**COMPUTER SCIENCE DEPARTMENT**

|  |
| --- |
| **Total Marks:** |
| **Obtained Marks:** |

Operating System Lab

**LAB TASK#10**

**Last date of Submission: May 2, 2025**

**Submitted To: Sir Jawad Naseer**

**Student Name: Ubaid Bin Waris**

**Reg Number: 2212416**

**COMPUTER SCIENCE DEPARTMENT**

**Question no.1**

**Write a C program to implement the deadlock detection algorithm. Take the Allocation, Request, and Available matrices as input and identify which processes are deadlocked. Extend the deadlock detection program to perform deadlock recovery by terminating one or more processes until the deadlock is resolved.**

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Function to find the process that will be terminated to resolve the deadlock

int findProcessToTerminate(bool finish[], int n) {

for (int i = 0; i < n; i++) {

if (!finish[i]) {

return i; // Return the first non-finished process

}

}

return -1; // No process found

}

// Function to check if the system is in a safe state

bool isSafe(int n, int m, int allocation[][MAX\_RESOURCES], int request[][MAX\_RESOURCES], int available[], bool finish[]) {

int work[MAX\_RESOURCES];

for (int i = 0; i < m; i++) {

work[i] = available[i];

}

// Try to find a process that can finish

bool progressMade;

do {

progressMade = false;

for (int i = 0; i < n; i++) {

if (!finish[i]) {

bool canFinish = true;

for (int j = 0; j < m; j++) {

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

canFinish = false;

break;

}

}

if (canFinish) {

// If the process can finish, release its resources

for (int j = 0; j < m; j++) {

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

}

finish[i] = true;

progressMade = true;

}

}

}

} while (progressMade);

// Check if all processes are finished

for (int i = 0; i < n; i++) {

if (!finish[i]) {

return false; // If any process is not finished, it's a deadlock

}

}

return true;

}

// Function to detect deadlock and handle recovery

void deadlockDetectionAndRecovery(int n, int m, int allocation[][MAX\_RESOURCES], int request[][MAX\_RESOURCES], int available[]) {

bool finish[MAX\_PROCESSES] = {false};

bool deadlockDetected = false;

// Step 1: Detect if there's a deadlock

if (!isSafe(n, m, allocation, request, available, finish)) {

deadlockDetected = true;

printf("\nDeadlock detected!\n");

// Step 2: Find which processes are involved in the deadlock and try to resolve it

while (deadlockDetected) {

printf("\nAttempting to resolve deadlock...\n");

int processToTerminate = findProcessToTerminate(finish, n);

if (processToTerminate != -1) {

printf("Terminating process %d to resolve deadlock...\n", processToTerminate);

// Release resources allocated to the terminated process

for (int j = 0; j < m; j++) {

available[j] += allocation[processToTerminate][j];

allocation[processToTerminate][j] = 0;

request[processToTerminate][j] = 0;

}

finish[processToTerminate] = true; // Mark the process as terminated

}

// Check again for deadlock after terminating a process

if (isSafe(n, m, allocation, request, available, finish)) {

deadlockDetected = false;

printf("Deadlock resolved successfully.\n");

} else {

deadlockDetected = true;

}

}

} else {

printf("\nNo deadlock detected.\n");

}

}

int main(void) {

int n, m;

int allocation[MAX\_PROCESSES][MAX\_RESOURCES], request[MAX\_PROCESSES][MAX\_RESOURCES], available[MAX\_RESOURCES];

// Input: Number of processes and number of resource types

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the number of resource types: ");

scanf("%d", &m);

// Input allocation matrix

printf("Enter the allocation matrix:\n");

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

}

}

// Input request matrix

printf("Enter the request matrix:\n");

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

scanf("%d", &request[i][j]);

}

}

// Input available resources vector

printf("Enter the available resources vector:\n");

for (int i = 0; i < m; i++) {

scanf("%d", &available[i]);

}

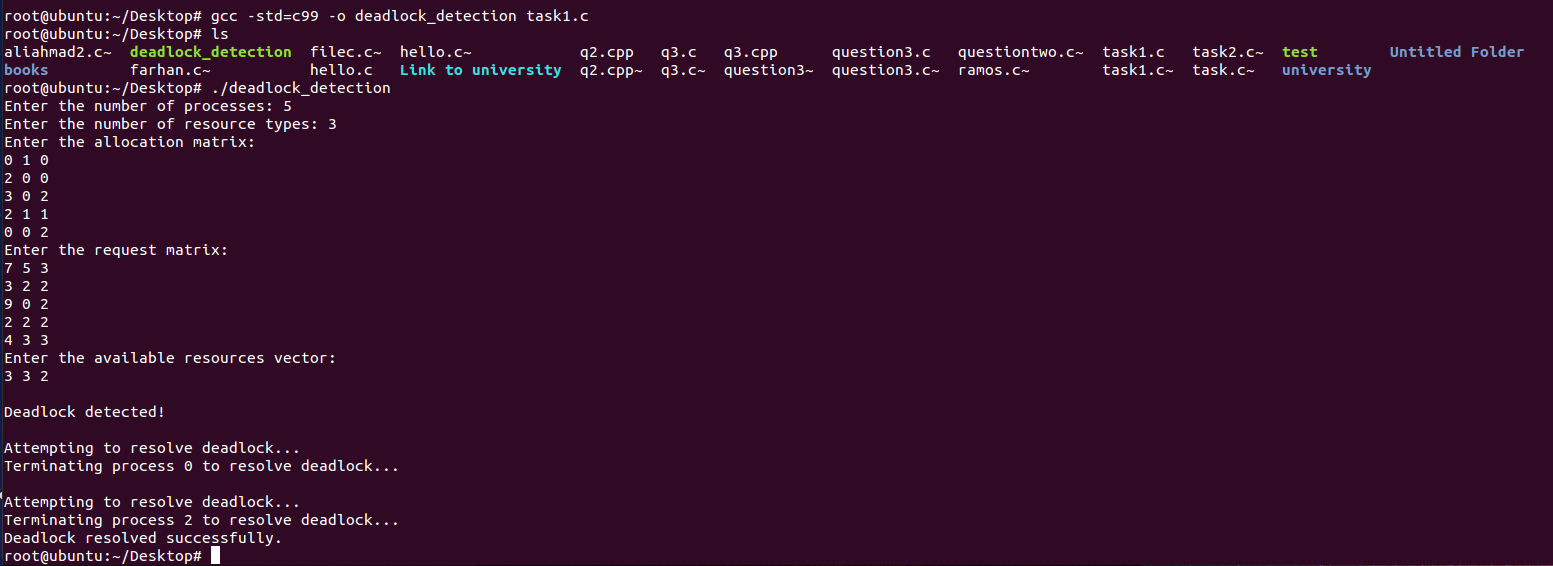
// Call deadlock detection and recovery function

deadlockDetectionAndRecovery(n, m, allocation, request, available);

return 0;

}

**Output**



**Question no.2**

**Write a C program to implement the Banker's Algorithm for deadlock avoidance. Take input for the number of processes, number of resources, allocation matrix, maximum demand matrix, and available resources. Display whether the system is in a safe state. Modify your Banker's Algorithm program to take an additional resource request from a particular process and check if the request can be safely granted.**

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Function to check if the system is in a safe state

bool isSafe(int n, int m, int allocation[][MAX\_RESOURCES], int maximum[][MAX\_RESOURCES], int available[], int safeSequence[]) {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {false};

int i, j, count = 0;

// Initialize work array with available resources

for (i = 0; i < m; i++) {

work[i] = available[i];

}

while (count < n) {

bool progressMade = false;

for (i = 0; i < n; i++) {

if (!finish[i]) {

// Check if the process can be completed

bool canFinish = true;

for (j = 0; j < m; j++) {

if (maximum[i][j] - allocation[i][j] > work[j]) {

canFinish = false;

break;

}

}

if (canFinish) {

// Add the resources allocated to the process to work

for (j = 0; j < m; j++) {

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

}

// Mark the process as finished

safeSequence[count++] = i;

finish[i] = true;

progressMade = true;

}

}

}

// If no process can be finished in this round, then the system is not in a safe state

if (!progressMade) {

return false;

}

}

return true;

}

// Function to request resources and check if it can be granted

bool requestResources(int n, int m, int allocation[][MAX\_RESOURCES], int maximum[][MAX\_RESOURCES], int available[], int processID, int request[]) {

int i;

// Check if request is less than or equal to the need

for (i = 0; i < m; i++) {

if (request[i] > maximum[processID][i] - allocation[processID][i]) {

printf("Error: Process has exceeded maximum claim\n");

return false;

}

}

// Check if request is less than or equal to available resources

for (i = 0; i < m; i++) {

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

printf("Error: Resources are not available\n");

return false;

}

}

// Pretend to allocate the resources

for (i = 0; i < m; i++) {

available[i] -= request[i];

allocation[processID][i] += request[i];

}

// Check if the system is still in a safe state after allocation

int safeSequence[MAX\_PROCESSES];

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("Request can be safely granted.\n");

return true;

} else {

// Rollback allocation if not safe

for (i = 0; i < m; i++) {

available[i] += request[i];

allocation[processID][i] -= request[i];

}

printf("Error: Request cannot be granted as it leads to an unsafe state\n");

return false;

}

}

int main() {

int n, m;

int allocation[MAX\_PROCESSES][MAX\_RESOURCES], maximum[MAX\_PROCESSES][MAX\_RESOURCES], available[MAX\_RESOURCES];

int safeSequence[MAX\_PROCESSES];

int i, j, processID;

int request[MAX\_RESOURCES];

// Input number of processes and resources

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the number of resource types: ");

scanf("%d", &m);

// Input allocation matrix

printf("Enter the allocation matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

}

}

// Input maximum matrix

printf("Enter the maximum demand matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &maximum[i][j]);

}

}

// Input available resources

printf("Enter the available resources vector:\n");

for (i = 0; i < m; i++) {

scanf("%d", &available[i]);

}

// Step 1: Check if the system is in a safe state

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("System is in a safe state.\n");

printf("Safe sequence: ");

for (i = 0; i < n; i++) {

printf("P%d ", safeSequence[i]);

}

printf("\n");

} else {

printf("System is not in a safe state.\n");

}

// Step 2: Request resources from a particular process

printf("\nEnter the process ID (0 to %d) to request resources: ", n - 1);

scanf("%d", &processID);

printf("Enter the resource request for process P%d: ", processID);

for (i = 0; i < m; i++) {

scanf("%d", &request[i]);

}

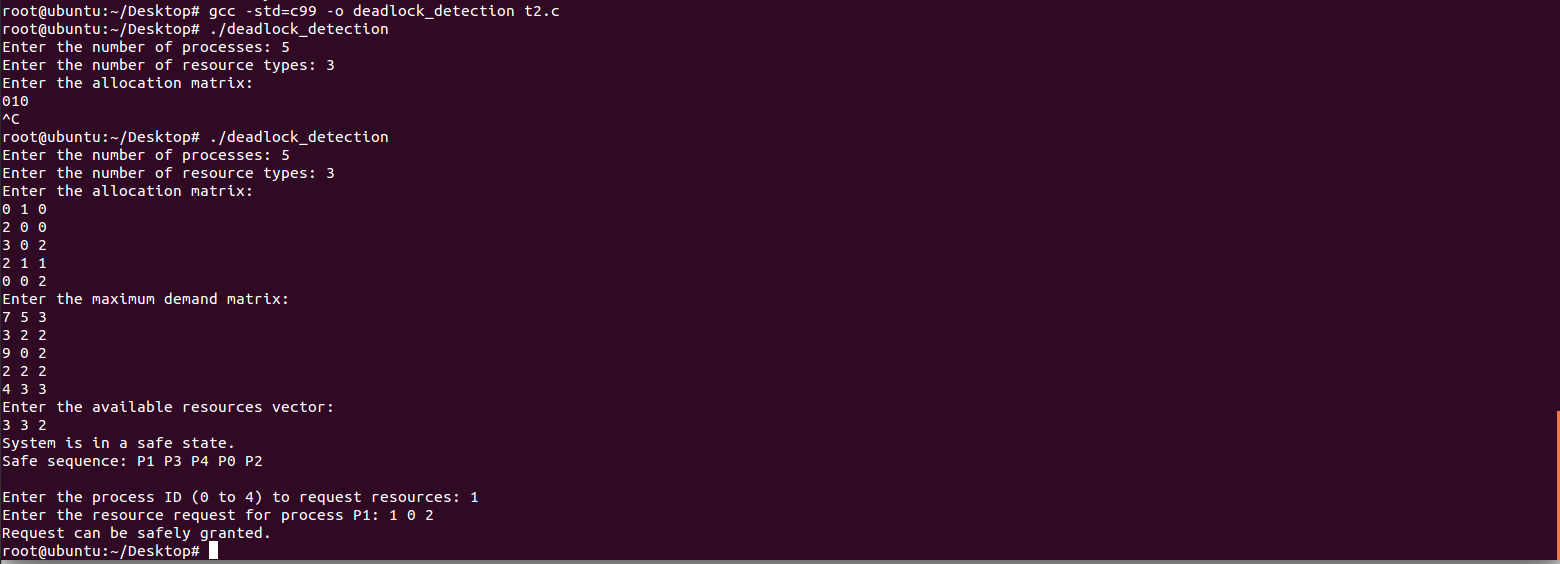
// Step 3: Check if the request can be granted safely

requestResources(n, m, allocation, maximum, available, processID, request);

return 0;

}

**Output**



**Question no.3**

**Write a C program to implement the Banker’s Algorithm for deadlock avoidance. Take input for the number of processes, number of resources, allocation matrix, maximum demand matrix, and available resources. Display whether the system is in a safe state.**

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Function to check if the system is in a safe state

bool isSafe(int n, int m, int allocation[][MAX\_RESOURCES], int maximum[][MAX\_RESOURCES], int available[], int safeSequence[]) {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {false};

int i, j, count = 0;

// Initialize work array with available resources

for (i = 0; i < m; i++) {

work[i] = available[i];

}

// Find a safe sequence

while (count < n) {

bool progressMade = false;

for (i = 0; i < n; i++) {

if (!finish[i]) {

// Check if the process can be completed

bool canFinish = true;

for (j = 0; j < m; j++) {

if (maximum[i][j] - allocation[i][j] > work[j]) {

canFinish = false;

break;

}

}

if (canFinish) {

// Add the resources allocated to the process to work

for (j = 0; j < m; j++) {

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

}

// Mark the process as finished

safeSequence[count++] = i;

finish[i] = true;

progressMade = true;

}

}

}

// If no process can be finished in this round, then the system is not in a safe state

if (!progressMade) {

return false;

}

}

return true;

}

int main() {

int n, m;

int allocation[MAX\_PROCESSES][MAX\_RESOURCES], maximum[MAX\_PROCESSES][MAX\_RESOURCES], available[MAX\_RESOURCES];

int safeSequence[MAX\_PROCESSES];

int i, j;

// Input number of processes and resources

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the number of resource types: ");

scanf("%d", &m);

// Input allocation matrix

printf("Enter the allocation matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

}

}

// Input maximum matrix

printf("Enter the maximum demand matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &maximum[i][j]);

}

}

// Input available resources

printf("Enter the available resources vector:\n");

for (i = 0; i < m; i++) {

scanf("%d", &available[i]);

}

// Step 1: Check if the system is in a safe state

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("System is in a safe state.\n");

printf("Safe sequence: ");

for (i = 0; i < n; i++) {

printf("P%d ", safeSequence[i]);

}

printf("\n");

} else {

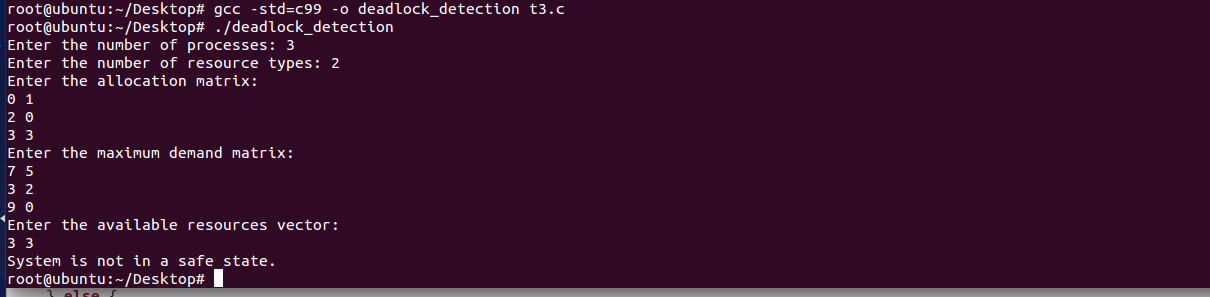
printf("System is not in a safe state.\n");

}

return 0;

}

**Output**



**Question no.4**

**Implement the Banker’s Algorithm where each process can request and release multiple instances of different resource types. Verify and show the resulting safe state or deadlock condition.**

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Function to check if the system is in a safe state

bool isSafe(int n, int m, int allocation[][MAX\_RESOURCES], int maximum[][MAX\_RESOURCES], int available[], int safeSequence[]) {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {false};

int i, j, count = 0;

// Initialize work array with available resources

for (i = 0; i < m; i++) {

work[i] = available[i];

}

// Find a safe sequence

while (count < n) {

bool progressMade = false;

for (i = 0; i < n; i++) {

if (!finish[i]) {

// Check if the process can be completed

bool canFinish = true;

for (j = 0; j < m; j++) {

if (maximum[i][j] - allocation[i][j] > work[j]) {

canFinish = false;

break;

}

}

if (canFinish) {

// Add the resources allocated to the process to work

for (j = 0; j < m; j++) {

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

}

// Mark the process as finished

safeSequence[count++] = i;

finish[i] = true;

progressMade = true;

}

}

}

// If no process can be finished in this round, then the system is not in a safe state

if (!progressMade) {

return false;

}

}

return true;

}

// Function to request resources and check if it can be granted

bool requestResources(int n, int m, int allocation[][MAX\_RESOURCES], int maximum[][MAX\_RESOURCES], int available[], int processID, int request[]) {

int i;

// Check if request is less than or equal to the need

for (i = 0; i < m; i++) {

if (request[i] > maximum[processID][i] - allocation[processID][i]) {

printf("Error: Process has exceeded maximum claim\n");

return false;

}

}

// Check if request is less than or equal to available resources

for (i = 0; i < m; i++) {

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

printf("Error: Resources are not available\n");

return false;

}

}

// Pretend to allocate the resources

for (i = 0; i < m; i++) {

available[i] -= request[i];

allocation[processID][i] += request[i];

}

// Check if the system is still in a safe state after allocation

int safeSequence[MAX\_PROCESSES];

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("Request can be safely granted.\n");

return true;

} else {

// Rollback allocation if not safe

for (i = 0; i < m; i++) {

available[i] += request[i];

allocation[processID][i] -= request[i];

}

printf("Error: Request cannot be granted as it leads to an unsafe state\n");

return false;

}

}

// Function to release resources and update the system state

void releaseResources(int n, int m, int allocation[][MAX\_RESOURCES], int available[], int processID, int release[]) {

int i;

// Release the resources

for (i = 0; i < m; i++) {

allocation[processID][i] -= release[i];

available[i] += release[i];

}

printf("Resources released by process P%d.\n", processID);

}

int main() {

int n, m;

int allocation[MAX\_PROCESSES][MAX\_RESOURCES], maximum[MAX\_PROCESSES][MAX\_RESOURCES], available[MAX\_RESOURCES];

int safeSequence[MAX\_PROCESSES];

int i, j;

// Input number of processes and resources

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the number of resource types: ");

scanf("%d", &m);

// Input allocation matrix

printf("Enter the allocation matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

}

}

// Input maximum matrix

printf("Enter the maximum demand matrix:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++) {

scanf("%d", &maximum[i][j]);

}

}

// Input available resources

printf("Enter the available resources vector:\n");

for (i = 0; i < m; i++) {

scanf("%d", &available[i]);

}

// Step 1: Check if the system is in a safe state

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("System is in a safe state.\n");

printf("Safe sequence: ");

for (i = 0; i < n; i++) {

printf("P%d ", safeSequence[i]);

}

printf("\n");

} else {

printf("System is not in a safe state.\n");

}

// Step 2: Request resources from a particular process

int processID, request[MAX\_RESOURCES];

printf("\nEnter the process ID (0 to %d) to request resources: ", n - 1);

scanf("%d", &processID);

printf("Enter the resource request for process P%d: ", processID);

for (i = 0; i < m; i++) {

scanf("%d", &request[i]);

}

// Step 3: Check if the request can be granted safely

requestResources(n, m, allocation, maximum, available, processID, request);

// Step 4: Release resources from a particular process

int release[MAX\_RESOURCES];

printf("\nEnter the process ID (0 to %d) to release resources: ", n - 1);

scanf("%d", &processID);

printf("Enter the resources to release from process P%d: ", processID);

for (i = 0; i < m; i++) {

scanf("%d", &release[i]);

}

// Step 5: Release resources and check new system state

releaseResources(n, m, allocation, available, processID, release);

// Recheck if the system is in a safe state after the release

if (isSafe(n, m, allocation, maximum, available, safeSequence)) {

printf("After resource release, the system is in a safe state.\n");

printf("Safe sequence: ");

for (i = 0; i < n; i++) {

printf("P%d ", safeSequence[i]);

}

printf("\n");

} else {

printf("After resource release, the system is not in a safe state.\n");

}

return 0;

}

**Output**

