Contents

[Overview 2](#_Toc187441456)

[Flow of the Program 2](#_Toc187441457)

[Functions in the Program 3](#_Toc187441458)

[Conclusion 3](#_Toc187441459)

[Code 4](#_Toc187441460)

[Output 16](#_Toc187441461)

# 

# Overview

This program is a resource allocation simulator based on the **Banker's Algorithm**, a widely used method in operating systems to avoid deadlock situations. The algorithm ensures that a system remains in a safe state by carefully managing resource requests and releases. The program simulates a system with a fixed number of resources and customers and offers features to request, release, and display system states, as well as detect and resolve deadlocks.

# Flow of the Program

The program consists of the following key steps:

1. **Initialization**: The user provides the available resources, the maximum demand of each customer, and the initially allocated resources. These inputs are stored in the respective data structures.
2. **Command Processing**: The program enters a loop where the user can issue commands:
   * **RQ (Request Resources)**: A customer requests additional resources. The program verifies if the request is valid and safe before allocation.
   * **RL (Release Resources)**: A customer releases some of its allocated resources back to the system.
   * **CS (Check State)**: Displays the current system state and detects deadlock, resolving it if necessary.
   * **exit**: Ends the program.
3. **Core Algorithm**: The program uses the Banker's Algorithm to ensure that any resource allocation maintains the system's safety. If a request leads to an unsafe state, the program denies the request and reverts any temporary changes.
4. **Deadlock Detection and Resolution**: If deadlock is detected, the program resolves it by preempting resources from one or more customers until the system returns to a safe state.

# Functions in the Program

1. **Initialization Functions**:
   * Inputs the available resources, maximum demand, and allocated resources for each customer.
2. **Key Functionalities**:
   * **is\_safe\_state**: Checks if the system is in a safe state by simulating resource allocation and ensuring all processes can eventually complete.
   * **request\_resources**: Processes a resource request from a customer, granting it only if the system remains in a safe state.
   * **release\_resources**: Releases resources allocated to a customer, updating the system's state.
3. **System State Management**:
   * **display\_state**: Outputs the available, maximum, allocated, and needed resources for all customers.
   * **detect\_deadlock**: Identifies if a deadlock exists in the system.
   * **resolve\_deadlock**: Resolves deadlocks by preempting resources from one or more customers.
4. **Command Processing**:
   * A loop that handles user commands (RQ, RL, CS, or exit), invoking the appropriate functions based on input.

# Conclusion

This program demonstrates the practical implementation of the Banker's Algorithm to manage shared resources among multiple customers effectively. It ensures that the system remains in a safe state, preventing deadlock and ensuring fairness in resource allocation. The code also handles deadlock resolution, showcasing robust resource management strategies.

The implementation is flexible for educational and experimental purposes, allowing users to test various scenarios of resource allocation, deadlock detection, and resolution. With minor extensions, this program could be adapted for real-world use in systems requiring reliable resource allocation management.

# Code

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define NUMBER\_OF\_CUSTOMERS 5

#define NUMBER\_OF\_RESOURCES 4

// Data structures

int available[NUMBER\_OF\_RESOURCES];

int maximum[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

int allocation[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

int need[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

char resource\_names[NUMBER\_OF\_RESOURCES] = {'A', 'B', 'C', 'D'}; // Resource names

// Function prototypes

bool is\_safe\_state();

int request\_resources(int customer\_num, int request[]);

void release\_resources(int customer\_num, int release[]);

void display\_state();

bool detect\_deadlock();

void resolve\_deadlock();

int main() {

// Initialize data

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

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

printf("Resource %c: ", resource\_names[i]);

if (scanf("%d", &available[i]) != 1) {

printf("Error: Invalid input for available resources.\n");

return 1;

}

}

printf("Enter the maximum resources for each customer:\n");

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

printf("Customer %d:\n", i);

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

printf(" Max %c: ", resource\_names[j]);

if (scanf("%d", &maximum[i][j]) != 1) {

printf("Error: Invalid input for maximum resources.\n");

return 1;

}

}

}

printf("Enter the allocated resources for each customer:\n");

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

printf("Customer %d:\n", i);

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

printf(" Allocated %c: ", resource\_names[j]);

if (scanf("%d", &allocation[i][j]) != 1) {

printf("Error: Invalid input for allocated resources.\n");

return 1;

}

// Calculate need matrix and ensure no negative values

need[i][j] = maximum[i][j] - allocation[i][j];

if (need[i][j] < 0) {

need[i][j] = 0; // Set negative values to zero

}

}

}

// Command interface

char command[3];

while (1) {

printf("\nEnter a command (RQ, RL, CS, or exit):\n");

scanf("%s", command);

if (strcmp(command, "RQ") == 0) {

int customer\_num, request[NUMBER\_OF\_RESOURCES];

printf("Enter customer number: ");

if (scanf("%d", &customer\_num) != 1) {

printf("Error: Invalid input for customer number.\n");

continue;

}

printf("Enter resource request:\n");

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

printf(" Request %c: ", resource\_names[i]);

if (scanf("%d", &request[i]) != 1) {

printf("Error: Invalid input for resource request.\n");

break;

}

}

if (request\_resources(customer\_num, request) == 0) {

printf("Request granted.\n");

if (is\_safe\_state()) {

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

} else {

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

}

} else {

printf("Request denied.\n");

}

} else if (strcmp(command, "RL") == 0) {

int customer\_num, release[NUMBER\_OF\_RESOURCES];

printf("Enter customer number: ");

if (scanf("%d", &customer\_num) != 1) {

printf("Error: Invalid input for customer number.\n");

continue;

}

printf("Enter resources to release:\n");

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

printf(" Release %c: ", resource\_names[i]);

if (scanf("%d", &release[i]) != 1) {

printf("Error: Invalid input for resource release.\n");

break;

}

}

release\_resources(customer\_num, release);

printf("Resources released.\n");

if (is\_safe\_state()) {

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

} else {

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

}

} else if (strcmp(command, "CS") == 0) {

display\_state();

if (detect\_deadlock()) {

printf("Deadlock detected! Resolving...\n");

resolve\_deadlock();

} else {

printf("No deadlock detected.\n");

}

} else if (strcmp(command, "exit") == 0) {

break;

} else {

printf("Invalid command. Use 'RQ', 'RL', 'CS', or 'exit'.\n");

}

}

return 0;

}

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

bool is\_safe\_state() {

int work[NUMBER\_OF\_RESOURCES];

bool finish[NUMBER\_OF\_CUSTOMERS] = {false};

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

work[i] = available[i];

}

do {

bool found = false;

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

if (!finish[i]) {

bool can\_allocate = true;

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

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

can\_allocate = false;

break;

}

}

if (can\_allocate) {

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

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

}

finish[i] = true;

found = true;

}

}

}

if (!found) {

break;

}

} while (1);

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

if (!finish[i]) {

return false;

}

}

return true;

}

// Function to request resources

int request\_resources(int customer\_num, int request[]) {

if (customer\_num < 0 || customer\_num >= NUMBER\_OF\_CUSTOMERS) {

printf("Error: Invalid customer number.\n");

return -1;

}

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

if (request[i] < 0 || request[i] > need[customer\_num][i]) {

printf("Error: Request exceeds the need or invalid input.\n");

return -1;

}

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

printf("Error: Not enough available resources.\n");

return -1;

}

}

// Temporarily allocate resources.

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

available[i] -= request[i];

allocation[customer\_num][i] += request[i];

need[customer\_num][i] -= request[i];

}

// Check if the state is safe.

if (!is\_safe\_state()) {

printf("Error: Request leads to unsafe state. Rolling back.\n");

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

available[i] += request[i];

allocation[customer\_num][i] -= request[i];

need[customer\_num][i] += request[i];

}

return -1;

}

return 0;

}

// Function to release resources

void release\_resources(int customer\_num, int release[]) {

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

if (release[i] < 0 || release[i] > allocation[customer\_num][i]) {

printf("Error: Invalid release amount for resource %c.\n", resource\_names[i]);

return;

}

allocation[customer\_num][i] -= release[i];

available[i] += release[i];

need[customer\_num][i] += release[i];

}

}

// Function to display the current system state

void display\_state() {

printf("\nCurrent System State:\n");

printf("Available resources:\n");

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

printf("%c: %d\n", resource\_names[i], available[i]);

}

printf("\nMaximum resources:\n");

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

printf("Customer %d: ", i);

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

printf("%c: %d ", resource\_names[j], maximum[i][j]);

}

printf("\n");

}

printf("\nAllocated resources:\n");

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

printf("Customer %d: ", i);

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

printf("%c: %d ", resource\_names[j], allocation[i][j]);

}

printf("\n");

}

printf("\nNeed resources:\n");

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

printf("Customer %d: ", i);

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

printf("%c: %d ", resource\_names[j], need[i][j]);

}

printf("\n");

}

}

// Function to detect deadlock

bool detect\_deadlock() {

int work[NUMBER\_OF\_RESOURCES];

bool finish[NUMBER\_OF\_CUSTOMERS] = {false};

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

work[i] = available[i];

}

while (1) {

bool progress = false;

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

if (!finish[i]) {

bool can\_allocate = true;

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

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

can\_allocate = false;

break;

}

}

if (can\_allocate) {

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

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

}

finish[i] = true;

progress = true;

}

}

}

if (!progress) {

break;

}

}

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

if (!finish[i]) {

return true;

}

}

return false;

}

// Function to resolve deadlock

void resolve\_deadlock() {

for (int i = 0; i < NUMBER\_OF\_CUSTOMERS; i++) {

bool deadlocked = detect\_deadlock();

if (!deadlocked) {

printf("Deadlock resolved.\n");

return;

}

// Preempt resources from the first deadlocked customer

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

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

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

allocation[i][j] = 0;

}

printf("Preempted resources from customer %d.\n", i);

}

if (detect\_deadlock()) {

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

} else {

printf("Deadlock resolved.\n");

}

}

………………………………………………………………………………………………………………………………………………………………

# Output









