

| **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)

**Implementation details:**

#include <iostream>

#include <vector>

using namespace std;

bool canAllocateResources(int process, vector<vector<int>>& needMatrix, vector<int>& availableResources) {

for (int resource = 0; resource < needMatrix[0].size(); resource++) {

if (needMatrix[process][resource] > availableResources[resource]) {

return false;

}

}

return true;

}

void inputMatrix(int numProcesses, int numResources, vector<vector<int>>& matrix, const string& name) {

cout << "Enter the " << name << " matrix:\n";

for (int process = 0; process < numProcesses; process++) {

for (int resource = 0; resource < numResources; resource++) {

cin >> matrix[process][resource];

}

}

}

void printMatrix(int numProcesses, int numResources, vector<vector<int>>& matrix, const string& name) {

cout << name << " matrix:\n";

for (int process = 0; process < numProcesses; process++) {

for (int resource = 0; resource < numResources; resource++) {

cout << matrix[process][resource] << " ";

}

cout << endl;

}

}

void handleResourceRequest(int numResources, vector<vector<int>>& allocationMatrix, vector<vector<int>>& needMatrix, vector<int>& availableResources) {

int processID;

cout << "Enter resource request process number (0 indexed) or -1 if none: ";

cin >> processID;

if (processID != -1) {

vector<int> request(numResources);

cout << "Enter the request for resources: ";

for (int resource = 0; resource < numResources; resource++) {

cin >> request[resource];

}

for (int resource = 0; resource < numResources; resource++) {

if (availableResources[resource] < request[resource]) {

cout << "Resource " << resource << " is not enough" << endl;

return;

}

if (needMatrix[processID][resource] < request[resource]) {

cout << "Need for resource " << resource << " is lower than request" << endl;

return;

}

}

cout << "Request is valid and granted" << endl;

for (int resource = 0; resource < numResources; resource++) {

availableResources[resource] -= request[resource];

allocationMatrix[processID][resource] += request[resource];

needMatrix[processID][resource] -= request[resource];

}

printMatrix(allocationMatrix.size(), numResources, needMatrix, "Updated Need");

printMatrix(allocationMatrix.size(), numResources, allocationMatrix, "Updated Allocation");

cout << "Updated Available Resources: ";

for (int resource = 0; resource < numResources; resource++) {

cout << availableResources[resource] << " ";

}

cout << endl;

}

}

bool isSafeState(int numProcesses, int numResources, vector<vector<int>>& allocationMatrix, vector<vector<int>>& needMatrix, vector<int>& availableResources) {

vector<bool> completed(numProcesses, false);

int completedCount = 0;

while (completedCount != numProcesses) {

bool found = false;

for (int process = 0; process < numProcesses; process++) {

if (!completed[process] && canAllocateResources(process, needMatrix, availableResources)) {

completedCount++;

cout << "P" << process << " ";

for (int resource = 0; resource < numResources; resource++) {

availableResources[resource] += allocationMatrix[process][resource];

}

completed[process] = true;

found = true;

}

}

if (!found) {

cout << "System is not in a safe state." << endl;

return false;

}

}

cout << endl << "System is in a safe state." << endl;

return true;

}

int main() {

int numProcesses, numResources;

cout << "Enter the number of processes: ";

cin >> numProcesses;

cout << "Enter the number of resources: ";

cin >> numResources;

vector<vector<int>> allocationMatrix(numProcesses, vector<int>(numResources));

vector<vector<int>> maxMatrix(numProcesses, vector<int>(numResources));

vector<vector<int>> needMatrix(numProcesses, vector<int>(numResources));

vector<int> availableResources(numResources);

inputMatrix(numProcesses, numResources, allocationMatrix, "Allocation");

inputMatrix(numProcesses, numResources, maxMatrix, "Maximum");

for (int process = 0; process < numProcesses; process++) {

for (int resource = 0; resource < numResources; resource++) {

needMatrix[process][resource] = maxMatrix[process][resource] - allocationMatrix[process][resource];

}

}

printMatrix(numProcesses, numResources, needMatrix, "Need");

cout << "Enter the available resources: ";

for (int resource = 0; resource < numResources; resource++) {

cin >> availableResources[resource];

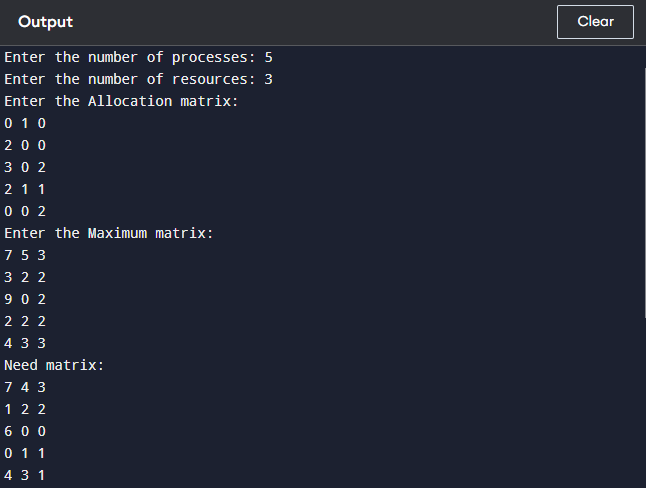
}

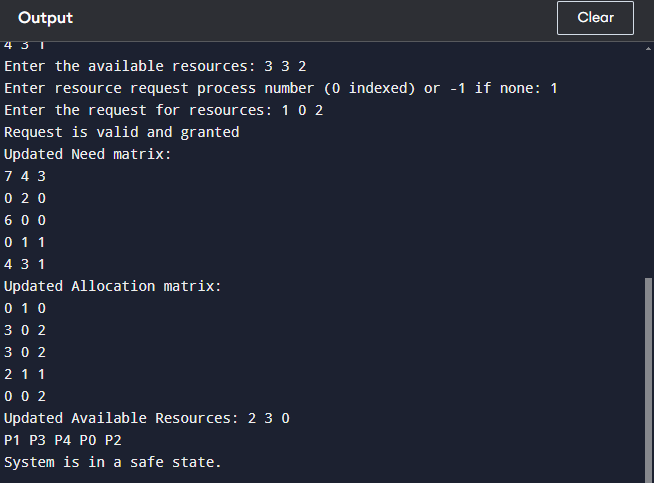
handleResourceRequest(numResources, allocationMatrix, needMatrix, availableResources);

isSafeState(numProcesses, numResources, allocationMatrix, needMatrix, availableResources);

return 0;

}





**Conclusion:**

**Learnt the implementation of Banker's Algorithm, ensuring deadlock avoidance through safe resource allocation management.**

**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" is a situation where the system can allocate resources to processes in such a way that each process can eventually complete. It is crucial for deadlock avoidance because in a safe state, there's always a sequence of processes that can be executed without causing deadlock.

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

The need matrix represents the remaining resources that each process needs to complete its task. It is calculated as:

This matrix helps in determining whether a process's request for resources can be safely granted.

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

**Prevention**: Proactively ensures that a deadlock cannot occur by eliminating one of the four necessary conditions for deadlock (e.g., by ensuring mutual exclusion).

**Avoidance**: Uses algorithms like the Banker's Algorithm to avoid unsafe states by checking resource requests.

**Detection**: Allows deadlocks to occur but periodically checks the system to detect and resolve deadlocks (e.g., by killing processes).

**Example**:

* **Prevention**: Circular wait prevention by ordering resource allocation.
* **Avoidance**: Banker's Algorithm.
* **Detection**: Using a wait-for graph.

1. **How does the Resource Allocation Graph (RAG) help in detecting potential deadlocks?**  
     
   The RAG helps in visualizing the resource allocation and requests in a system. A cycle in the graph indicates a potential deadlock. By detecting such cycles, RAG helps to identify deadlocks early.

1. **In the Banker’s Algorithm, what steps are taken if a resource request leads to an unsafe state?**  
     
   If a resource request leads to an unsafe state, the system denies the request. The system rolls back any changes made by the request and restores the resources to the previous state to avoid potential deadlock.

1. **Analyze the impact of resource allocation policies on system throughput and process starvation. How can these policies be optimized?**  
     
   Poor resource allocation policies can reduce throughput by keeping resources idle or causing deadlocks. Conversely, strict policies may cause process starvation, where some processes are indefinitely delayed. Optimizing policies involves balancing fairness and efficiency, for example, by using algorithms like round-robin scheduling or resource prioritization.

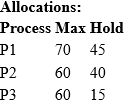
1. **Consider a system with total of 150 units of memory allocated to three processes as shown:**

| **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.**

**Total Memory:** 150 units



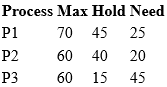
**1. Calculating Available Resources**

Available=Total−(Hold1​+Hold2​+Hold3​)

Available=150−(45+40+15)=150−100=50

**Available Resources:** 50 units

**2. Calculating the Need Matrix**

****

* **P1 Need:** 70−45=2570 - 45 = 2570−45=25
* **P2 Need:** 60−40=2060 - 40 = 2060−40=20
* **P3 Need:** 60−15=4560 - 15 = 4560−15=45

**3. Request by P4**

**1. Request (Max = 60, Need = 25)**

* **P4 requests:** 25 units
* **Available after allocation:**

50−25=2550 - 25 = 2550−25=25

* **Check if the system remains in a safe state:**
  + Can we satisfy the needs of P1, P2, or P3 with 25 available?
    - **P1:** Needs 25 (can finish) → Available = 25+45=7025 + 45 = 7025+45=70
    - **P2:** Needs 20 (can finish) → Available = 70+40=11070 + 40 = 11070+40=110
    - **P3:** Needs 45 (can finish)
* **Sequence of termination:** P1 → P2 → P3 → P4
* **Conclusion:** **Safe to grant.**

**2. Request (Max = 60, Need = 35)**

* **P4 requests:** 35 units
* **Available after allocation:**

50−35=1550 - 35 = 1550−35=15

* **Check safe state:**
  + With 15 available, no process can finish since:
    - **P1:** Needs 25
    - **P2:** Needs 20
    - **P3:** Needs 45
* **Conclusion:** **Not safe to grant.**

Therefore,the first request by P4 is safe to grant, while the second request is not.

**Date: 17/10/24 Signature of faculty in-charge**