###### Experiment Number: 01

**TITLE:**

**PROBLEM STATEMENT**:In an embedded system application Dining Philosophers problem algorithm is used to design a software that uses shared memory between neighboring processes to consume the data. The Data is generated by different Sensors/WSN system Network and stored in MOngoDB (NoSQL). Implementation be done using Scala/ Python/ C++/ Java. Design using Client-Server architecture. Perform Reliability Testing. Use latest open source software modeling, Designing and testing tool/Scrum-it/KADOS, NoSQLUnit and Camel.

**OBJECTIVES:**

1. To understand the working of dining philosophers problem .
2. To understand the application of dining philosphers problem in a shared

memory system .

1. To understand the client-server architecture

**THEORY:**

The dining philosophers problem is a classic concurrency problem dealing with synchronization. Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them. Eating is not limited by the remaining amounts of spaghetti or stomach space, an infinite supply and an infinite demand are assumed.

**Problems :**

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve, i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible.

To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows :

\_ think until the left fork is available; when it is, pick it up;

\_ think until the right fork is available; when it is, pick it up;

\_ when both forks are held, eat for a fixed amount of time;

\_ then, put the right fork down;

\_ then, put the left fork down;

\_ repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

**Resource starvation :**

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

**Mutual Exclusion :**

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

**Solutions :**

**Resource hierarchy solution:**

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks he plans to use.

The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks. While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance.

For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 2 and 3 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

**Arbitrary solution:**

Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until he has picked up both his forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex. In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

**Chandy/ Misra solution:**

In 1984, K. Mani Chandy and J. Misra proposed a different solution to the dining philosophers problem to allow for arbitrary agents (numbered P1, ..., Pn) to contend for an arbitrary number of resources, unlike Dijkstra's solution. It is also completely distributed and requires no central authority after initialization. However, it violates the requirement that "the philosophers do not speak to each other" (due to the request messages).

* For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (n for agent Pn). Each fork can either be dirty or clean. Initially, all forks are dirty.
* When a philosopher wants to use a set of resources (i.e. eat), he must obtain the forks from his contending neighbors. For all such forks he does not have, he sends a request message.
* When a philosopher with a fork receives a request message, he keeps the fork if it is clean, but gives it up when it is dirty. If he sends the fork over, he cleans the fork before doing so.
* After a philosopher is done eating, all his forks become dirty. If another philosopher had previously requested one of the forks, he cleans the fork and sends it.

This solution also allows for a large degree of concurrency, and will solve an arbitrarily large problem. It also solves the starvation problem. The clean /dirty labels act as a way of giving preference to the most "starved" processes, and a disadvantage to processes that have just "eaten". One could compare their solution to one where philosophers are not allowed to eat twice in a row without letting others use the forks in between. Their solution is more exible than that, but has an element tending in that direction.

**MATHEMATICAL MODEL:**

The Dining Philosopher’s problem can be described as a problem space P:

Where, P = { S , I , O , F , E }

S = { Start state consisting of initial state of client and server, initial state of sockets. }

I = { Input to the system: I1 and I2 }

Where I1 = { hostname }

I2 = { port }

O = { Output state : O1 , O2 }

where O1 = { Allow lock on the shared file to the client. }

O2 = { Deny lock on the shared file to the client. }

F = { Functions used : f1 , f2 , f3 , f4 }

where f1 = { Status of the mutex lock }

f2 = { Acquire lock on the shared file }

f3 = { Release lock on the shared file }

f4 = { Connect(), accept() }

E = { End state of the system : E1 , E2 }

where E1 = { Success state : Client are able to access the files in the shared memory without any deadlock. }

E2 = { Failure state : Deadlock in acquiring the files. }

**Algorithm :**

**Input :-**

Port , Hostname

**Output :-**

Producers and consumers interacting with each other using shared memory with release and acquire feature.

1. Initiate the shared memory and connection between the producer and consumer

* Init(shared-memory)
* Connect(port, host)

2. Producer produces file into shared memory and waits for connection.

* Produce(txt)
* Wait(consumer)

3. Consumers are in the thinking mode and any one of the consumers want to access shared file.

* Request(txt)

4. Apply lock on the shared memory region.

* Lock(txt,consumer id)

5. All other consumers are not allowed in the shared region and wait for unlocking the file and go to Step 3

6. Consumer holding the lock will unlock the shared memory region and other consumers can now access the region and Go to step 4.

**Execution Steps :**

Execute the program with the following commands:

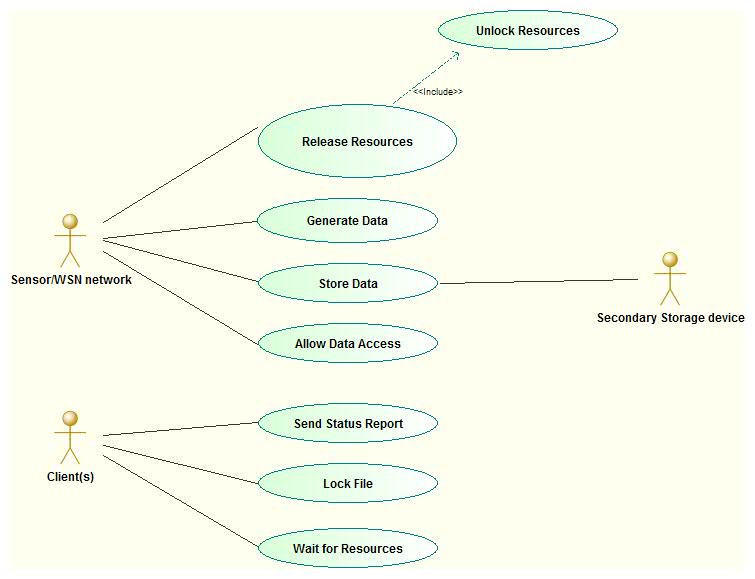
1. First run producer code:

* javac Server Producer.java
* java Server Producer

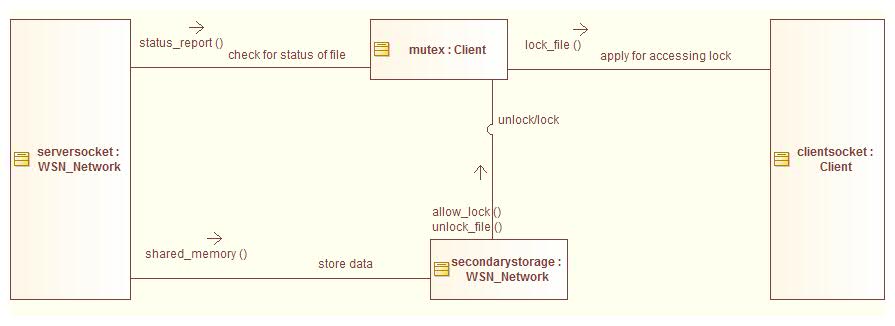
2. Next run consumer code on different terminals one by one :

* javac Consumer.java
* java Consumer

**UML DIAGRAMS :**



**Communication diagram :**



**TEST CASES :**

Reliability Testing is about exercising an application so that failures are discovered and removed before the system is deployed. The purpose of reliability testing is to determine product reliability, and to determine whether the software meets the customer’s reliability requirements. According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment.

* **Positive test Cases**

1. Test case 1

* Test case id : T1
* Unit to test : No. of client connections
* Execution Steps :

Run the server

Run a number of clients simultaneously

* Type of test : Load Test
* Expected Result : Server is able to accept almost 10 connections

reliably

* Actual Result : Server accepting all client connections
* Remark : Pass

2. Test case 2

* Test case id : T2
* Unit to test : File locking and unlocking
* Execution Steps :

Run the server

Run a number of clients simultaneously

Try to access the \_le by all the clients at the same time

* Type of test : Stress Test
* Expected Result : Only single client allowed access to the file
* Actual Result : At a time only one client is able to access the

shared file from the server

* Remark : Pass

**Negative test Cases**

1. Test case 1

* Test case id : T3
* Unit to test : Acquiring and releasing locks
* Execution Steps :

Run the server

Run a number of clients simultaneously

Try to acquire locks on the already locked file

* Type of test : Load Test
* Expected Result : The server should get terminated due to lot of

exceptions

* Actual Result : Server application doesnot crash
* Remark : Fail

2. Test case 2

* Test case id : T4
* Unit to test : File availability
* Execution Steps :

Delete the file

Run the server

Run a number of clients simultaneously

Try to access the file by the clients

* Type of test : Stress Test
* Expected Result : The client application shouldnot work properly
* Actual Result : Client application crashes
* Remark : Pass

**CONCLUSION:**

Hence we have successfully implemented a client-server architecture using shared

memory. Producers produce data and consumers consume data in a deadlock-

free manner.

**COURSE OUTCOMES ACHIEVED:**

CO-II : To use software design methods and testing.

CO-III : To solve problems for multi-core or distributed, concurrent/Parallel

environments

**FAQ’s**

1. What is dining philospher's problem ? In what problem scenario can we

use dining philospher's algorithm ?

2. What are the necessary conditions for deadlock to occur ?

3. What do you mean by the terms deadlock-free and lockout-free ?

4. What is mutex and semaphore ?

5. Enlist the strategies to break a deadlock ?

**CODE :**

**Producer Code :**

//package cl3\_pract;

import java.io.BufferedInputStream;

import java.io.BufferedReader;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

public class Server\_Producer {

public static String flag=null,flag1=null;

public final static int SOCKET\_PORT = 13267; // you may change this

public final static String FILE\_TO\_SEND = "s.txt";

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

FileInputStream fis = null;

BufferedInputStream bis = null;

OutputStream os = null;

ServerSocket servsock = null;

Socket sock = null;

try {

servsock = new ServerSocket(SOCKET\_PORT);

while (true) {

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

try {

sock = servsock.accept();

System.out.println("Accepted connection : " + sock);

// send file

DataOutputStream outToClient = new DataOutputStream(sock.getOutputStream());

BufferedReader inFromClient = new BufferedReader(new InputStreamReader(sock.getInputStream()));

//outToClient.writeBytes("1" + '\n');

flag = inFromClient.readLine();

if(flag.equals("1") && flag1==null){

flag1="1";

outToClient.writeBytes("Can access the file!");

System.out.println("flag=="+flag);

}

else{

flag1=null;

flag="1";

System.out.println("flag=="+flag);

outToClient.writeBytes("Cannot access the file!");

}

System.out.println("Done.");

}

finally {

if (bis != null) bis.close();

if (os != null) os.close();

if (sock!=null) sock.close();

}

}

}

finally {

if (servsock != null) servsock.close();

}

}

}

**Consumer Code :**

package cl3\_pract;

import java.awt.Container;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.io.BufferedReader;

import java.io.DataOutputStream;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.RandomAccessFile;

import java.net.Socket;

import java.net.UnknownHostException;

import java.nio.channels.FileChannel;

import java.nio.channels.FileLock;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

public class Consumer {

public static JFrame fr;

public static JLabel title,status\_of\_file;

public static JButton acquire,release;

public static Container c;

public static RandomAccessFile file;

public static FileLock fileLock = null;

public static Socket sclient;

public static DataOutputStream outToServer;

public static BufferedReader inFromServer;

public static int flg;

public Consumer() throws Exception, IOException {

//Socket

sclient=new Socket("127.0.0.1",13267);

System.out.println("Connected.....");

outToServer = new DataOutputStream(sclient.getOutputStream());

inFromServer = new BufferedReader(new InputStreamReader(sclient.getInputStream()));

//GUI part starts

fr=new JFrame();

c=fr.getContentPane();

c.setLayout(null);

title=new JLabel("Dinig Philosopher problem",JLabel.CENTER);

status\_of\_file=new JLabel("Status:: Unknown",JLabel.CENTER);

acquire=new JButton("Acquire");

release=new JButton("Release");

fr.setBounds(0, 0, 400, 200);

title.setBounds(0, 20, fr.getWidth(), 30);

status\_of\_file.setBounds(0, 50, fr.getWidth(), 30);

acquire.setBounds(90, 90,190,30);

release.setBounds(90, 90,190,30);

release.setVisible(false);

c.add(release);

c.add(acquire);

c.add(status\_of\_file);

c.add(title);

fr.setVisible(true);

fr.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

//GUI part end

//Access file here

try{

//FileInputStream input = new FileInputStream("\\\\Nits\_sankpal-pc\\d\\s.txt");

FileInputStream input = new FileInputStream("s.txt");

int data = input.read();

System.out.println(data);

}catch (Exception exception){

//exception.printStackTrace();

}

//event handling on buttons in GUI

acquire.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent arg0) {

try {

//file = new RandomAccessFile("\\\\Nits\_sankpal-pc\\d\\s.txt", "rw");

String sentence;

outToServer.writeBytes("1" + '\n');

sentence = inFromServer.readLine();

if(sentence.contains("Cannot")){

}

else{

acquire.setVisible(false);

release.setVisible(true);

}

System.out.println("msg::"+sentence);

status\_of\_file.setText("status::"+sentence);

// sclient.close();

file = new RandomAccessFile("s.txt", "rw");

FileChannel fileChannel = file.getChannel();

//acquire lock exclusively

fileLock = fileChannel.tryLock();

if (fileLock != null){

System.out.println("File is locked");

//status\_of\_file.setText("Status:: "+sentence);

}

} catch (IOException e) {

// TODO Auto-generated catch block

//e.printStackTrace();

}

try{

// FileInputStream input = new FileInputStream("\\\\Nits\_sankpal-pc\\d\\s.txt");

FileInputStream input = new FileInputStream("s.txt");

int data = input.read();

System.out.println(data);

}catch (Exception exception){

//exception.printStackTrace();

//status\_of\_file.setText("Status:: File is acquired");

//status\_of\_file.setText("Status::"+sen);

}

}

});

release.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent arg0) {

acquire.setVisible(true);

release.setVisible(false);

//Release lock

if (fileLock != null){

try {

//flg=1;

fileLock.release();

status\_of\_file.setText("Status:: File is released");

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

});

}

public static void main(String[] args) throws Exception, Exception {

Consumer pd=new Consumer();

}

}

**Output Screenshots :**

