IARE
INSTITUTE OF
AERONAUTICAL ENGINEERING
(An Autonomous Institute affiliated to JNTUH, Hyderabad)
Dundigal, Hyderabad - 500 043

LABORATORY WORK BOOK

Name of the Student : RAGHERLA SANTHOSH

Roll Number									
2	3	9	5	1	A	1	2	6	3

Class: IT-B Semester: 03

Course Code : ACSD10 Course Name : OS Laboratory

Name of the Course Faculty: Mr. N. Raghava Rao Faculty ID : IARE 10924

Exercise Number : 07 Week Number : 07 Date : 01/11/2024

S. No.	Exercise Number	EXERCISE NAME	MARKS AWARDED: Aim/ Preparation	Algorithm / Procedure: Performance in the Lab	Source Code: Calculations and Graphs	Program Execution: Results and Error Analysis	Viva - Voce	Total
			4	4	4	4	4	20
1	7.1	Urban OS - RAG	4	4	4	4	4	20
2	7.2	Urban OS - WFG						
3	7.3	The Library Conference						
4	7.4	The Dining Philosophers						
5	7.5	Banker's Algorithm						
6								
7								
8								
9								
10								
11								
12								

Santhosh
Signature of the Student

Signature of the Faculty

7. Resource Allocation.

7.1 UrbanOS – Resource Allocation Graph (RAG)

AIM:- Write a Program On UrbanOS using Resource Allocation for UrbanOS employs.

PROGRAM :-

```
class ResourceAllocationGraph:
  def __init__(self):
    self.graph = defaultdict(list)
    self.process = {'A', 'B', 'C', 'D'}
    self.resources = {'CPU', 'Memory', 'File', 'Network'}
    self.allocated = {resource: None for resource in
                      self.resources}
    self.requests = defaultdict(list)
  def request_resource(self, process, resource):
    if self.allocated[resource] is None:
```

ROLL NUMBER :

```
Self. allocated [resource] = Process
Print (f"{Process} allocated {resource}")
else :
Self. graph [Process]. append (resource)
Self. graph [resource]. append (Self. allocated [resource])
Print (f"{process} requests {resource}, waiting for it
        to be freed by {Self. allocated [resource]}")
def release_resource (Self, process, resource):
 if Self. allocated [resource] == process :
  Self. allocated [resource] = None
  Self. graph [Process]. remove (resource)
  Self. graph [resource]. remove (Process)
  Print (f"{process} released {resource}")
 else :
  Print (f"{Process} does not hold {resource}
        cannot release")
:
```

```
def detect_deadlock(self):
    visited = set()
    stack = set()

    def dfs(node):
        if node in stack:
            return True
        if node in visited:
            return False
        visited.add(node)
        stack.add(node)
        for neighbor in self.graph[node]:
            if dfs(neighbor):
                return True
        stack.remove(node)
        return False

    for node in list(self.processes) +
                list(self.resources):
```

```
if node not in visited :
  if dfs (node):
    Print(" Deadlock detected in RAG ")
    return True
  Print (" NO deadlock detected ")
  return False
def Show_rag ( Self ) :
  Print (" Resource Allocation Graph:")
  for node, neighbors in Self. graph. items():
    Print (f" {node} -> {','. join (neighbors)} ")
rag = ResourceAllocationGraph()
rag. request_ resource ('A', 'CPU')
rag. request_ resource ('A', 'Memory')
rag. request_ resource (' B', 'CPU')
rag. request_ resource ('B', 'File')
rag. request_ resource ('C', 'GPU')
```

```
rag.request_resource('C', 'Memory')
rag.request_resource('D', 'Network')
rag.request_resource('D', 'CPU')
rag.show_rag()
if rag.detect_deadlock():
    Print("Deadlock resolution needed")
else:
    Print("System running without feedback")
rag.show_rag()
rag.detect_deadlock()
```

OUTPUT :-

The Program is executed Successfully.

7.2 UrbanOS - Wait - for - Graph (WFG)..

AIM :- Write a program for UrbanOS - WFG, which helps manage dependencies and prevent potential deadlock situations among concurrent processes.:

PROGRAM :-

```
from collections import defaultdict
class WaitForGraph :
  def __init__(self):
    self.graph = defaultdict(list)
    self.processes = {'A', 'B', 'C', 'D'}
    self.resources = {'CPU', 'Traffic Data', 'Memory',
                      'Network', 'Emergency Services',
                      'Database'}
  def add_dependency(self, waiting_process,
                     blocking_process):
```

```
Self . graph [waiting_process] . append (blocking_process)
Print (f" Process {waiting_process} is waiting for
        Process {blocking_process}")
def remove_dependency (Self, waiting_process,
                       blocking_process) :
if blocking_process in Self . graph [waiting_process] :
 Self . graph [waiting_process] . remove (blocking_process)
 Print (f" Process {waiting_process} no longer waits
        for Process {blocking_process}")
def detect_deadlock (Self) :
 Visited = Set()
 Stack = Set()
def dfs (node) :
 if node in Stack :
   return True
 if node in Visited :
   return False
```

```
        Visited.add (node)
        Stack.add (node)
wfg = WaitForGraph()
wfg.add_dependency ('A', 'B')
wfg.add_dependency ('B', 'C')
wfg.add_dependency ('C', 'D')
wfg.add_dependency ('D', 'A')
wfg.show_wfg()
if wfg.detect_deadlock():
    print ("Deadlock resolution needed")
else:
    print ("System running without deadlock")
wfg.remove_dependency ('D', 'A')
print ("\n After resolving dependency:")
wfg.show_wfg()
wfg.detect_deadlock()
```

OUTPUT :- Executed Successfully.

7.3 The Library Conference.

AIM :- Write a Program on Determine if there is deadlock in the System, If So describe the Circular wait Condition and Suggest a way to Prevent or resolve the deadlock.

PROGRAM :-

```
from collections import defaultdict
class ResourceAllocationGraph:
  def __init__(Self):
    Self.graph = defaultdict(list)
    Self.resources = {'R1', 'R2', 'R3', 'P1', 'P2'}
    Self.teams = {'A', 'B', 'C'}
  def add_holding(Self, holding_team, resource):
    Self.graph[resource].append(holding_team)
    Print(f"Resource {resource} is held by Team
          {holding_team}")
```

```
def detect_deadlock (self):
    visited = set()
    stack = set()
def dfs (node) :
    if node in stack :
        return True
    if node in visited : :
        return False
visited . add (node)
stack . add (node)
for neighbor in self. graph [node] :
    if dfs ( neighbor) :
        return True
stack . remove ( node)
return False

rag = ResourceAllocationGraph( )
rag. add _holding ('A', 'R1')
rag. add_ dependency ( 'A', 'P1')
```

```
rag.add_holding('B', 'P2')
rag.add_dependency('B', 'R2')
rag.add_holding('C', 'R3')
rag.add_dependency('C', 'P2')
rag.show_rag()
if rag.detect_deadlock():
    Print("\n Deadlock detected. A->B->C->B")
    Print("Resolution Strategies:")
    Print("1. prevention: Request all resources at once
          to avoid circular wait")
    Print("2. Avoidance: Use Banker's Algorithm to
          ensure safe allocation")
    Print("3. Detection & Recovery: Deadlock cycle")
else:
    Print("System running without deadlock")
```

OUTPUT : -

The Program is executed Successfully.

7.4 # The Dining Philosophers.

AIM :- Write a Program to Analyze the System for Potential deadlock for the Dining Philosophers & What Strategies can be used to Prevent or resolve the deadlock.

PROGRAM :-

```
import threading
import time
class philospher(threading.Thread):
  def __init__(self, id, left_utensil, right_untensil):
    threading.Thread.__init__(self).
    self.id = id
    self.left_utensil = left_utensil
    self.right_utensil = right_utensil
  def run(self):
    while True:
```

ROLL NUMBER :

```
    Print (f" Philosopher {self.id} is thinking")
    time.sleep(1)
utensils = [threading.Lock() for _ in range(5)]
Philosophers = [
  Philosopher (1, utensils[0], utensils[1]),
  Philosopher (2, utensils[1], utensils[2]),
  Philosopher (3, utensils[2], utensils[3]),
  Philosopher (4, utensils[3], utensils[4]),
  Philosopher (5, utensils[4], utensils[0]),
]
for philosopher in philosophers:
  philosopher.join().
```

OUTPUT :-

The Program is Executed Successfully..

7.5 Banker's Algorithm.

AIM :- Write a program on Banker's Algorithm, It is a deadlock avoidance algorithm used in operating systems to manage resource allocation & prevent deadlocks.

PROGRAM :-

```
available = [3, 2, 2]
allocation = [
 [0, 1, 0],
 [2, 0, 0],
 [3, 0, 2],
 [2, 1, 1],
 [0, 0, 2],
]
maximum = [
 [7, 5, 3],
 [3, 2, 2,],
 [9, 0, 2],
]
```

ROLL NUMBER :

```
def calculate_need (maximum, allocation):
 need = []
 for i in range ( len(maximum)):
  need.append ( [maximum [i][j] - allocation [i][j] for j in
                range (len (maximum [0]))])
 return need

Print ( " Initial State:")
Print (" Available :", Work )
Print ("Need matrix:", need )
if len ( Safe_Sequence) = = len (allocation ):
  Print (" System is in a Safe State").
  Print (" Safe Sequence:", ["P"+ Str (i+1) for i in
            Safe_Sequence] )
  return True
else : print (" System is not in a Safe State")
       return False
is_Safe (available, allocation, maximum).
```

OUTPUT :- Executed Successfully.