C

**Batch: C1 Roll No.: 16010122221**

**Experiment No. 08**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **TITLE:** Implementation of Deadlock Avoidance Policy. |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 3.** To understand the concepts of process synchronization and deadlock.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems”, McGraw Hill Third Edition.**
3. **Sumitabha Das “ UNIX Concepts & Applications”, McGraw Hill Second**

**Edition.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Knowledge of deadlocks and all deadlock avoidance methods.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Description of the application to be implemented**:

The Banker's algorithm is a [resource allocation](http://en.wikipedia.org/wiki/Resource_allocation) and [deadlock](http://en.wikipedia.org/wiki/Deadlock) avoidance [algorithm](http://en.wikipedia.org/wiki/Algorithm) developed by [Edsger Dijkstra.](http://en.wikipedia.org/wiki/Edsger_Dijkstra)

# DATA STRUCTURES

(where *n* is the number of processes in the system and *m* is the number of resource types) **Available:**

* It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
* Available[j] = k means there are **‘k’** instances of resource type **Rj Max:**
* It is a 2-d array of size ‘**n\*m’** that defines the maximum demand of each process in a system.
* Max [i, j] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj.**

**Allocation:**

* It is a 2-d array of size **‘n\*m’** that defines the number of resources of each type currently allocated to each process.
* Allocation [i, j] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj Need:**
* It is a 2-d array of size **‘n\*m’** that indicates the remaining resource need of each process.
* Need [i, j] = k means process **Pi** currently need **‘k’** instances of resource type **Rj**
* **Need [i, j] = Max [i, j] – Allocation [i, j]**

**Implementation details:**

n = int(input("Enter number of processes: "))

m = int(input("Enter number of resources: "))

alloc = []

print("Enter allocation matrix: ")

for i in range(n):

    temp = list(map(int, input().split()))

    alloc.append(temp)

max = []

print("\nEnter max matrix: ")

for i in range(n):

    temp = list(map(int, input().split()))

    max.append(temp)

avail = list(map(int, input("\nEnter available resources: ").split()))

f = [0] \* n

ans = [0] \* n

ind = 0

need = [[0 for i in range(m)] for i in range(n)]

for i in range(n):

    for j in range(m):

        need[i][j] = max[i][j] - alloc[i][j]

for y in range(m):

    for k in range(5):

        for i in range(n):

            if f[i] == 0:

                flag = 0

                for j in range(m):

                    if need[i][j] > avail[j]:

                        flag = 1

                        break

                if flag == 0:

                    ans[ind] = i

                    ind += 1

                    for y in range(m):

                        avail[y] += alloc[i][y]

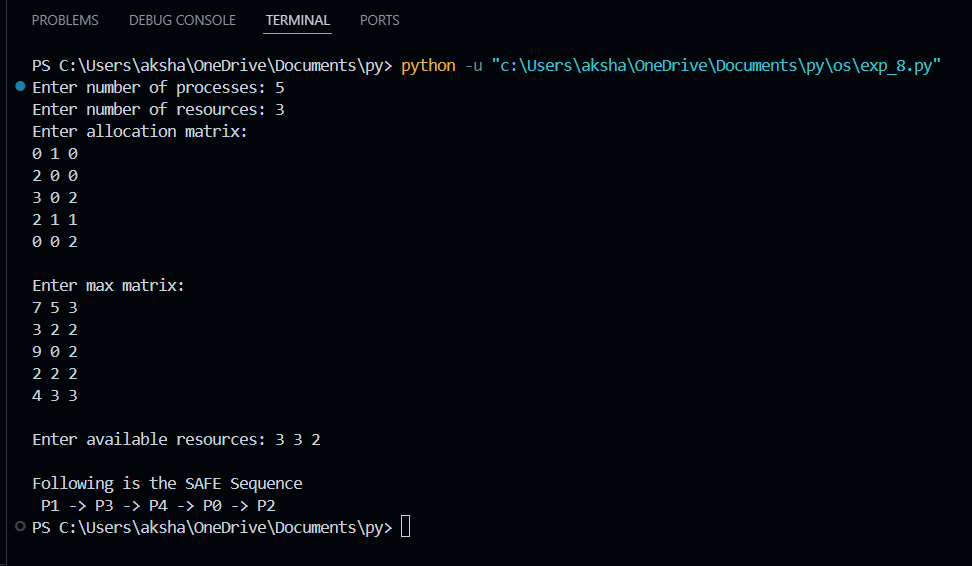
                    f[i] = 1

print("\nFollowing is the SAFE Sequence")

for i in range(n - 1):

    print(" P", ans[i], " ->", sep="", end="")

print(" P", ans[n - 1], sep="")



**Conclusion:**

In this experiment we learned and successfully implemented banker’s algorithm for resource allocation and deadlock avoidance.

**Post Lab Descriptive Questions**

1. Explain the concept of a “safe state” in the context of the Banker’s Algorithm. Why is it important for deadlock avoidance?

A **safe state** in the Banker's Algorithm is one in which there exists a sequence of processes (a safe sequence) such that each process can finish its execution even if all of them demand the maximum resources they have declared. In a safe state:

* There is a way to allocate resources such that every process can complete its task eventually without causing a deadlock.

**Importance for Deadlock Avoidance**: Ensuring the system stays in a safe state is critical because it guarantees that no set of processes can cause a circular wait, which is a key condition for deadlock. If a system enters an **unsafe state**, it could lead to deadlock, where no progress is possible due to inter-process resource dependency.

1. Describe the role of the need matrix in the Banker’s Algorithm. How is it calculated and used?

The **Need matrix** represents the additional resources that each process may need to complete its execution. It is calculated as:

Need[i][j]=Max[i][j]−Alloc[i][j]

Where:

* Max[i][j]: The maximum resources of type j that process i might need.
* Alloc[i][j]: The resources of type j that are currently allocated to process i.

**Usage**:

* The Need matrix is checked against the available resources to determine whether a process can be safely executed. A process can proceed if its resource needs (from the Need matrix) are less than or equal to the currently available resources.

1. What are the differences between deadlock prevention, avoidance, and detection? Provide examples of each.

· **Deadlock Prevention**: This strategy ensures that at least one of the four necessary conditions for deadlock (mutual exclusion, hold and wait, no preemption, and circular wait) is violated. Examples include:

* · **Mutual exclusion**: Restricting access to critical resources.
* **Circular wait prevention**: Enforcing a hierarchical ordering of resources.

· **Deadlock Avoidance**: The system dynamically checks whether allocating a requested resource could lead to a deadlock. The Banker's Algorithm is an example of deadlock avoidance, as it ensures the system stays in a safe state.

· · **Deadlock Detection**: In this approach, the system allows processes to request resources freely and periodically checks for deadlocks using algorithms like the **Resource Allocation Graph (RAG)** cycle detection. If a deadlock is detected, corrective actions like process termination or resource preemption are taken.

·

1. How does the Resource Allocation Graph (RAG) help in detecting potential deadlocks?

The **Resource Allocation Graph (RAG)** is a directed graph that represents the allocation of resources to processes. In a RAG:

* **Processes** are represented by circles, and **resources** by squares.
* An edge from a process to a resource represents a resource request, while an edge from a resource to a process represents a resource allocation.

**Deadlock Detection**:

* A deadlock is indicated by a **cycle** in the RAG. If there is a cycle and resources are non-preemptive, then the system is in a deadlock state.

1. In the Banker’s Algorithm, what steps are taken if a resource request leads to an unsafe state?

If granting a resource request leads to an unsafe state, the request is **denied**, and the process must wait. The steps are:

1. Simulate the allocation of requested resources.
2. Check if the resulting state is safe by ensuring there is a safe sequence for all processes.
3. If a safe sequence exists, grant the request. Otherwise, reject the request and leave the system unchanged.
4. Analyze the impact of resource allocation policies on system throughput and process starvation. How can these policies be optimized?

**System Throughput**: Resource allocation policies directly influence throughput by determining how quickly processes can complete their execution. A more **efficient** allocation policy maximizes throughput by ensuring that resources are allocated in a way that reduces idle time and increases process completion rates.

**Process Starvation**: Poorly designed policies can lead to starvation, where certain processes are perpetually denied resources due to the continued prioritization of others. For example, policies based solely on process priority can lead to **low-priority process starvation**.

**Optimization**:

* Use dynamic resource allocation and fair-share algorithms to avoid starvation and ensure better throughput. Algorithms like **fair scheduling** or **round-robin allocation** help maintain balance between high-throughput and fairness, reducing the chance of starvation.

1. Consider a system with total of 150 units of memory allocated to three processes as shown:

|  |  |  |
| --- | --- | --- |
| **Process** | **Max** | **Hold** |
| P1 | 70 | 45 |
| P2 | 60 | 40 |
| P3 | 60 | 15 |

Apply Banker’s algorithm to determine whether it would be safe to grant each of the following request. If yes, indicate sequence of termination that could be possible.

1. The P4 process arrives with max need of 60 and initial need of 25 units.
2. The P4 process arrives with max need of 60 and initial need of 35 units.

System details:

- Total memory: 150 units.

- Initial allocation (Hold matrix) and maximum needs (Max matrix) are given:

| Process | Max | Hold |

|--------|-----|------|

| P1 | 70 | 45 |

| P2 | 60 | 40 |

| P3 | 60 | 15 |

Case 1: P4 Process with Max Need of 60 and Initial Need of 25

- Initial available memory:

Total available = 150 - (45 + 40 + 15) = 50 units.

- P4's initial need is 25 units, and the available memory is 50 units. This request can be granted.

- Need matrix after granting P4's request (Max need = 60, Initial allocation = 25):

| Process | Max | Hold | Need |

|--------|-----|------|------|

| P1 | 70 | 45 | 25 |

| P2 | 60 | 40 | 20 |

| P3 | 60 | 15 | 45 |

| P4 | 60 | 25 | 35 |

- Safe sequence:

Possible sequence: P2 → P1 → P4 → P3.

Case 2: P4 Process with Max Need of 60 and Initial Need of 35

- Initial available memory:

Available memory = 50 units.

- P4's initial need is 35 units, and there are 50 units available. This request can also be granted.

- Need matrix after granting P4's request (Max need = 60, Initial allocation = 35):

| Process | Max | Hold | Need |

|--------|-----|------|------|

| P1 | 70 | 45 | 25 |

| P2 | 60 | 40 | 20 |

| P3 | 60 | 15 | 45 |

| P4 | 60 | 35 | 25 |

- Safe sequence:

Possible sequence: P2 → P1 → P4 → P3.

In both cases, it is safe to grant the request, and a valid sequence can be found.

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**