**Task 1:** Write a short note on Banker’s algorithm stating its main purpose and working mechanism.

**Banker’s Algorithm:**

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. Its main purpose is to ensure a safe allocation of resources in a system, preventing deadlock by carefully analyzing requests and determining if granting them would keep the system in a safe state.

**Main Purpose:**

The primary objective of the Banker's algorithm is to manage multiple processes competing for a limited number of resources without leading to deadlock. It evaluates resource requests and only grants them if they do not compromise the system's safety, ensuring that there is always a sequence in which all processes can complete without getting stuck.

**Working Mechanism:**

The Banker's algorithm operates based on the following key steps:

**Data Structures:**

* Available: A vector representing the number of available instances of each resource type.
* Max: A matrix defining the maximum demand of each process for each resource.
* Allocation: A matrix indicating the number of resources of each type currently allocated to each process.
* Need: A matrix representing the remaining resource needs of each process, calculated as Max - Allocation.
* Request Handling: When a process makes a request for resources, the algorithm proceeds through the following steps:
* Check Request Validity: Ensure the request does not exceed the process's maximum claim (Need matrix).
* Provisional Allocation: Temporarily allocate the requested resources to the process and update the data structures (Available, Allocation, and Need).
* Safety Check: Perform a safety algorithm to check if the system remains in a safe state after the provisional allocation. The safety algorithm works by:
* Initializing a Work vector as a copy of Available and a Finish vector for all processes, initially set to false.
* Finding a process whose needs can be met with the current Work vector and marking it as finished, then updating the Work vector by adding the allocated resources of that process.
* Repeating until all processes are finished (system is in a safe state) or no further processes can be marked finished (unsafe state).

**Decision Making:**

* If the system remains in a safe state after the provisional allocation, the resources are officially allocated to the process.
* If the system falls into an unsafe state, the provisional allocation is rolled back, and the process must wait, preventing resource allocation until the state is safe.

**Task 2:** Implement the Banker’s Algorithm explained above in C language.

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int available[MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

bool finish[MAX\_PROCESSES];

bool is\_safe(int process\_id);

void request\_resources(int process\_id, int request[]);

void release\_resources(int process\_id, int release[]);

int main() {

available[0] = 3;

available[1] = 3;

available[2] = 2;

max[0][0] = 7; max[0][1] = 5; max[0][2] = 3;

max[1][0] = 3; max[1][1] = 2; max[1][2] = 2;

max[2][0] = 9; max[2][1] = 0; max[2][2] = 2;

max[3][0] = 2; max[3][1] = 2; max[3][2] = 2;

max[4][0] = 4; max[4][1] = 3; max[4][2] = 3;

allocation[0][0] = 0; allocation[0][1] = 1; allocation[0][2] = 0;

allocation[1][0] = 2; allocation[1][1] = 0; allocation[1][2] = 0;

allocation[2][0] = 3; allocation[2][1] = 0; allocation[2][2] = 2;

allocation[3][0] = 2; allocation[3][1] = 1; allocation[3][2] = 1;

allocation[4][0] = 0; allocation[4][1] = 0; allocation[4][2] = 2;

for (int i = 0; i < MAX\_PROCESSES; ++i) {

for (int j = 0; j < MAX\_RESOURCES; ++j) {

need[i][j] = max[i][j] - allocation[i][j]; }}

for (int i = 0; i < MAX\_PROCESSES; ++i) {

finish[i] = false;}

int request[] = {1, 0, 2};

request\_resources(1, request);

int release[] = {2, 0, 0};

release\_resources(2, release);

return 0; }

bool is\_safe(int process\_id) {

int work[MAX\_RESOURCES];

bool finish\_copy[MAX\_PROCESSES];

for (int i = 0; i < MAX\_RESOURCES; ++i) {

work[i] = available[i];

}

for (int i = 0; i < MAX\_PROCESSES; ++i) {

finish\_copy[i] = finish[i];

}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

work[i] -= need[process\_id][i];

allocation[process\_id][i] += need[process\_id][i];

need[process\_id][i] = 0;

}

finish\_copy[process\_id] = true;

bool all\_finished = false;

while (!all\_finished) {

all\_finished = true;

for (int i = 0; i < MAX\_PROCESSES; ++i) {

if (!finish\_copy[i]) {

bool can\_finish = true;

for (int j = 0; j < MAX\_RESOURCES; ++j) {

if (need[i][j] > work[j]) {

can\_finish = false;

break;}}

if (can\_finish) {

finish\_copy[i] = true;

all\_finished = false;

for (int j = 0; j < MAX\_RESOURCES; ++j) {

work[j] += allocation[i][j];}}}}}

for (int i = 0; i < MAX\_PROCESSES; ++i) {

if (!finish\_copy[i]) {

return false;}}

return true;

}

void request\_resources(int process\_id, int request[]) {

printf("Process %d requesting resources: ", process\_id);

for (int i = 0; i < MAX\_RESOURCES; ++i) {

printf("%d ", request[i]);

}

printf("\n");

for (int i = 0; i < MAX\_RESOURCES; ++i) {

if (request[i] > need[process\_id][i]) {

printf("Error: Request exceeds need\n");

return;

}

}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

if (request[i] > available[i]) {

printf("Error: Request exceeds available\n");

return;}}

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] -= request[i];

allocation[process\_id][i] += request[i];

need[process\_id][i] -= request[i];

}

if (is\_safe(process\_id)) {

printf("Request granted\n");

} else {

printf("Request denied\n");

for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] += request[i];

allocation[process\_id][i] -= request[i];

need[process\_id][i] += request[i];}}}

void release\_resources(int process\_id, int release[]) {

printf("Process %d releasing resources: ", process\_id);

for (int i = 0; i < MAX\_RESOURCES; ++i) {

printf("%d ", release[i]);

}

printf("\n");

for (int i = 0; i < MAX\_RESOURCES; ++i) {

if (release[i] > allocation[process\_id][i]) {

printf("Error: Release exceeds allocation\n");

return;

}

} for (int i = 0; i < MAX\_RESOURCES; ++i) {

available[i] += release[i];

allocation[process\_id][i] -= release[i];

need[process\_id][i] += release[i];

}

printf("Resources released\n");

for (int i = 0; i < MAX\_PROCESSES; ++i) {

if (!finish[i]) {

bool can\_finish = true;

for (int j = 0; j < MAX\_RESOURCES; ++j) {

if (need[i][j] > available[j]) {

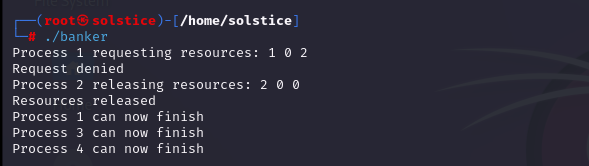
can\_finish = false;

break;}}

if (can\_finish) {

printf("Process %d can now finish\n", i);}}}}

**Output:**

****