DEADLOCK DETECTION

Harsha Vardhan Reddy
Vamsi Krishna

Project Overview

Objective:

Implement a program to detect and resolve deadlocks in a Resource Allocation Graph (RAG).

Key Features:

Deadlock detection using cycle detection.
Deadlock resolution via process termination.
Visual representation of RAG.
Reset and priority simulation features.

Resource Allocation Graph (RAG)

What is RAG?
A directed graph where:
Nodes represent processes (P) and resources (R).
Edges indicate resource allocation or requests.

Example Components:
Processes: P0, P1, P2
Resources: R0, R1, R2

Key Concept:
Allocation Edge: Resource assigned to a process (e.g., R0 → P0).
Request Edge: Process requesting a resource (e.g., P0 → R1).

Key Code Components

1. **Graph Functions:**
   - initGraph: Initialize the RAG with a given number of nodes.
   - addEdge: Add edges between nodes (process-resource or resource-process).
   - removeEdge: Remove edges when processes release resources.
   - displayRAG: Visualizes the RAG structure.

2. **Deadlock Detection Functions:**
   - detectCycle: Uses Depth-First Search (DFS) to find cycles.
   - detectCycleUtil: Helper function for cycle detection.

3. **Deadlock Resolution Functions:**
   - resolveDeadlock: Identifies processes in the cycle and terminates one to resolve deadlock.

RAG Creation Example

```
initGraph(&graph, totalNodes);
addEdge(&graph, 0, 3);  // P0 → R3
addEdge(&graph, 1, 4);  // P1 → R4
addEdge(&graph, 2, 5);  // P2 → R5
addEdge(&graph, 3, 1);  // R3 → P1
addEdge(&graph, 4, 2);  // R4 → P2
addEdge(&graph, 5, 0);  // R5 → P0
```

Resulting Graph Visualization:

```
 P0 ---> R3 <--- P1
   |               |
   |               |
  V               V
  R0 <--- P2 --->R5
```

Semaphore:

Wait and signal functions

```
/**
 * Function: wait
 * ----------------
 * Simulates the request of a resource by a process.
 * Adds an edge from the process to the resource in the resource allocation graph,
 * indicating that the process is waiting for the resource.
 */
void wait(int process, int resource) {
    // Add an edge from the process to the resource in the resource graph.
    // This represents that the process is now waiting for the resource.
    addEdge(&resourceGraph, process, resource);
    printf("Process P%d is requesting Resource R%d.\n", process, resource);

    // Check if adding this edge introduces a deadlock in the system.
    if (isDeadlock(&resourceGraph)) {
        // Notify that a deadlock has been detected.
        // This warns the user/system about potential problems due to this request.
        printf("Warning: Deadlock detected. Process P%d's request may cause issues.\n", process);
    }
}

/**
 * Function: signal
 * ----------------
 * Simulates the release of a resource by a process.
 * Removes the edge from the process to the resource in the resource allocation graph,
 * indicating that the resource is no longer held or needed by the process.
 */
void signal(int process, int resource) {
    // Remove the edge from the process to the resource in the resource graph.
    // This signifies that the process has released the resource.
```

Deadlock Detection :

Code :

```
// Function to check if there is a deadlock in the Resource Allocation Graph
(RAG).
int isDeadlock(Graph *g) {    // Iterate through all nodes in the graph to detect cycles.
    for (int i = 0; i < g->numNodes; i++) {        // If a cycle is detected starting from
node i, a deadlock exists.
        if (detectCycle(g, i))
            return 1;    }
    return 0; // No deadlock found.}
```

Logic :

1. Uses detectcycle to find cycles in the graph
2. If cycle exists deadlock is confirmed.

Deadlock Resolution

- **Process:** Select a process involved in the deadlock (e.g., P0).
- Remove edges associated with the terminated process.
- Free up resources to resolve the deadlock

```
nction to resolve deadlock by terminating a process and releasing its resources.
resolveDeadlock(Graph *g) {
rintf("Attempting to resolve deadlock...\n");
or (int i = 0; i < g->numNodes; i++) {
  // Check if a cycle (deadlock) exists starting from node `i`.
  if (detectCycle(g, i)) {
      printf("Process P%d is involved in deadlock.\n", i);
      printf("Terminating Process P%d to resolve deadlock.\n", i);

      // Release all resources held by Process P[i].
      for (int j = 0; j < g->numNodes; j++) {
          removeEdge(g, i, j);
      }

      printf("Deadlock resolved.\n");
      return; // Exit after resolving the deadlock.
  }

rintf("No deadlock to resolve.\n");
```

Priority Simulation

```
setPriority(0, 1);  // Set priority of P0 to 1
setPriority(1, 2);  // Set priority of P1 to 2
setPriority(2, 3);  // Set priority of P2 to 3
```

Purpose: Allows processes to have priorities to influence scheduling or deadlock resolution strategies.

OUTPUT

```
[vamsigudipudi@Vamsis-MacBook-Air OS % gcc -o deadlock_detection main.c graph.c semaphore.c deadlock_detection.
[
vamsigudipudi@Vamsis-MacBook-Air OS % ./deadlock_detection

Setting priority of Process P0 to 1
Setting priority of Process P1 to 2
Setting priority of Process P2 to 3
Simulating resource allocation scenario...
Resource Allocation Graph created.
Resource Allocation Graph:
Node 0 -> Node 3
Node 1 -> Node 4
Node 2 -> Node 5
Node 3 -> Node 1
Node 4 -> Node 2
Node 5 -> Node 0
Deadlock detected after allocations.
Attempting to resolve deadlock...
Process P0 is involved in deadlock.
Terminating Process P0 to resolve deadlock.
Deadlock resolved.
Resetting Resource Allocation Graph...
Graph reset successfully.
vamsigudipudi@Vamsis-MacBook-Air OS % $
```

Here a RAG is drawn, a deadlock is detected , and the deadlock is resolved

Before and After Resolution

- Before Deadlock Resolution:

- ```
   P0 ---> R3 <--- P1
    |              |
    |              |
    V              V
   R0 <--- P2 ---> R5
  ```

- After Deadlock Resolution:

- ```
   P1 ---> R4 <--- P2
    |              |
    |              |
    V              V
   R1 <--- P2 ---> R5
  ```

Conclusion :

Key Takeaways:

- Deadlock detection is crucial for resource allocation in concurrent systems.
- The RAG model simplifies detection and resolution.
- Simulations allow testing under various scenarios.

Thank You