UNIVERSITI MALAYA

WIA2004 Operating Systems

Lab 7: Deadlock Management

Group： F5U

Occ: 10

Lecturer: Dr. Fazidah Othman

GROUP MEMBERS	MATRICS NUMBER
ZHANG ZHIYANG	S2193685
MEILIN LI	S2174975
HUMYRA TASMIA	S2176677
HUSNA NADIAH BINTI ROSTHAM	22060027
HUSNA BINTI IHSANUDDIN	U2102305

Table of Content

1.0 Introduction

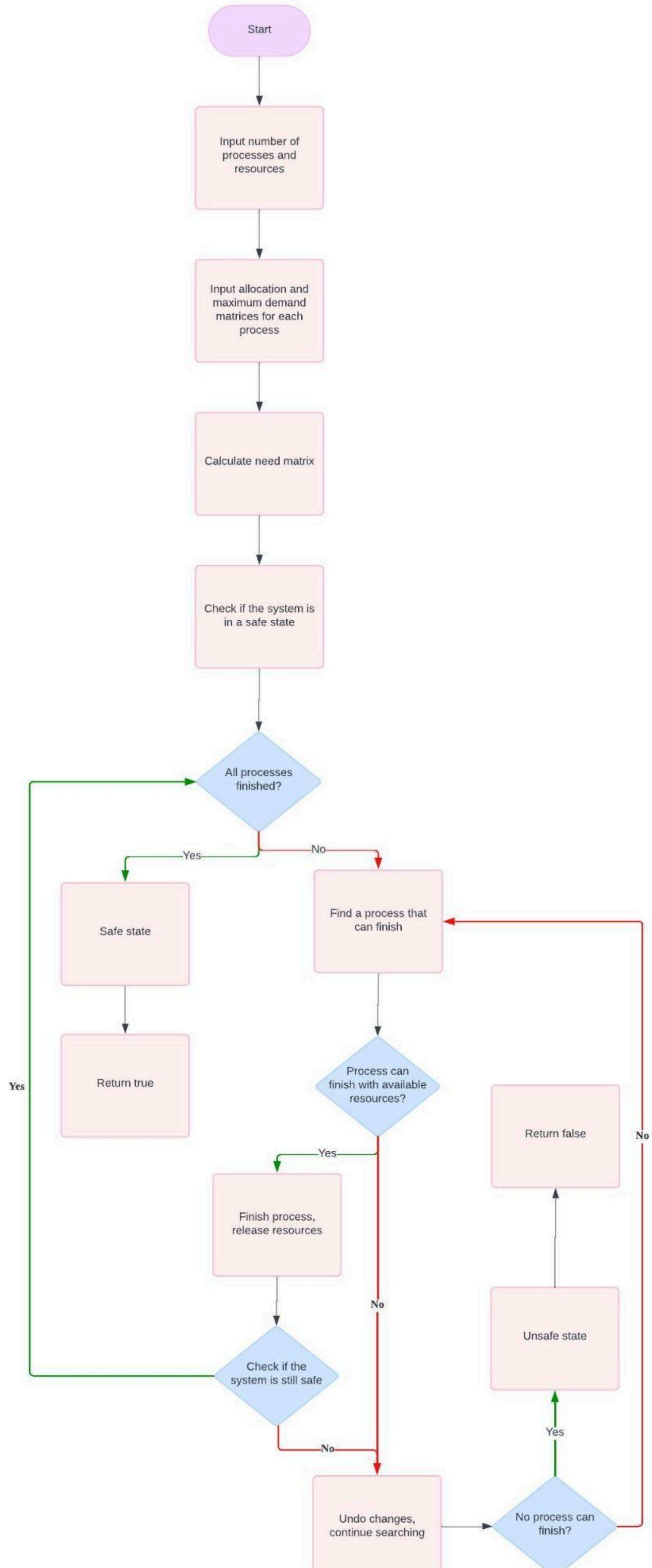
Deadlock is a situation in computing where a set of processes become stuck in a cycle of waiting for each other to release resources. Specifically, each process in the set is waiting for a resource that is currently held by another process in the same set, creating a scenario where none of the processes can proceed. The necessary conditions needed for deadlock include mutual exclusion, hold and wait, no preemption, and circular wait.

The Banker's Algorithm is a resource allocation and deadlock avoidance technique used in operating systems to ensure efficient sharing of resources among multiple processes. It operates by simulating the allocation of resources to processes while checking for safety at each step, ensuring that granting resource requests will not lead to deadlock. Named after banking practices, where loans are granted based on the availability of funds to ensure that all customers' needs can be met, the algorithm similarly ensures that the system remains in a "safe state" where processes can continue execution without the risk of deadlock.

Each process informs the system about its resource needs and how long it will use them. This helps the system allocate resources efficiently. The algorithm tracks the maximum resources each process can use, preventing deadlock. However, it has limitations: it works only with a fixed number of processes, doesn't allow processes to change their needs during execution, and requires processes to declare their maximum needs upfront. Additionally, resource allocation must occur within a set time limit, typically one year.

2.0 Methodology

Flowchart



Start
Input number of processes and resources
Input allocation and maximum demand matrices for each process
Calculate need matrix
Check if the system is in a safe state
All processes finished?
Yes
No
Safe state
Return true
Find a process that can finish
Process can finish with available resources?
Yes
No
Finish process, release resources
Check if the system is still safe
Yes
No
Undo changes, continue searching
No process can finish?
Yes
No
Unsafe state
Return false

Pseudocode

```
Class BankersAlgorithm:
    Integer numberOfProcesses
    Integer numberOfResources
    Array available
    Array maximum
    Array allocation
    Array need

    Constructor BankersAlgorithm(numberOfProcesses, numberOfResources):
        Initialize numberOfProcesses, numberOfResources
        Initialize available, maximum, allocation, need arrays

    Method inputMatrices(scanner):
        Print "Enter details for each process:"
        For each process i from 0 to numberOfProcesses:
            Print "P" + i + " Allocation: "
            Read allocation for process i from scanner
            For each resource j from 0 to numberOfResources:
                Set allocation[i][j] to allocation value for resource j
            Print "P" + i + " Max: "
            Read maximum claim for process i from scanner
            For each resource j from 0 to numberOfResources:
                Set maximum[i][j] to maximum claim value for resource j

        Print "Enter the Available Resources:"
        Read available resources from scanner
        For each resource i from 0 to numberOfResources:
            Set available[i] to available resource value for resource i

        Call calculateNeed()

    Method calculateNeed():
        For each process i from 0 to numberOfProcesses:
            For each resource j from 0 to numberOfResources:
                Set need[i][j] to maximum[i][j] - allocation[i][j]

    Method isSafeState():
        Initialize work as a copy of available
        Initialize finish as an array of boolean values of size numberOfProcesses,
all set to false
        Initialize safeSequence as an array of integers of size numberOfProcesses
        Initialize count to 0
```

```
        While count < numberOfProcesses:
            Set foundProcess to false
            For each process i from 0 to numberOfProcesses:
                If process i is not finished (finish[i] is false):
                    Set canProceed to true
                    For each resource j from 0 to numberOfResources:
                        If need[i][j] > work[j]:
                            Set canProceed to false
                            Break out of inner loop
                    If canProceed is true:
                        For each resource k from 0 to numberOfResources:
                            Increment work[k] by allocation[i][k]
                        Add process i to safeSequence at position count
                        Increment count by 1
                        Set finish[i] to true
                        Set foundProcess to true
                        Print "P" + i + " is visited: ("
                        For each resource k from 0 to numberOfResources:
                            Print work[k]
                            If k is not the last resource:
                                Print ","
                        Print ")"
            If no process is found:
                Print "Deadlock! The system is in an unsafe state."
                Return false

        Print "\nThe system is in a safe state."
        Print "The safe sequence is: "
        For each process i from 0 to numberOfProcesses:
            Print "P" + safeSequence[i]
            If i is not the last process:
                Print " -> "
        Print ""
        Return true

    Method requestResources(processID, request):
        For each resource i from 0 to numberOfResources:
            If request[i] > need[processID][i]:
                Print "Error: Process has exceeded its maximum claim."
                Return false
            If request[i] > available[i]:
                Print "Resources are not available."
                Return false

        For each resource i from 0 to numberOfResources:
            Decrement available[i] by request[i]
```

```
            Increment allocation[processID][i] by request[i]
            Decrement need[processID][i] by request[i]

        If isSafeState() returns false:
            Print "Request cannot be granted. Rolling back."
            For each resource i from 0 to numberOfResources:
                Increment available[i] by request[i]
                Decrement allocation[processID][i] by request[i]
                Increment need[processID][i] by request[i]
            Return false

        Return true

    Method printState():
        Print "Process\tAllocation\tMax\t\tNeed"
```

3.0 Implementation

3.1 Coding

```java
import java.util.Scanner;

public class Lab_7 {
    private int numberOfProcesses;
    private int numberOfResources;
    private int[] available;
    private int[][] maximum;
    private int[][] allocation;
    private int[][] need;

    public Lab_7(int numberOfProcesses, int numberOfResources) {
        this.numberOfProcesses = numberOfProcesses;
        this.numberOfResources = numberOfResources;
        this.available = new int[numberOfResources];
        this.maximum = new int[numberOfProcesses][numberOfResources];
        this.allocation = new int[numberOfProcesses][numberOfResources];
        this.need = new int[numberOfProcesses][numberOfResources];
    }
```

```java
//enter allocation matric & max matrix
public void inputMatrices(Scanner scanner) {
    System.out.println("Enter details for each process:");
    for (int i = 0; i < numberOfProcesses; i++) {
        System.out.println("P" + i + " Allocation: ");
        String[] alloc = scanner.nextLine().split(",");
        for (int j = 0; j < numberOfResources; j++) {
            allocation[i][j] = Integer.parseInt(alloc[j]);
        }
        System.out.println("P" + i + " Max: \t");
        String[] max = scanner.nextLine().split(",");
        for (int j = 0; j < numberOfResources; j++) {
            maximum[i][j] = Integer.parseInt(max[j]);
        }
    }

    System.out.println("Enter the Available Resources:");
    String[] avail = scanner.nextLine().split(",");
    for (int i = 0; i < numberOfResources; i++) {
        available[i] = Integer.parseInt(avail[i]);
    }

    calculateNeed();
}
```

```java
//need matrix = max matrix - allocation matrix
private void calculateNeed() {
    for (int i = 0; i < numberOfProcesses; i++) {
        for (int j = 0; j < numberOfResources; j++) {
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }
}
```

```java
public boolean isSafeState() {
    int[] work = available.clone();
    boolean[] finish = new boolean[numberOfProcesses];
    int[] safeSequence = new int[numberOfProcesses];
    int count = 0;

    while (count < numberOfProcesses) {
        boolean foundProcess = false;
        for (int i = 0; i < numberOfProcesses; i++) {
            if (!finish[i]) {
                boolean canProceed = true;
                for (int j = 0; j < numberOfResources; j++) {
                    if (need[i][j] > work[j]) {
                        canProceed = false;
                        break;
                    }
                }
                if (canProceed) {
                    for (int k = 0; k < numberOfResources; k++) {
                        work[k] += allocation[i][k];
                    }
                    safeSequence[count++] = i;
                    finish[i] = true;
                    foundProcess = true;
                    System.out.print("P" + i + " is visited:\t(");
                    for (int k = 0; k < numberOfResources; k++) {
                        System.out.print(work[k]);
                        if (k != numberOfResources - 1) System.out.print(",");
                    }
                    System.out.println(")");
                }
```

```java
                    System.out.println(")");
                }
            }
        }
        if (!foundProcess) {
            System.out.println("Deadlock! The system is in an unsafe state.");
            return false;
        }
    }

    System.out.println("\nThe system is in a safe state.");
    System.out.print("The safe sequence is: ");
    for (int i = 0; i < numberOfProcesses; i++) {
        System.out.print("P" + safeSequence[i]);
        if (i != numberOfProcesses - 1) System.out.print(" -> ");
    }
    System.out.println();
    return true;
```

```java
public boolean requestResources(int processID, int[] request) {
    for (int i = 0; i < numberOfResources; i++) {
        if (request[i] > need[processID][i]) {
            System.out.println("Error: Process has exceeded its maximum claim.");
            return false;
        }
        if (request[i] > available[i]) {
            System.out.println("Resources are not available.");
            return false;
        }
    }

    for (int i = 0; i < numberOfResources; i++) {
        available[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }

    if (!isSafeState()) {
        System.out.println("Request cannot be granted. Rolling back.");
        for (int i = 0; i < numberOfResources; i++) {
            available[i] += request[i];
            allocation[processID][i] -= request[i];
            need[processID][i] += request[i];
        }
        return false;
    }

    return true;
}
```

```java
public void printState() {
    System.out.println("Process\tAllocation\tMax\t\tNeed");
    for (int i = 0; i < numberOfProcesses; i++) {
        System.out.print("P" + i + "\t");
        for (int j = 0; j < numberOfResources; j++) {
            System.out.print(allocation[i][j] + (j == numberOfResources - 1 ? "" : ","));
        }
        System.out.print("\t\t");
        for (int j = 0; j < numberOfResources; j++) {
            System.out.print(maximum[i][j] + (j == numberOfResources - 1 ? "" : ","));
        }
        System.out.print("\t\t");
        for (int j = 0; j < numberOfResources; j++) {
            System.out.print(need[i][j] + (j == numberOfResources - 1 ? "" : ","));
        }
        System.out.println();
    }
}
```

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter number of processes:");
    int numberOfProcesses = scanner.nextInt();
    scanner.nextLine();

    System.out.println("Enter number of resources:");
    int numberOfResources = scanner.nextInt();
    scanner.nextLine();  // Consume the newline character

    Lab_7 bankersAlgorithm = new Lab_7(numberOfProcesses, numberOfResources);
    bankersAlgorithm.inputMatrices(scanner);

    System.out.println("Enter the process number(PID) making the request:");
    int processID = scanner.nextInt();
    scanner.nextLine();  // Consume the newline character

    System.out.println("Enter the requested resources:");
    String[] requestInput = scanner.nextLine().split(",");
    int[] request = new int[numberOfResources];
    try {
        for (int i = 0; i < numberOfResources; i++) {
            request[i] = Integer.parseInt(requestInput[i]);
        }
    } catch (Exception e) {
        System.out.println("Error reading input for requested resources: " + e.getMessage());
        e.printStackTrace();
        scanner.close();
        return;
    }
```

```java
        if (bankersAlgorithm.requestResources(processID, request)) {
            System.out.println("The request has been granted.");
        } else {
            System.out.println("The request has been denied.");
        }

        bankersAlgorithm.printState();
        scanner.close();
    }
}
```

3.2 Sample output

3.2.1 **No deadlock**

Allocation matrix:

P0: [0, 1, 0]
P1: [2, 0, 0]
P2: [3, 0, 2]
P3: [2, 1, 1]
P4: [0, 0, 2]

Max matrix:

P0: [7, 5, 3]
P1: [3, 2, 2]
P2: [9, 0, 2]
P3: [2, 2, 2]
P4: [4, 3, 3]

Need matrix:

P0: [7, 4, 3]
P1: [1, 2, 2]
P2: [6, 0, 0]
P3: [0, 1, 1]
P4: [4, 3, 1]

User input:

```
Enter number of processes:
5
Enter number of resources:
3
Enter details for each process:
P0 Allocation:
0,1,0
P0 Max:
7,5,3
P1 Allocation:
2,0,0
P1 Max:
3,2,2
P2 Allocation:
3,0,2
P2 Max:
9,0,2
P3 Allocation:
2,1,1
P3 Max:
2,2,2
P4 Allocation:
0,0,2
P4 Max:
4,3,3
Enter the Available Resources:
3,3,2
Enter the process number(PID) making the request:
1
Enter the requested resources:
1,0,2
```

Step 1: Check the Request

- Process (PID): 1
- Request: (1, 0, 2)

Step 2: Allocate Resources

- Update Available Resources: [3-1, 3-0, 2-2] = [2, 3, 0]
- Update Allocation for P1(Allocation[P1] + Request): [2+1, 0+0, 0+2] = [3, 0, 2]
- Update Need for P1(Need[P1] - Request): [1-1, 2-0, 2-2] = [0, 2, 0]

Step 3: Check for Safety

Initialize:

- Work Array: [2, 3, 0]
- Finish Array: [false, false, false, false, false]
- Safe Sequence: []

Iterate through each process to check if it can proceed:

1. Check Process P0:
    - Need[P0] is [7, 4, 3], but Work is [2, 3, 0], so P0 cannot proceed.
    - Break; i++
2. Check Process P1:
    - Need[P1] is [0, 2, 0], and Work is [2, 3, 0], so P1 can proceed.
    - work[] =work[] + allocation[1] = [2,3,0] + [3,0,2] = [5,3,2]
    - i++
3. Check Process P2:
    - Need[P2] is [6, 0, 0], but Work is [5, 3, 2], so P2 cannot proceed.
    - Break; i++
4. Check Process P3:
    - Need[P3] is [0, 1, 1], and Work is [5, 3, 2], so P3 can proceed.
    - work[] =work[] + allocation[3] = [5,3,2] + [2,1,1] = [7,4,3]
    - i++
5. Check Process P4:
    - Need[P4] is [4, 3, 1], and Work is [2, 3, 0], so P4 can proceed.
    - work[] =work[] + allocation[4] = [7,4,3] + [0,0,2] = [7,4,5]
    - i++

6. Finish Array: [false, true, false, true, true]

   ->not all element are true(count<numberOfProcess)

   ->while()loop keep executing

7. Check Process P0:
   - Need[P0] is [7, 4, 3], and Work is [7, 4, 5], so P0 can proceed.
   - work[] =work[] + allocation[0] = [7,4,5] + [0,1,0] = [7,5,5]
   - Break; i++
8. Check Process P2:
   - Need[P2] is [6, 0, 0], and Work is [7, 5, 5], so P2 can proceed.
   - work[] =work[] + allocation[2] = [7,5,5] + [3,0,2] = [10,5,7]
   - Break; i++
9. Finish Array: [true, true, true, true, true]

   ->all element are true(count !< numberOfProcess)

   ->while()loop step executing

10. Safe state find,no deadlock

Sample output:

```
P1 is visited:  (5,3,2)
P3 is visited:  (7,4,3)
P4 is visited:  (7,4,5)
P0 is visited:  (7,5,5)
P2 is visited:  (10,5,7)

The system is in a safe state.
The safe sequence is: P1 -> P3 -> P4 -> P0 -> P2
The request has been granted.
Process Allocation      Max             Need
P0      0,1,0           7,5,3           7,4,3
P1      3,0,2           3,2,2           0,2,0
P2      3,0,2           9,0,2           6,0,0
P3      2,1,1           2,2,2           0,1,1
P4      0,0,2           4,3,3           4,3,1
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  47.649 s
Finished at: 2024-05-14T15:32:47+08:00
------------------------------------------------------------------------
```

3.2.2 **Exist deadlock:**

Allocation matrix:

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max matrix:

A	B	C	
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Need matrix:

A	B	C	
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

User input:

```
Enter number of processes:
5
Enter number of resources:
3
Enter details for each process:
P0 Allocation:
0,1,0
P0 Max:
7,5,3
P1 Allocation:
2,0,0
P1 Max:
3,2,2
P2 Allocation:
3,0,2
P2 Max:
9,0,2
P3 Allocation:
2,1,1
P3 Max:
2,2,2
P4 Allocation:
0,0,2
P4 Max:
4,3,3
Enter the Available Resources:
1,1,0
Enter the process number(PID) making the request:
1
Enter the requested resources:
1,0,0
```

Step 1: Check the Request

- Process (PID): 1
- Request: (1, 0, 0)

Step 2: Allocate Resources

- Update Available Resources: [1-1, 1-0, 0-0] = [0, 1, 0]
- Update Allocation for P1: [2+1, 0+0, 0+0] = [3, 0, 0]
- Update Need for P1: [1-1, 2-0, 2-0] = [0, 2, 2]

Step 3: Check for Safety

Initialize:

- Work Array: [0, 1, 0]
- Finish Array: [false, false, false, false, false]
- Safe Sequence: []

Iterate through each process to check if it can proceed:

11. Check Process P0:
    - Need[P0] is [7, 4, 3], but Work is [0, 1, 0], so P0 cannot proceed.
12. Check Process P1:
    - Need[P1] is [1, 2, 2], but Work is [0, 1, 0], so P1 cannot proceed.
13. Check Process P2:
    - Need[P2] is [6, 0, 0], but Work is [0, 1, 0], so P2 cannot proceed.
14. Check Process P3:
    - Need[P3] is [0, 1, 1], but Work is [0, 1, 0], so P3 cannot proceed.
15. Check Process P4:
    - Need[P4] is [4, 3, 1], but Work is [0, 1, 0], so P4 cannot proceed.

Deadlock Result

Since none of the processes can proceed, the system is in a deadlock state. The system will output that no safe sequence was found and declare the system to be in an unsafe state:

Sample output:

```
Deadlock! The system is in an unsafe state.
Request cannot be granted. Rolling back.
The request has been denied.
Process Allocation      Max             Need
P0      0,1,0           7,5,3           7,4,3
P1      2,0,0           3,2,2           1,2,2
P2      3,0,2           9,0,2           6,0,0
P3      2,1,1           2,2,2           0,1,1
P4      0,0,2           4,3,3           4,3,1
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  53.513 s
Finished at: 2024-05-14T21:39:09+08:00
------------------------------------------------------------------------
```

4.0 Conclusion

The Banker's algorithm is a valuable tool for ensuring safe resource allocation in systems with multiple competing requests. It prevents deadlocks by simulating resource allocation scenarios and guaranteeing that the system can reach a state where all processes can finish their tasks. In essence, it prioritizes system stability by ensuring sufficient resources are always available to meet future demands.

However, the Banker's algorithm does have limitations. It requires advanced knowledge of the maximum resource needs for each process, which may not always be practical. Additionally, it doesn't allow for dynamic changes in resource requirements or the addition of new processes mid-execution.

Despite these limitations, the Banker's algorithm remains a crucial concept for understanding resource allocation and deadlock avoidance in various systems, including operating systems and banking environments.

5.0 References

- Banker’s Algorithm in Operating System - javatpoint. (n.d.). Www.javatpoint.com. *https://www.javatpoint.com/bankers-algorithm-in-operating-system*
- OS Deadlocks Introduction - javatpoint. (2011). Www.javatpoint.com. *https://www.javatpoint.com/os-deadlocks-introduction*
- Banker’s Algorithm in Operating System. (2018, January 4). GeeksforGeeks. *https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system/*