LAB-7

AIM: Implement an Algorithm for DeadLock Detection.

THEORY:

A deadlock is a situation in a multiprogramming environment where a set of processes becomes blocked because each process is waiting for a resource that another process holds.

Conditions for Deadlock (Coffman's Conditions):

- **Mutual Exclusion**: At least one resource is held in a non-shareable mode.
- **Hold and Wait**: A process is holding resources and waiting for additional resources.
- **No Preemption**: Resources cannot be forcibly removed.
- **Circular Wait**: A set of processes forms a circular chain where each process is waiting for a resource held by the next.

Problems in Deadlock Detection

1. **Complexity**: Deadlock detection algorithms can be computationally expensive.
2. **State Explosion**: Large systems with many processes and resources may create very complex dependency graphs.
3. **False Positives**: Detection algorithms may incorrectly signal deadlocks in systems with slow but progressing execution.
4. **Overheads**: Frequent detection checks consume processing power.

A **Resource Allocation Graph (RAG)** is a graphical representation used to model the allocation of resources to processes in a system. It is widely used to analyze and detect deadlocks in systems.

A RAG can be visualized as a bipartite graph with:

- Two types of nodes: processes and resources.
- Two types of directed edges: **requests** and **allocations**.

RAG Advantages

1. **Visualization**: Helps system administrators easily identify deadlocks through cycles.

2. **Granularity:** Provides detailed information about resource allocation and process requests.
3. **Versatility**: Can model systems with single or multiple resource instances.

RAG Limitations

1. **Complexity for Large Systems**: For systems with many processes and resources, the graph becomes complex and hard to manage.
2. **Multiple Instances of Resources**: Requires additional checks for deadlock detection.

Algorithm for Deadlock Detection Using RAG

Input:

- List of processes and resources.
- Edges indicating requests and allocations.

Steps:

1. Construct the RAG:
    - Add nodes for processes and resources.
    - Add edges based on requests and allocations.
2. Cycle Detection:
    - Use Depth-First Search (DFS) to traverse the graph.
    - Maintain a stack to track visited nodes.
    - If a node is revisited while still in the stack, a cycle is detected.
3. Determine Deadlock:
    - If a cycle exists:
        - For single-instance resources: Deadlock exists.
        - For multiple-instance resources: Verify resource availability.

A **Wait-for Graph (WFG)** is a simplified version of the **Resource Allocation Graph (RAG)**, used specifically for deadlock detection. It represents only the processes and how they depend on each other to acquire resources.

In WFG there is a directed graph where:

- **Nodes** represent processes (P1,P2,…,Pn).
- **Edges** represent "wait-for" relationships (Pi→Pj).

Pi→Pj: Process Pi is waiting for a resource held by process Pj.

A cycle in the WFG directly indicates a deadlock because it represents circular waiting among processes.

Algorithm for Deadlock Detection Using WFG

Input:

- List of processes.
- Wait-for relationships as directed edges (Pi→Pj).

Steps:

1. Construct the WFG:
    - Represent processes as nodes.
    - Add edges for wait-for relationships based on resource requests.
2. Cycle Detection:
    - Use Depth-First Search (DFS) to traverse the graph.
    - Maintain a stack to detect cycles during traversal.
3. Check for Deadlock:
    - If a cycle exists in the WFG, deadlock is present.

Output:

- Report whether deadlock exists.

Advantages of WFG

1. **Simplified Representation**:
    - Only processes and their dependencies are shown.
    - Easier to understand and analyze compared to RAG.
2. **Direct Cycle Detection**:
    - A cycle in the graph directly indicates a deadlock, eliminating additional checks.
3. **Efficiency**:
    - Smaller graph size compared to RAG, making traversal faster.

CODE:

RAG:

```
from collections import defaultdict
class RAG:
    def __init__(self):
        # Initialize the graph using defaultdict of lists to store
edges.
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        """Add a directed edge from node u to node v.
        In the context of RAG:
        - If u is a process and v is a resource, it means the process
u is requesting resource v.
        - If u is a resource and v is a process, it means the
resource u is allocated to process v."""
        self.graph[u].append(v)  # Add v to the adjacency list of u

    def is_cycle(self, visited, rec_stack, node):
        """Detect cycles using Depth-First Search (DFS).
        A cycle in the RAG means a deadlock exists."""
        visited[node] = True  # Mark the current node as visited
        rec_stack[node] = True  # Mark the node in the recursion
stack to track the DFS path
        # Recurse for all neighbors of the current node
        for neighbor in self.graph[node]:
            if not visited[neighbor]:  # If the neighbor is not
visited, continue DFS
                if self.is_cycle(visited, rec_stack, neighbor):
                    return True  # If a cycle is detected, return
True
            elif rec_stack[neighbor]:  # If the neighbor is already
in the recursion stack, a cycle is found
                return True

        rec_stack[node] = False  # Remove the node from the recursion
stack after DFS
        return False  # No cycle detected from this node
```

```
    def detect_deadlock(self):
        """Check if a deadlock exists by detecting cycles in the
graph."""
        visited = {node: False for node in self.graph}  # Track
visited nodes to prevent revisiting
        rec_stack = {node: False for node in self.graph}  # Track
nodes in the current DFS recursion stack

        # Iterate through all nodes in the graph
        for node in self.graph:
            if not visited[node]:  # If the node is not visited,
perform DFS from that node
                if self.is_cycle(visited, rec_stack, node):
                    return True  # Deadlock detected if a cycle is
found
        return False  # No deadlock if no cycles are found

rag = RAG()
print("Enter the edges for the Resource Allocation Graph:")
print("Format: P1 R1 (Process P1 requests Resource R1 or vice
versa)")
print("Type 'done' when finished.")
while True:
    edge = input("Enter edge (u v): ").strip()
    if edge.lower() == "done":
        break
    # Try to split the input into two parts (u and v) and add the
edge to the graph
    try:
        u, v = edge.split()
        rag.add_edge(u, v)  # Add an edge from u to v in the graph
    except ValueError:
        print("Invalid format. Please enter in the form 'u v'.")
if rag.detect_deadlock():
    print("Deadlock Detected!")
else:
    print("No Deadlock Detected.")
```

WFG:

```
from collections import defaultdict
class WFG:
    def __init__(self):
        # Initialize the graph using defaultdict of lists to store edges.
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        """Add a directed edge from process u to process v.
        This means process u is waiting for process v to release a
resource."""
        self.graph[u].append(v)  # Add v to the adjacency list of u

    def is_cycle(self, visited, rec_stack, node):
        """Detects cycles using Depth-First Search (DFS)."""
        visited[node] = True  # Mark the current node as visited
        rec_stack[node] = True  # Mark the node in the recursion stack to
track the DFS path
        # Recurse for all neighbors of the current node
        for neighbor in self.graph[node]:
            if not visited[neighbor]:  # If the neighbor is not visited,
continue DFS
                if self.is_cycle(visited, rec_stack, neighbor):
                    return True  # If a cycle is detected, return True
            elif rec_stack[neighbor]:  # If the neighbor is already in the
recursion stack, a cycle is found
                return True
        rec_stack[node] = False  # Remove the node from the recursion
stack after DFS
        return False  # No cycle detected from this node

    def detect_deadlock(self):
        """Check if a deadlock exists by detecting cycles in the graph."""
        visited = {node: False for node in self.graph}  # Track visited
nodes to prevent revisiting
        rec_stack = {node: False for node in self.graph}  # Track nodes in
the current DFS recursion stack
        # Iterate through all nodes in the graph
        for node in self.graph:
```

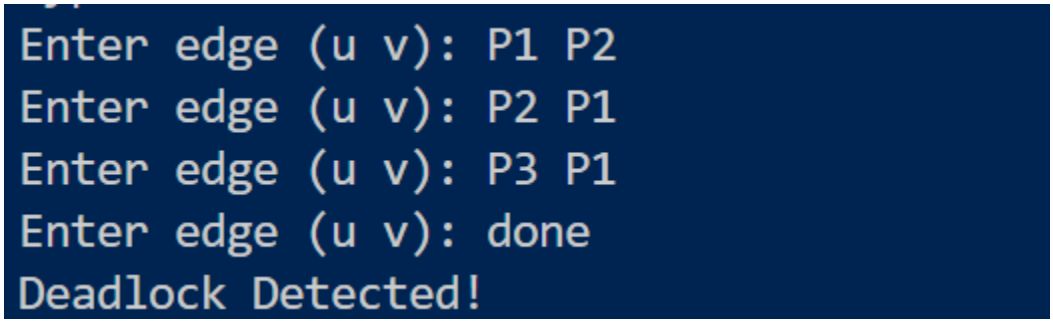
```
            if not visited[node]:  # If the node is not visited, perform
DFS from that node
                if self.is_cycle(visited, rec_stack, node):
                    return True  # Deadlock detected if a cycle is found
        return False  # No deadlock if no cycles are found

wfg = WFG()
print("Enter the edges for the Wait-for Graph:")
print("Format: P1 P2 (Process P1 is waiting for Process P2)")
print("Type 'done' when finished.")
while True:
    edge = input("Enter edge (u v): ").strip()
    if edge.lower() == "done":
        break
    # Try to split the input into two parts (u and v) and add the edge to
the graph
    try:
        u, v = edge.split()
        wfg.add_edge(u, v)  # Add an edge from u to v in the graph
    except ValueError:
        print("Invalid format. Please enter in the form 'u v'.")

if wfg.detect_deadlock():
    print("Deadlock Detected!")
else:
    print("No Deadlock Detected.")
```

OUTPUT:

RAG:

```
Enter edge (u v): P1 R2
Enter edge (u v): R2 P2
Enter edge (u v): P2 R1
Enter edge (u v): R1 P1
Enter edge (u v): P3 R1
Enter edge (u v): done
Deadlock Detected!
```

WFG:



CONCLUSION:

Both Resource Allocation Graphs (RAG) and Wait-for Graphs (WFG) are powerful tools for detecting deadlock in operating systems, but they each serve different needs. RAG is comprehensive and provides a complete picture of the resource management system, while WFG simplifies the deadlock detection process by focusing only on processes and their dependencies. In practice, WFG is typically preferred for deadlock detection because of its efficiency and simplicity, especially when dealing with large systems. However, for detailed analysis, particularly in resource-heavy environments, RAG may be more useful as it provides deeper insights into both resource allocation and request patterns.