Name:Atharva Pawar Roll No:-9427 Exp:8

BECOMPS A

**Aim:** To implement Mutual Exclusion / Deadlock Detection

**Lab Outcome:**

Demonstrate Mutual Exclusion algorithms and deadlock handling

**Theory:**

**Mutual Exclusion in Distributed System:**

Mutual exclusion is a concurrency control property which is introduced to prevent race

conditions. It is the requirement that a process cannot enter its critical section while

another concurrent process is currently present or executing in its critical section i.e., only

one process is allowed to execute the critical section at any given instance of time.

In Distributed systems, we neither have shared memory nor a common physical clock and

therefore we cannot solve mutual exclusion problem using shared variables. To eliminate

the mutual exclusion problem in distributed system approach based on message passing is

used.

Requirements of Mutual exclusion Algorithm:

• **No Deadlock:**

Two or more site should not endlessly wait for any message that will never arrive.

• **No Starvation:**

Any site should not wait indefinitely to execute critical section while other sites are

repeatedly executing critical section

• **Fairness:**

Each site should get a fair chance to execute critical section. Any request to execute

critical section must be executed in the order they are made.

• **Fault Tolerance:**

In case of failure, it should be able to recognize it by itself in order to continue

functioning without any disruption.

Solution to distributed mutual exclusion:

As we know shared variables or a local kernel cannot be used to implement mutual

exclusion in distributed systems. Message passing is a way to implement mutual exclusion.

Below are the three approaches based on message passing to implement mutual exclusion

in distributed systems:Token Based Algorithm:

• A unique token is shared among all the sites.

• If a site possesses the unique token, it is allowed to enter its critical section

• This approach uses sequence number to order requests for the critical section.

• Each requests for critical section contains a sequence number. This sequence

number is used to distinguish old and current requests.

• This approach insures Mutual exclusion as the token is unique

• Example: Suzuki-Kasami’s Broadcast Algorithm

Non-token based approach:

• A site communicates with other sites in order to determine which sites should

execute critical section next. This requires exchange of two or more successive round

of messages among sites.

• This approach use timestamps instead of sequence number to order requests for the

critical section.

• When ever a site make request for critical section, it gets a timestamp. Timestamp is

also used to resolve any conflict between critical section requests.

• All algorithm which follows non-token based approach maintains a logical clock.

Logical clocks get updated according to Lamport’s scheme

• Example: Lamport's algorithm, Ricart–Agrawala algorithm

Quorum based approach:

• Instead of requesting permission to execute the critical section from all other sites,

Each site requests only a subset of sites which is called a quorum.

• Any two subsets of sites or Quorum contains a common site.

• This common site is responsible to ensure mutual exclusion

• Example: Maekawa’s Algorithm

**Deadlock Handling:**

The following are the strategies used for Deadlock Handling in Distributed System:

1. **Deadlock Prevention:** As the name implies, this strategy ensures that deadlock can never

happen because system designing is carried out in such a way. If any one of the deadlock

causing conditions is not met then deadlock can be prevented. Following are the three methods used for preventing deadlocks by making one of the deadlock conditions to be

unsatisfied:

Collective Requests: In this strategy, all the processes will declare the required

resources for their execution beforehand and will be allowed to execute only if there

is the availability of all the required resources. When the process ends up with

processing then only resources will be released. Hence, the hold and wait condition

of deadlock will be prevented.

But the issue is initial resource requirements of a process before it starts are based

on an assumption and not because they will be required. So, resources will be

unnecessarily occupied by a process and prior allocation of resources also affects

potential concurrency.

Ordered Requests: In this strategy, ordering is imposed on the resources and thus,

process requests for resources in increasing order. Hence, the circular wait condition

of deadlock can be prevented.

An ordering strictly indicates that a process never asks for a low resource while

holding a high one.

There are two more ways of dealing with global timing and transactions in

distributed systems, both of which are based on the principle of assigning a global

timestamp to each transaction as soon as it begins.

It is better to give priority to the old processes because of their long existence and

might be holding more resources.

It also eliminates starvation issues as the younger transaction will eventually be out

of the system.

Pre-emption: Resource allocation strategies that reject no-pre-emption conditions

can be used to avoid deadlocks.

Wait-die: If an older process requires a resource held by a younger process, the

latter will have to wait. A young process will be destroyed if it requests a resource

controlled by an older process.

Wound-wait: If an old process seeks a resource held by a young process, the young

process will be pre-empted, wounded, and killed, and the old process will resume

and wait. If a young process needs a resource held by an older process, it will have to

wait.

2. **Deadlock Avoidance:** In this strategy, deadlock can be avoided by examining the state of

the system at every step. The distributed system reviews the allocation of resources and

wherever it finds an unsafe state, the system backtracks one step and again comes to the safe state. For this, resource allocation takes time whenever requested by a process. Firstly,

the system analysis occurs whether the granting of resources will make the system in a safe

state or unsafe state then only allocation will be made.

A safe state refers to the state when the system is not in deadlocked state and order is there

for the process regarding the granting of requests.

An unsafe state refers to the state when no safe sequence exists for the system. Safe

sequence implies the ordering of a process in such a way that all the processes run to

completion in a safe state.

3. **Deadlock Detection and Recovery:** In this strategy, deadlock is detected and an attempt

is made to resolve the deadlock state of the system. These approaches rely on a Wait-For

Graph (WFG), which is generated and evaluated for cycles in some methods.

The following two requirements must be met by a deadlock detection algorithm:

Progress: In a given period, the algorithm must find all existing deadlocks. There

should be no deadlock existing in the system which is undetected under this

condition. To put it another way, after all, wait-for dependencies for a deadlock have

arisen, the algorithm should not wait for any additional events to detect the

deadlock.

No False Deadlocks: Deadlocks that do not exist should not be reported by the

algorithm which is called phantom or false deadlocks.

There are different types of deadlock detection techniques:

Centralized Deadlock Detector: The resource graph for the entire system is managed

by a central coordinator. When the coordinator detects a cycle, it terminates one of

the processes involved in the cycle to break the deadlock. Messages must be passed

when updating the coordinator’s graph. Following are the methods:

A message must be provided to the coordinator whenever an arc is created or

removed from the resource graph.

Hierarchical Deadlock Detector: In this approach, deadlock detectors are arranged in

a hierarchy. Here, only those deadlocks can be detected that fall within their range.

Distributed Deadlock Detector: In this approach, detectors are distributed so that all

the sites can fully participate to resolve the deadlock state. In one of the following

below four classes for the Distributed Detection Algorithm- The probe-based scheme

can be used for this purpose. It follows local WFGs to detect local deadlocks and

probe messages to detect global deadlocks. There are four classes for the Distributed Detection Algorithm:

• *Path-pushing*: In path-pushing algorithms, the detection of distributed deadlocks is

carried out by maintaining an explicit global WFG.

• *Edge-chasing*: In an edge-chasing algorithm, probe messages are used to detect the

presence of a cycle in a distributed graph structure along the edges of the graph.

• *Diffusion computation*: Here, the computation for deadlock detection is dispersed

throughout the system’s WFG.

• *Global state detection*: The detection of Distributed deadlocks can be made by taking

a snapshot of the system and then inspecting it for signs of a deadlock.

**Implementation:**

*Suzuki-Kasami Algorithm*:

1. **To enter Critical section:**

• When a site Si wants to enter the critical section and it does not have the

token then it increments its sequence number **RNi[i]** and sends a request

message **REQUEST(i, sn)** to all other sites in order to request the token.

Here **sn** is update value of **RNi[i]**

• When a site Sj receives the request message **REQUEST(i, sn)** from site Si, it

sets **RNj[i]** to maximum of **RNj[i]** and **sn** i.e **RNj[i]** = max(**RNj[i]**, **sn**).

• After updating **RNj[i]**, Site Sj sends the token to site Si if it has token

and **RNj[i]** = **LN[i]** + 1

2. **To execute the critical section:**

• Site Si executes the critical section if it has acquired the token.

3. **To release the critical section:**

After finishing the execution Site Si exits the critical section and does following:

• sets **LN[i]** = **RNi[i]** to indicate that its critical section request **RNi[i]** has been

executed

• For every site Sj, whose ID is not present in the token queue **Q**, it appends

its ID to **Q** if **RNi[j]** = **LN[j]** + 1 to indicate that site Sj has an outstanding

request.

• After above updation, if the Queue **Q** is non-empty, it pops a site ID from

the **Q** and sends the token to site indicated by popped ID.

• If the queue **Q** is empty, it keeps the token

**Message Complexity:**

The algorithm requires 0 message invocation if the site already holds the idle token at the

time of critical section request or maximum of N message per critical section execution.

This N messages involves

• (N – 1) request messages

• 1 reply message

Code:import keyboard

import time

import threading

runningP = -1

# RN arrays of processes

RN = {

0: [0, 0, 0, 0, 0],

1: [0, 0, 0, 0, 0],

2: [0, 0, 0, 0, 0],

3: [0, 0, 0, 0, 0],

4: [0, 0, 0, 0, 0]

}

token = {

"token\_owner": 2,

"Q": [],

"LN": [0, 0, 0, 0, 0],

"isRunning": False

}

def dispCurrentRNState():

for key, value in RN.items():

print(key, ": ", value)

def updateRN(processNo, sequenceNumber):

for key, value in RN.items():

value[processNo] = max(value[processNo], sequenceNumber)

# Execute cs and remaining tasks

def executeCS(processForCS):

print("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

print(f"Process {processForCS} executing CS...")

print('Token owner is: {}'.format(token["token\_owner"]))

time.sleep(10)

print(f'\nProcess {processForCS} has completed running CS')

# Process completed CS

token["isRunning"] = False

#update LN

token["LN"][processForCS] = RN[processForCS][processForCS]

# print(f"Process Completed CS")

#Check For Outstanding Requests # For every site Sj, whose ID is not present in the token queue Q, it appends

its ID to Q if RNi[j] = LN[j] + 1 to indicate that site Sj has an outstanding

request.

for index, val in enumerate(RN[token["token\_owner"]]):

# print("Running P: ", runningP)

if(val == token["LN"][index] + 1 and index != runningP and index not in

token["Q"]):

# outstanding Requests

print(f'Process {index}\'s request is outstanding, it will be added to

Token\'s Queue')

token["Q"].append(index)

print(f'Queue: {token["Q"]}')

#Handing out the token

if(len(token["Q"]) != 0):

# pop a process from the queue and give it the token

poppedPs = token["Q"].pop(0)

token["token\_owner"] = poppedPs

token["isRunning"] = True

executeCS(poppedPs)

if \_\_name\_\_ == "\_\_main\_\_":

# print("Press Key E to exit")

print("Running Main Again")

# Display Current State of RN Arrays

print("Current RN Arrays: ")

dispCurrentRNState()

print(" ")

print('Token owner is: {}'.format(token["token\_owner"]))

while True:

if(token["isRunning"]):

processes = input(

"Enter Process Numbers which want to access C.S separated

by space (Click N for None): ")

if(processes != 'N'):

psList = processes.strip().split(" ")

print(" ")

for ps in psList:

processForCS = int(ps)

print(f"\*\*\*\*\* Process {processForCS} \*\*\*\*\*")

seqNo = RN[processForCS][processForCS]+1

# Broadcasting Request

print(f"Process No.: {processForCS}")

print(f"Sequence No.: {seqNo}")print(f"Broadcasting Request ({processForCS} ,

{seqNo}) .......")

time.sleep(2)

print("Broadcast complete")

print(" ")

# Updating RN Arrays

print("Updating RN Arrays at all process sites")

updateRN(processForCS, seqNo)

print("Current RN Arrays: ")

dispCurrentRNState()

print(" ")

else:

processForCS = int(input("Enter Process No. which wants to

access C.S: "))

seqNo = RN[processForCS][processForCS]+1

# Broadcasting Request

print(f"Process No.: {processForCS}")

print(f"Sequence No.: {seqNo}")

print(f"Broadcasting Request ({processForCS} , {seqNo})

.......")

time.sleep(2)

print("Broadcast complete")

print(" ")

# Updating RN Arrays

print("Updating RN Arrays at all process sites")

updateRN(processForCS, seqNo)

print("Current RN Arrays: ")

dispCurrentRNState()

print(" ")

# Check condition of sending token: RNj[i] = LN[i] + 1

if(RN[token["token\_owner"]][processForCS] ==

token["LN"][processForCS] + 1):

# give the token

print(f"Conditions met, giving token to

{processForCS}...")

token["token\_owner"] = processForCS

# print(f"New Token Owner: {token["token\_owner"]}")

print('Token owner is: {}'.format(token["token\_owner"]))

token["isRunning"] = True

runningP = processForCS

thread = threading.Thread(target=executeCS,

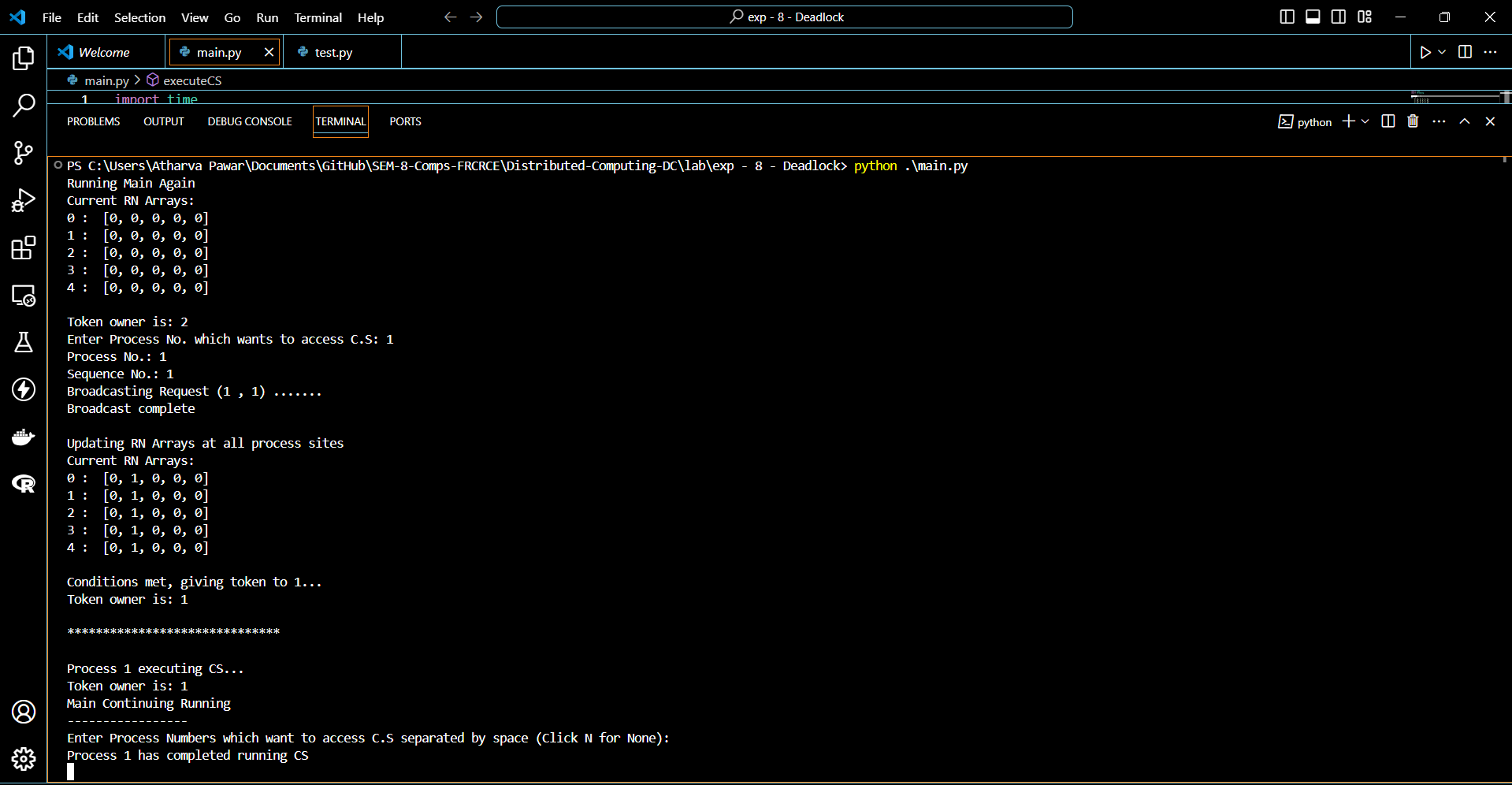
args=(processForCS, ))

thread.start()print("Main Continuing Running")

if keyboard.is\_pressed('E'):

break

**Output:**



**Conclusion:**

In conclusion, our experiment focused on studying mutual exclusion and deadlock handling

in a distributed system. We implemented a simulation of the Suzuki-Kasami algorithm as a

solution to the challenges we identified. The algorithm demonstrated how processes in a

distributed system can coordinate with each other to prevent race conditions and ensure

mutual exclusion, without causing deadlocks.

**Postlab Questions:**

1. Explain the different ways of recovery from deadlock.

2. What are the features of CMH algorithm

