//Experiment : 5

// Aim : Implement token ring based mutual exclusion algorithm.

theory:

In distributed systems, mutual exclusion and critical section are concepts related to concurrent access and synchronization of shared resources among multiple processes or nodes. Here's an explanation of these terms:

Mutual Exclusion:

1. Mutual exclusion refers to the property that ensures that only one process or node can access a shared resource or enter a critical section at a time. It prevents simultaneous access to the shared resource, which could lead to race conditions or data inconsistencies. Mutual exclusion guarantees that conflicting operations on shared resources do not occur concurrently, thereby preserving data integrity and correctness.

Critical Section:

1. A critical section is a specific portion of a program or code segment that accesses shared resources. It contains operations or instructions that need to be executed atomically or in an exclusive manner. While a process is executing within the critical section, it must be ensured that no other process can enter the same critical section concurrently to avoid conflicts and ensure correct execution of the program.

To enforce mutual exclusion and manage critical sections in a distributed system, various synchronization mechanisms and algorithms can be used, such as:

* Locks and Semaphores: These are traditional synchronization primitives that can be used to protect critical sections by acquiring and releasing locks or counting semaphores.
* Distributed Locks: Distributed lock algorithms enable multiple processes or nodes to acquire and release locks across a distributed environment. Examples include the Ricart-Agrawala algorithm, Maekawa's algorithm, and the Chandy-Lamport algorithm.
* Atomic Transactions: Transactional mechanisms provide a way to ensure atomicity, consistency, isolation, and durability (ACID properties) when accessing shared resources. Distributed transactions allow multiple processes or nodes to coordinate and execute transactions in a distributed manner.
* Distributed Mutual Exclusion Algorithms: Various distributed mutual exclusion algorithms, such as the Ricart-Agrawala algorithm, Lamport's bakery algorithm, and the token-based algorithms, ensure that processes or nodes take turns accessing shared resources.

The goal of these synchronization mechanisms and algorithms is to coordinate access to shared resources in a distributed system, avoiding conflicts and ensuring the correct execution of critical sections while maintaining data integrity and consistency. By enforcing mutual exclusion and managing critical sections effectively, distributed systems can achieve proper synchronization and avoid race conditions, deadlocks, and other concurrency-related issues.

what is mutual exclusion

Mutual exclusion refers to a mechanism or technique used in concurrent programming to ensure that only one thread or process can access a shared resource or critical section at a time. It aims to prevent conflicts and maintain data integrity when multiple threads or processes concurrently access shared resources.

The concept of mutual exclusion is based on the idea that certain sections of code or critical resources should not be accessed simultaneously by multiple threads. This is because concurrent access can lead to race conditions, data corruption, and inconsistent results.

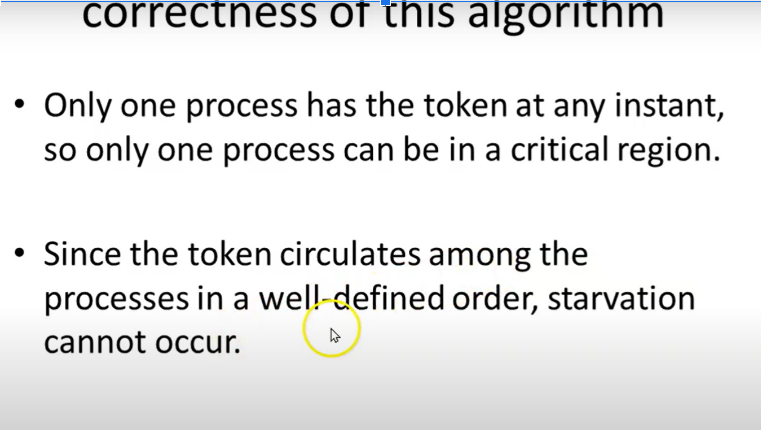
To achieve mutual exclusion, synchronization techniques such as locks, semaphores, or mutexes are used. These mechanisms allow threads or processes to acquire a lock or semaphore before accessing the shared resource. When a thread holds the lock, other threads requesting access to the same resource are blocked or put in a waiting state until the lock is released. Once the thread completes its work on the shared resource, it releases the lock, allowing other threads to acquire it and access the resource.

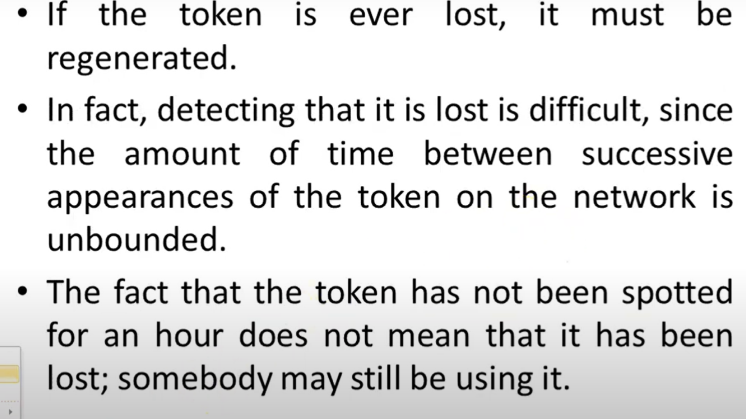
By enforcing mutual exclusion, synchronization mechanisms ensure that only one thread can access the critical section at a time, preventing data races and maintaining the integrity of shared data. This helps in avoiding conflicts and ensuring the correctness of concurrent programs.

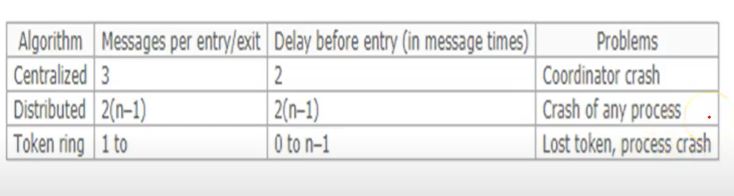
Critical Section :

In distributed systems, a critical section refers to a part of a program or code that accesses shared resources or variables, and where concurrent access by multiple processes or nodes can lead to unexpected or incorrect results. The critical section needs to be executed in a mutually exclusive manner to maintain consistency and avoid race conditions.

In a distributed system, where multiple processes or nodes are running concurrently and communicating over a network, ensuring mutual exclusion in critical sections becomes more challenging. The goal is to allow only one process at a time to execute the critical section while preventing others from accessing it simultaneously.







algo:

Token ring-based mutual exclusion algorithms are used in distributed systems to ensure that only one process at a time can access a critical section or resource. These algorithms use a logical ring structure among the processes, where a token circulates around the ring to grant access to the critical section. One commonly used token ring-based mutual exclusion algorithm is the Suzuki-Kasami algorithm. Here is an explanation of the algorithm:

1. Initialization:
   * Each process in the distributed system is assigned a unique identifier (ID).
   * The processes are organized in a logical ring structure, where each process has a connection to its adjacent processes.
2. Token Acquisition:
   * The token is initially held by a designated process or is created as a "null" token.
   * To acquire the token and access the critical section, a process needs to satisfy certain conditions.
3. Requesting the Token:
   * When a process wants to enter the critical section, it sends a request message to its immediate successor in the ring.
   * The request message includes the process's ID and any additional information required by the algorithm.
4. Processing Requests:
   * When a process receives a request message:
     + If it currently holds the token, it checks if it needs to enter the critical section.
     + If it does not need to enter the critical section, it forwards the request to its successor.
     + If it needs to enter the critical section, it retains the token and sends a grant message to the requesting process.
   * When a process receives a grant message:
     + If it requested the token and is waiting for it, it receives the token and enters the critical section.
     + If it did not request the token or is not waiting for it, it forwards the grant message to its predecessor.
5. Releasing the Token:
   * After a process finishes executing the critical section, it releases the token.
   * The process sends a release message to its successor, indicating that the token is available.
6. Token Circulation:
   * The token circulates around the ring based on the request and grant messages.
   * When a process releases the token, it becomes available for other processes to acquire.

By following this algorithm, processes can take turns accessing the critical section using the circulating token. This ensures mutual exclusion, as only the process holding the token can enter the critical section at any given time.

It's important to note that the Suzuki-Kasami algorithm is a simplified explanation of token ring-based mutual exclusion. There are other variations and enhancements to this algorithm that address issues such as failures, deadlocks, and performance optimization in distributed systems.

what if token is dead

ChatGPT

If the token becomes dead or unavailable in a token ring-based mutual exclusion algorithm, it can cause issues in the system, as no process will be able to acquire the token and access the critical section. In such a scenario, the system needs to have mechanisms in place to handle the dead token situation and recover from it. Here are a few approaches that can be taken:

1. Election of a New Token Holder:
   * When a process detects that the token is dead or missing, it can initiate an election process to select a new token holder.
   * The election process can involve the processes in the ring exchanging messages to determine which process should become the new token holder.
   * Once a new token holder is elected, it starts circulating the token in the ring again, allowing processes to acquire it and access the critical section.
2. Timeout Mechanism:
   * Each process can have a timer associated with the token.
   * If a process does not receive the token within a specified timeout period, it assumes the token is dead or lost.
   * When a process detects the timeout, it can trigger the election of a new token holder as mentioned above.
3. Backup Token:
   * In addition to the primary token, the system can maintain a backup token held by a designated backup process.
   * The backup process monitors the status of the primary token holder and takes over the token if it becomes dead or unavailable.
   * This approach ensures that even if the primary token fails, the backup token can be activated to continue the circulation in the ring.

These are just a few strategies that can be employed to handle a dead token scenario in a token ring-based mutual exclusion algorithm. The choice of approach depends on the specific requirements and characteristics of the distributed system. It is essential to carefully design and implement the recovery mechanisms to ensure the proper functioning and availability of the token.

implementation:

//Step 1 – Write and Compile Server program

import java.io.\*; import

java.net.\*;

public class MutualServer implements Runnable

{

Socket socket=null; static

ServerSocket ss;

MutualServer(Socket newSocket)

{

this.socket=newSocket;

}

public static void main(String args[]) throws IOException

{

ss=new ServerSocket(7000);

System.out.println("Server Started");

while(true)

{

Socket s = ss.accept();

MutualServer es = new MutualServer(s); Thread

t = new Thread(es);

t.start();

}

}

public void run()

{

try

{BufferedReader

in

=

new

BufferedReader(new

InputStreamReader(socket.getInputStream()));

while(true)

{

System.out.println(in.readLine());

}

}

catch(Exception e){ }

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Step 2 – Write and Compile First client program ClientOne.java.

import java.io.\*;

import java.net.\*; public

class ClientOne

{

public static void main(String args[])throws IOException

{

Socket s=new Socket("localhost",7000);

PrintStream out = new PrintStream(s.getOutputStream());

ServerSocket ss = new ServerSocket(7001);

Socket s1 = ss.accept();

BufferedReader in1 = new BufferedReader(new

InputStreamReader(s1.getInputStream()));

PrintStream out1 = new PrintStream(s1.getOutputStream()); BufferedReader br = new

BufferedReader(new InputStreamReader(System.in));

String str="Token";

while(true)

{

if(str.equalsIgnoreCase("Token"))

{

System.out.println("Do you want to send some data");

System.out.println("Enter Yes or No"); str=br.readLine();

if(str.equalsIgnoreCase("Yes"))

{System.out.println("Enter the data");

str=br.readLine();

out.println(str);

}

out1.println("Token");

}

System.out.println("Waiting for Token");

str=in1.readLine();

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*88

//Step 3 – Write and Compile Second client program

import java.io.\*; import

java.net.\*; public class

ClientTwo

{

public static void main(String args[])throws IOException

{

Socket s=new Socket("localhost",7000);

PrintStream out = new PrintStream(s.getOutputStream()); Socket

s2=new Socket("localhost",7001); BufferedReader in2 = new

BufferedReader(new InputStreamReader(s2.getInputStream()));

PrintStream out2 = new PrintStream(s2.getOutputStream()); BufferedReader br = new

BufferedReader(new InputStreamReader(System.in));

String str;

while(true)

{

System.out.println("Waiting for Token");

str=in2.readLine();

if(str.equalsIgnoreCase("Token"))

{

System.out.println("Do you want to send some data");

System.out.println("Enter Yes or No"); str=br.readLine();

if(str.equalsIgnoreCase("Yes")){

System.out.println("Enter the data"); str=br.readLine();

out.println(str);

}

out2.println("Token");

}

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 4 – Run Server Program and keep it running till we connect the clients.

Step 5 – Open new Command prompt and Run ClientOne Program on it and keep it running till ClientTwo starts.

Step 6 – Open one more Command prompt to Run ClientTwo Program. The output allows both the

clients to use token and share their messages with each other using Token Ring. To send the message,

the client has to accept the token by typing type Yes followed by the message alternately and has to

type No to release the token.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

c:\javac MutualServer.java

c:\javac ClientOne.java

c:\javac ClientTwo.java

Step 4 – Run Server Program and keep it running till we connect the clients.

c:\java MutualServer

Step 5 – Open new Command prompt and Run ClientOne Program on it and keep it running till ClientTwo starts.

c:\java ClientOne

Step 6 – Open one more Command prompt to Run ClientTwo Program. The output allows both the

clients to use token and share their messages with each other using Token Ring. To send the message,

c:\java ClientTwo

