



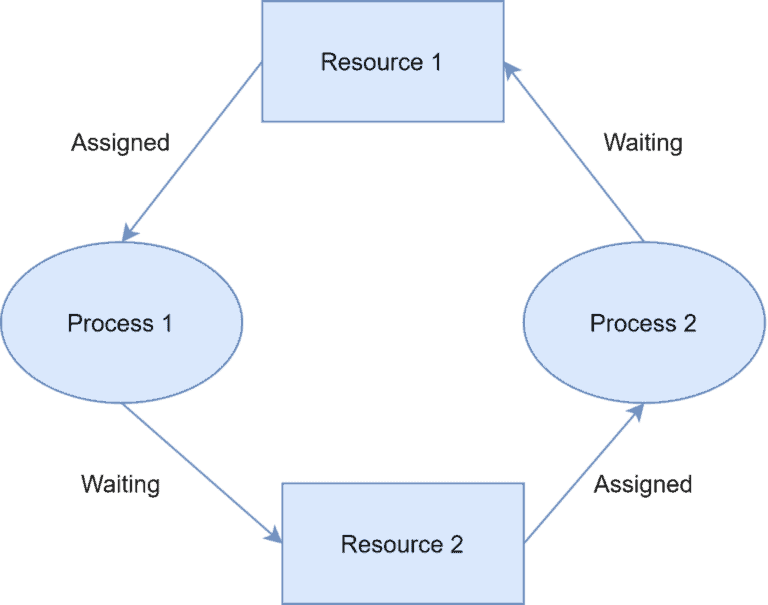
Above example is the same as the previous example except that, the process P3 requesting for resource R1. So the table becomes as shown in below.



So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0). So you can't fulfill any one requirement. Therefore, it is in deadlock. Therefore every cycle in a multi-instance resource type graph is not a deadlock. If there has to be a deadlock, there has to be a cycle. So in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock but not sufficient.

1. **Activity 3**

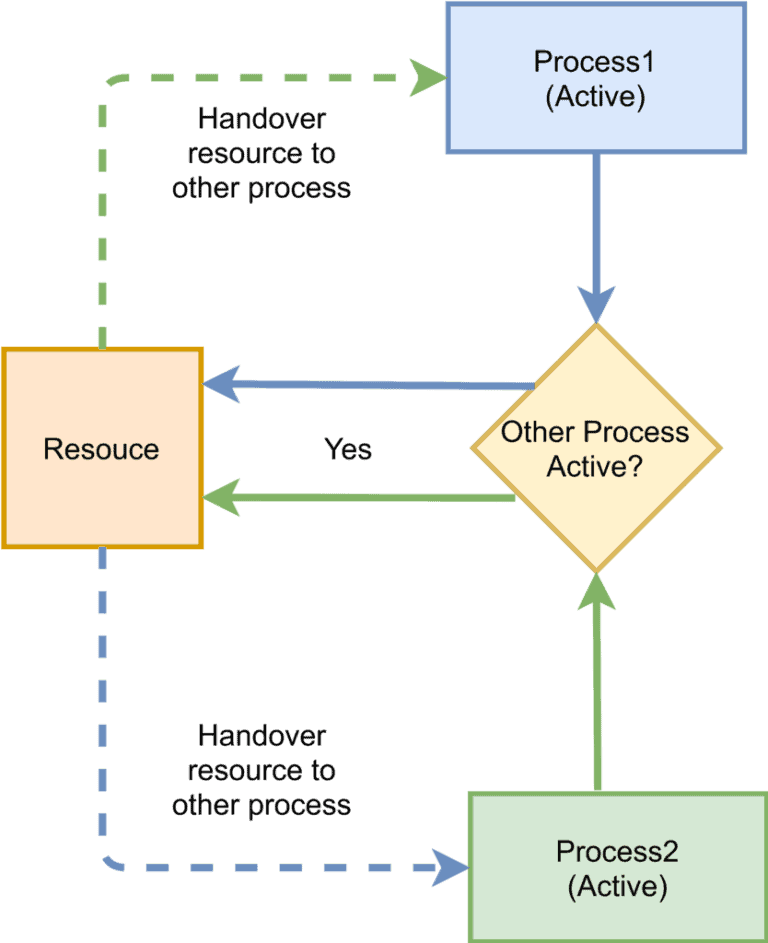
A deadlock is a situation in which processes block each other due to resource acquisition and none of the processes makes any progress as they wait for the resource held by the other process.



The above figure shows the deadlock scenario between process 1 and process 2. Both processes are holding one resource and waiting for the other resource held by the other process. This is a deadlock situation as neither process 1 or process 2 can make progress until one of the processes gives up its resource.

3.1. What Is Livelock?

In the case of a livelock, the states of the processes involved in a live lock scenario constantly change. On the other hand, the processes still depend on each other and can never finish their tasks.

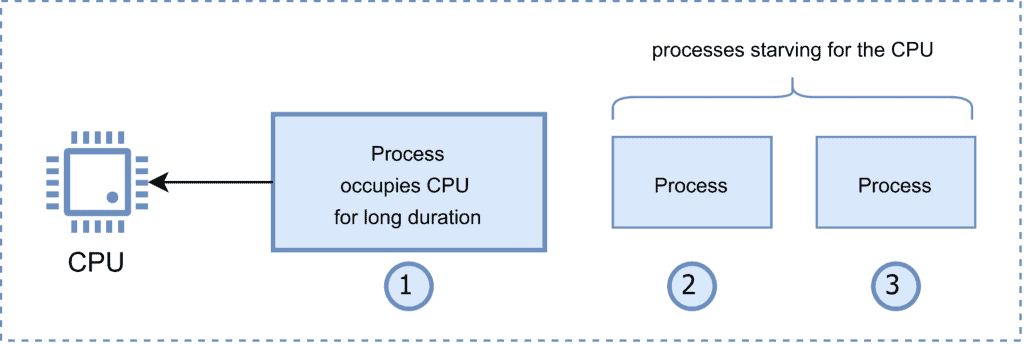


The above figure shows an example of livelock. Both “process 1” and “process 2” need a common resource. Each process checks whether the other process is in an active state. If so, then it hands over the resource to the other process. However as both, the process is inactive status, both kept on handing over the resource to each other indefinitely.

A real-world example of livelock occurs when two people make a telephone call to each other and both find the line is busy. Both gentlemen decide to hang up and attempt to call after the same time interval. Thus, in the next retry too, they ended up in the same situation. This is an example of a live lock as this can go on forever.

What Is Starvation?

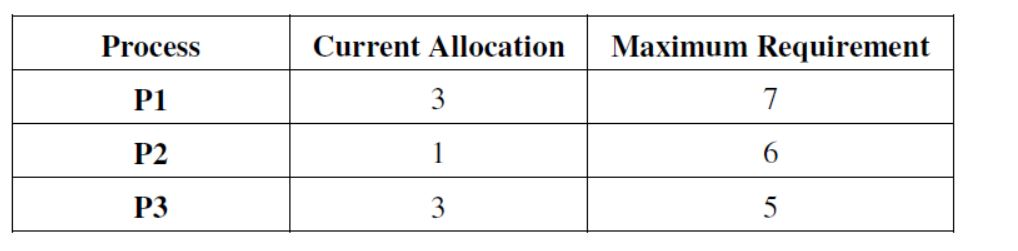
Starvation is an outcome of a process that is unable to gain regular access to the shared resources it requires to complete a task and thus, unable to make any progress.



The above figure shows an example of starvation of “process 2” and “process 3” for the CPU as “process 1” is occupying it for a long duration.

1. Activity 4

A system shares 9 tape drives. The current allocation and maximum requirement of tape drives for three processes are shown below:

best describe current state of the system ?

Require P1 =7-3 =4

P2 =6-1 =5

P3 =5-3 =2

Available 9-(3+1+3) =2

P3->P2->P1

OR

P3->P1->P2

Hence Safe and NO Deadlocked

**3. Fast Learner Activity**

1. Activity 1

**1. Oracle Database Deadlock (2019)**

**Scenario**: A financial institution experienced recurring deadlocks in their Oracle 12c database during end-of-day batch processing.

**Root Cause**:

* Multiple batch jobs updating customer accounts in different orders
* Job A: Locked account X, then attempted to lock account Y
* Job B: Locked account Y, then attempted to lock account X
* Circular wait condition created

**Resolution**:

* Implemented consistent ordering for account updates (always process accounts in ID order)
* Added deadlock detection and retry logic in application code
* Reduced transaction isolation level where appropriate

**2. Linux Kernel Deadlock (2016, Kernel 4.4)**

**Scenario**: Users reported system hangs during certain filesystem operations.

**Technical Details**:

* Process A held inode lock, waiting for journal transaction
* Process B (journal thread) held transaction lock, waiting for inode lock
* Both in uninterruptible sleep (D state)

**Resolution**:

* Kernel patch to restructure locking hierarchy
* Added lockdep (lock dependency checker) instrumentation
* Improved lock ordering in the ext4 filesystem driver

**3. SQL Server Deadlock in E-Commerce (2020)**

**Scenario**: High-traffic e-commerce site experienced checkout failures during peak periods.

**Analysis**:

* Deadlocks between inventory updates and order creation
* 60% involved three or more processes in complex wait chains
* Schema design encouraged lock contention on product records

**Solutions Implemented**:

* Optimized queries to reduce lock durations
* Implemented application-level queuing for high-demand products
* Added NOLOCK hints for non-critical reports
* Introduced deadlock victim prioritization

**4. Activity 4**

Common Deadlock Patterns

1. **Resource Deadlocks**:
   * Processes competing for the same set of resources in different orders
   * Common in database transactions
2. **Communication Deadlocks**:
   * Processes waiting for messages from each other
   * Frequent in distributed systems
3. **Livelocks**:
   * Processes keep changing state but make no progress
   * Often seen in poorly implemented retry logic

Prevention and Mitigation Strategies

1. **Database-Specific**:
   * Consistent access order for tables/rows
   * Appropriate transaction isolation levels
   * Optimistic concurrency control where possible
   * Query timeouts and deadlock victim selection
2. **Operating System**:
   * Proper lock ordering in kernel code
   * Lock hierarchy validation tools (like lockdep)
   * Preemption-safe locking mechanisms
3. **Application-Level**:
   * Exponential backoff for retries
   * Circuit breakers for persistent deadlocks
   * Distributed lock services (e.g., ZooKeeper)

**Lessons Learned**

1. Deadlocks often surface only under specific load conditions or timing
2. Prevention is more effective than detection/recovery in production systems
3. Proper instrumentation (logging, monitoring) is crucial for diagnosis
4. Many deadlocks stem from architectural flaws rather than implementation bugs