

VIDHI ROHIRA
S.Y B.TECH
SEM III
COMPUTER ENGINEERING

OS LAB 7

231071052

BATCH – C

LABORATORY 7

AIM:- Implement an algorithm for deadlock detection.

THEORY:-

A **deadlock** is a situation in a multiprogramming environment where a group of processes becomes blocked because each process is waiting for a resource that is held by another process. Essentially, each process is stuck in a state where it is waiting on others, and no progress can be made.

Conditions for Deadlock (Coffman's Conditions)

For deadlock to occur, the following four conditions must be present simultaneously (these are known as **Coffman's Conditions**):

1. Mutual Exclusion:

- At least one resource is held in a non-shareable mode. This means that only one process can use a resource at any given time.

2. Hold and Wait:

- A process is holding at least one resource and is waiting to acquire additional resources that are being held by other processes.

3. No Preemption:

- Resources cannot be forcibly removed from the processes holding them. A process must release its resources voluntarily when it is done using them.

4. Circular Wait:

- A set of processes exists in which each process is waiting for a resource held by the next process in the set, forming a circular chain of dependencies.

Problems in Deadlock Detection

While deadlock detection is important to prevent or resolve deadlocks, there are several challenges associated with it:

1. Complexity:

- Deadlock detection algorithms can be **computationally expensive**. As systems scale with more processes and resources, detecting deadlocks can require significant processing power.

2. State Explosion:

- In large systems with many processes and resources, the number of possible states grows exponentially. This phenomenon, known as the **state explosion problem**, makes it difficult to track and analyze all possible interactions between processes and resources.

3. False Positives:

- Detection algorithms may occasionally **incorrectly identify deadlocks**, especially in systems where processes are executing slowly but are still making progress. These false positives can lead to unnecessary intervention and resource reallocation.

4. Overheads:

- Performing frequent checks for deadlocks introduces additional **overheads**. These checks consume valuable processing power and can slow down system performance, particularly if done too frequently or in large systems.

We mainly use **two algorithms in deadlock detection** which are:

- 1] **RAG** (Resource Allocation Graph)
- 2] **WFG** (Wait for Graph)

Resource Allocation Graph (RAG)

A **Resource Allocation Graph (RAG)** is a graphical representation used to model the allocation of resources to processes in a system. It is a powerful tool for analyzing and detecting **deadlocks** in systems. In a RAG, resources and processes are represented as nodes, and the relationships between them are depicted using directed edges.

Structure of a Resource Allocation Graph

A **RAG** is essentially a **bipartite graph** with two types of nodes and two types of directed edges:

- **Nodes:**
 1. **Process Nodes:** Represent processes in the system.
 2. **Resource Nodes:** Represent resources in the system.
- **Edges:**
 1. **Request Edges:** Directed edges from a process to a resource, indicating that the process is requesting a resource.
 2. **Allocation Edges:** Directed edges from a resource to a process, indicating that the resource is allocated to the process.

Advantages of Using RAG for Deadlock Detection

1. **Visualization:**
 - RAG provides an intuitive graphical representation of the system's state, allowing system administrators to easily identify deadlocks by detecting cycles in the graph.
2. **Granularity:**
 - It offers detailed information about the allocation of resources and the requests made by processes. This can help identify exactly where the system is bottlenecked.
3. **Versatility:**
 - RAGs can model systems with both **single** and **multiple instances** of resources, making them flexible for different types of resource allocation systems.

Limitations of RAG

1. Complexity for Large Systems:

- As the number of processes and resources increases, the RAG becomes more complex and harder to manage. The graph's size and intricacy can hinder its effective use in large-scale systems.

2. Multiple Instances of Resources:

- In systems with multiple instances of resources, the RAG needs to account for additional factors, making deadlock detection more complicated. Special handling is required to verify whether deadlock truly exists, as the graph can no longer simply rely on the presence of a cycle.

ALGORITHM

Input:

- **List of processes and resources:** Defines the set of processes and resources in the system.
- **Edges indicating requests and allocations:** Defines the current resource allocation and process request relationships.

Steps for Deadlock Detection:

1. Construct the RAG:

- Add **nodes** for all processes and resources.
- Add **edges** based on the requests and allocations made by the processes:
 - A **request edge** from a process to a resource if the process is requesting that resource.
 - An **allocation edge** from a resource to a process if the resource is allocated to the process.

2. Cycle Detection:

- Use **Depth-First Search (DFS)** to traverse the graph and look for cycles.
- Maintain a **stack** to track the visited nodes during the traversal.
- If a node is revisited while still in the stack, a cycle is detected. This indicates the potential for a deadlock, as processes are waiting on each other in a circular manner.

3. Determine Deadlock:

- If a cycle is detected:
 - **For single-instance resources:** A cycle directly implies a **deadlock** since there is only one instance of the resource, and the processes are mutually blocking each other.
 - **For multiple-instance resources:** In this case, a cycle does not necessarily imply deadlock. Additional checks must be made to verify if resources are available for allocation to break the cycle. If resources are available to allow progress, then no deadlock exists.

CODE:-

```
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."""

        # Ensure that all nodes (both processes and resources) are initialized
        in visited and rec_stack
        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
```

```

        # To include all nodes (both sources and targets), we need to ensure
        that even isolated nodes are initialized
        all_nodes = set(self.graph.keys()) # Processes with outgoing edges
        for neighbors in self.graph.values():
            all_nodes.update(neighbors) # Add all processes that are waiting
            for resources (target nodes)

        # Initialize visited and rec_stack for all nodes
        for node in all_nodes:
            if node not in visited:
                visited[node] = False
            if node not in rec_stack:
                rec_stack[node] = False

        # Iterate through all nodes in the graph and perform DFS if not
        visited
        for node in all_nodes:
            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

# Main execution: Get user input to create the Resource Allocation Graph and
detect deadlock
rag = RAG()

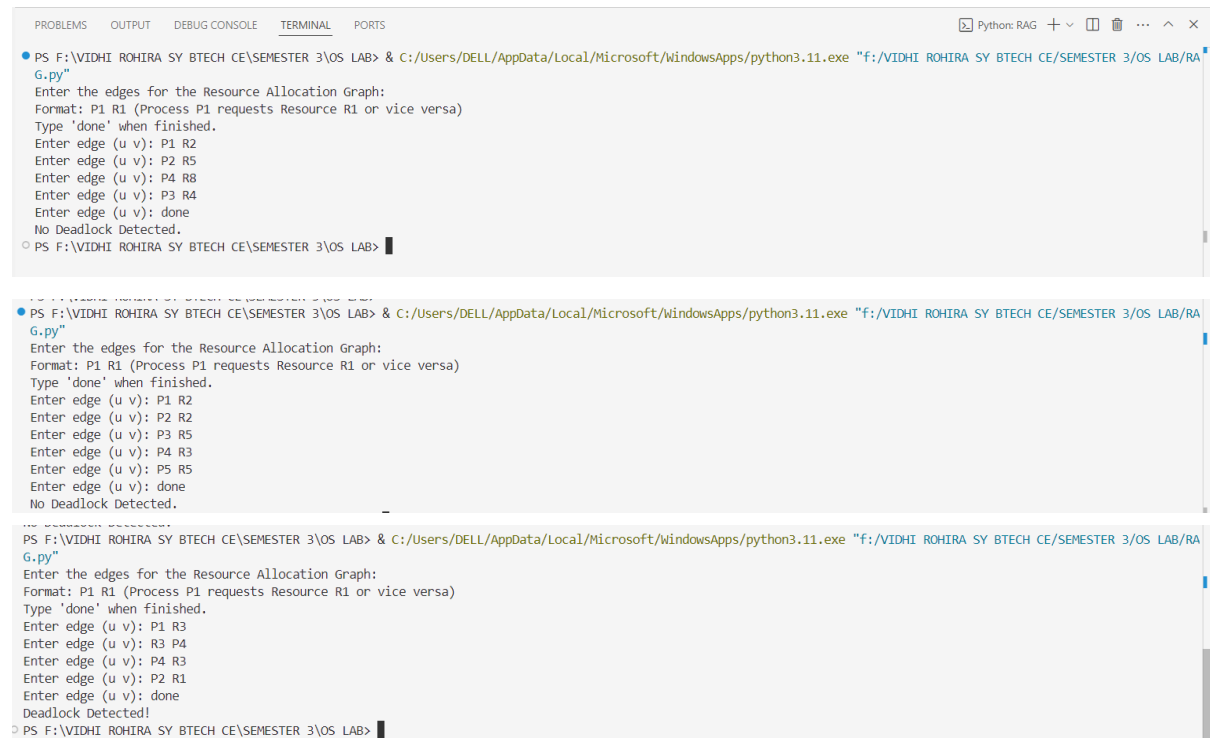
print("Enter the edges for the Resource Allocation Graph:")
print("Format: P1 R1 (Process P1 requests Resource R1 or vice versa)")
print("Type 'done' when finished.")

while True:
    edge = input("Enter edge (u v): ").strip()
    if edge.lower() == "done":
        break
    try:
        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'.")

# Detect deadlock
if rag.detect_deadlock():
    print("Deadlock Detected!")
else:
    print("No Deadlock Detected.")

```


OUTPUT:-



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python: RAG + - [ ] ... ^ x
• PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/RAG.py"
Enter the edges for the Resource Allocation Graph:
Format: P1 R1 (Process P1 requests Resource R1 or vice versa)
Type 'done' when finished.
Enter edge (u v): P1 R2
Enter edge (u v): P2 R5
Enter edge (u v): P4 R8
Enter edge (u v): P3 R4
Enter edge (u v): done
No Deadlock Detected.
○ PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>

• PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/RAG.py"
Enter the edges for the Resource Allocation Graph:
Format: P1 R1 (Process P1 requests Resource R1 or vice versa)
Type 'done' when finished.
Enter edge (u v): P1 R2
Enter edge (u v): P2 R2
Enter edge (u v): P3 R5
Enter edge (u v): P4 R3
Enter edge (u v): P5 R5
Enter edge (u v): done
No Deadlock Detected.
○ PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>

• PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/RAG.py"
Enter the edges for the Resource Allocation Graph:
Format: P1 R1 (Process P1 requests Resource R1 or vice versa)
Type 'done' when finished.
Enter edge (u v): P1 R3
Enter edge (u v): R3 P4
Enter edge (u v): P4 R3
Enter edge (u v): P2 R1
Enter edge (u v): done
Deadlock Detected!
○ PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

Wait-for Graph (WFG)

A **Wait-for Graph (WFG)** is a simplified version of the **Resource Allocation Graph (RAG)** used specifically for detecting deadlocks. In the WFG:

- **Nodes** represent processes (e.g., P1, P2, ..., Pn).
- **Edges** represent "wait-for" relationships: $P_i \rightarrow P_j$ means **Process P_i is waiting for a resource held by Process P_j.**

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

Advantages of WFG:

1. Simplified Representation:

- Only processes and their dependencies are shown.
- Easier to understand and analyze compared to the more complex RAG.

2. Direct Cycle Detection:

- A cycle in the graph directly indicates a deadlock, eliminating additional checks.

3. Efficiency:

- The graph size is smaller compared to RAG, making traversal faster and more efficient.

ALGORITHM

Input:

- List of processes.
- Wait-for relationships as directed edges ($P_i \rightarrow P_j$).

Steps:

1. Construct the WFG:

- Represent processes as nodes.
- Add directed 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 a deadlock exists.

CODE:-

```
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."""
        # Initialize visited and rec_stack for all nodes in the graph
        visited = {node: False for node in self.graph} # Track visited nodes
        rec_stack = {node: False for node in self.graph} # Track nodes in the
        current DFS recursion stack

        # Ensure that all nodes mentioned in the graph (both u and v in edges)
        are initialized in visited and rec_stack
        all_nodes = set(self.graph.keys()) # Processes with outgoing edges
        for neighbors in self.graph.values():
            all_nodes.update(neighbors) # Add all processes that are waiting
            for others (in the edge list)
```

```

        # Initialize visited and rec_stack for all nodes, even if they have no
        outgoing edges
        for node in all_nodes:
            if node not in visited:
                visited[node] = False
            if node not in rec_stack:
                rec_stack[node] = False

        # Iterate through all nodes in the graph and perform DFS if not
        visited
        for node in all_nodes:
            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

# Create an instance of the WFG class and test
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:
        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:-

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python: WFG + - [ ] ... ^ x
● PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/windowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/WF
G.py"
Enter the edges for the Wait-for Graph:
Format: P1 P2 (Process P1 is waiting for Process P2)
Type 'done' when finished.
Enter edge (u v): P1 P2
Enter edge (u v): P2 P3
Enter edge (u v): P3 P4
Enter edge (u v): P4 P1
Enter edge (u v): done
Deadlock Detected!
○ PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> █
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python: WFG + - [ ] ... ^ x
● PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/windowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/WF
G.py"
Enter the edges for the Wait-for Graph:
Format: P1 P2 (Process P1 is waiting for Process P2)
Type 'done' when finished.
Enter edge (u v): P1 P2
Enter edge (u v): P2 P3
Enter edge (u v): P3 P4
Enter edge (u v): P4 P7
Enter edge (u v): done
No Deadlock Detected.
○ PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> █
```

CONCLUSION:-

Resource Allocation Graphs (RAG) and Wait-for Graphs (WFG) are both crucial for detecting deadlock in operating systems, but they serve different purposes. RAG offers a comprehensive representation of both processes and resources, showing how processes request and are allocated resources. While it provides detailed insights, it can be complex and less efficient for large systems. On the other hand, WFG simplifies deadlock detection by focusing only on the relationships between processes, making it more efficient and scalable, especially in large systems. WFG is typically preferred for deadlock detection due to its simplicity, while RAG is better suited for detailed analysis in resource-heavy environments. Thus, WFG is more efficient for large-scale deadlock detection, while RAG provides a deeper understanding of resource and process interactions.