6. Simulate Bankers Algorithm for Dead Lock Avoidance.

The Banker's Algorithm is a deadlock avoidance algorithm that's used by an operating system to manage resource allocation and ensure that deadlock situations don't occur.

Sure, let's simulate the Banker's Algorithm for deadlock avoidance with an example. In this example, we have 5 processes and 3 types of resources (A, B, and C).

Process | Maximum | Allocated

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

P0 | 7 5 3 | 0 1 0

P1 | 3 2 2 | 2 0 0

P2 | 9 0 2 | 3 0 2

P3 | 2 2 2 | 2 1 1

P4 | 4 3 3 | 0 0 2

The total available resources are 10 A, 5 B, and 7 C.

We'll simulate the Banker's Algorithm to check if there is a safe sequence for these processes. Here's the step-by-step simulation:

Step 1: Calculate the Need Matrix

The Need matrix is calculated as Maximum - Allocated.

Need Matrix:

A B C

P0 7 4 3

P1 1 2 2

P2 6 0 0

P3 0 1 1

P4 4 3 1

Step 2: Initialize Work and Finish Arrays

Work = Available Resources = [10, 5, 7]

Finish = [False, False, False, False, False]

Step 3: Find a process i such that Finish[i] is False and Need[i] <= Work

Start the loop to find a safe sequence. For each iteration, we find a process that satisfies the condition Finish[i] is False and Need[i] <= Work:

- Iteration 1: P1 satisfies the condition (Finish[1] is False and Need[1] <= Work). We add P1 to the sequence.

Sequence: [P1]

- Iteration 2: P3 satisfies the condition (Finish[3] is False and Need[3] <= Work). We add P3 to the sequence.

Sequence: [P1, P3]

- Iteration 3: P4 satisfies the condition (Finish[4] is False and Need[4] <= Work). We add P4 to the sequence.

Sequence: [P1, P3, P4]

- Iteration 4: P0 satisfies the condition (Finish[0] is False and Need[0] <= Work). We add P0 to the sequence.

Sequence: [P1, P3, P4, P0]

- Iteration 5: P2 satisfies the condition (Finish[2] is False and Need[2] <= Work). We add P2 to the sequence.

Sequence: [P1, P3, P4, P0, P2]

Now, all processes are in the sequence. Check if Finish[i] is True for all i.

Finish = [True, True, True, True, True]

Since all processes are in the sequence and Finish[i] is True for all i, this is a safe sequence:

Safe Sequence: [P1, P3, P4, P0, P2]

This sequence shows the order in which the processes can safely execute to avoid a deadlock.

Certainly, here's a simple C program to simulate the Banker's Algorithm for deadlock avoidance with an example. This program takes the maximum and allocated resources for each process, along with the total available resources, and then determines if there is a safe sequence for the processes:

#include <stdio.h>

int main() {

int num\_processes = 5;

int num\_resources = 3;

int max\_resources[5][3] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int allocated\_resources[5][3] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int total\_resources[3] = {10, 5, 7};

// Calculate the need matrix

int need[5][3];

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

need[i][j] = max\_resources[i][j] - allocated\_resources[i][j];

}

}

// Initialize work and finish arrays

int work[3];

int finish[5] = {0};

for (int i = 0; i < num\_resources; i++) {

work[i] = total\_resources[i];

}

int sequence[5];

int count = 0;

while (count < num\_processes) {

int found = 0;

for (int i = 0; i < num\_processes; i++) {

if (finish[i] == 0) {

int j;

for (j = 0; j < num\_resources; j++) {

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

break;

}

}

if (j == num\_resources) {

for (int k = 0; k < num\_resources; k++) {

work[k] += allocated\_resources[i][k];

}

sequence[count++] = i;

finish[i] = 1;

found = 1;

}

}

}

if (found == 0) {

break; // No suitable process found, indicating possible deadlock

}

}

if (count < num\_processes) {

printf("No safe sequence found. Deadlock might occur.\n");

} else {

printf("Safe Sequence: ");

for (int i = 0; i < num\_processes; i++) {

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

if (i < num\_processes - 1) {

printf(" -> ");

}

}

printf("\n");

}

return 0;

}

This C program simulates the Banker's Algorithm with your example and prints the safe sequence if one is found.

7. Simulate Bankers Algorithm for Dead Lock Prevention.

The Banker's Algorithm is used for deadlock prevention in operating systems. It ensures that resources are allocated in a way that won't lead to deadlock. Here's a simplified simulation:

Let's assume we have 3 resources (A, B, C) and 5 processes (P0, P1, P2, P3, P4). We also know the maximum resource needs for each process and the currently allocated resources:

**Maximum resource needs:**

A B C

P0 7 5 3

P1 3 2 2

P2 9 0 2

P3 2 2 2

P4 4 3 3

**Currently allocated resources:**

A B C

P0 0 1 0

P1 2 0 0

P2 3 0 2

P3 2 1 1

P4 0 0 2

**Available resources:**

A B C

3 3 2

Now, let's simulate the Banker's Algorithm:

1. Calculate the need matrix (Maximum - Allocated):

A B C

P0 7 4 3

P1 1 2 2

P2 6 0 0

P3 0 1 1

P4 4 3 1

2. Initialize two arrays: `work` (initially equal to the available resources) and `finish` (all set to `false`).

3. Loop through the processes:

a. For each process `P[i]`, check if `finish[i]` is `false` and if `need[i]` is less than or equal to `work`. If true, grant resources and update `work`.

4. Repeat step 3 until no more processes can be granted resources or all processes have finished.

The algorithm will ensure that resources are allocated in a way that doesn't lead to deadlock. If you have a specific scenario or request for more details, please let me know.

Certainly! Here's a simple C program that simulates the Banker's Algorithm for deadlock prevention:

#include <stdio.h>

int processes = 5; // Number of processes

int resources = 3; // Number of resource types

// Function to check if a process can be allocated resources

int canAllocate(int need[5][3], int available[3], int process) {

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

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

return 0;

}

}

return 1;

}

// Function to find a safe sequence

void findSafeSequence(int allocation[5][3], int max[5][3], int available[3]) {

int need[5][3];

int work[3];

int finish[5] = {0};

// Initialize need matrix

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

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

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

}

}

// Initialize work vector

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

work[j] = available[j];

}

// Find a safe sequence

int safeSeq[5];

int count = 0;

while (count < processes) {

int found = 0;

for (int p = 0; p < processes; p++) {

if (finish[p] == 0 && canAllocate(need, work, p)) {

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

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

}

safeSeq[count++] = p;

finish[p] = 1;

found = 1;

}

}

if (!found) {

printf("System is in an unsafe state.\n");

return;

}

}

printf("System is in a safe state.\nSafe sequence: ");

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

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

if (i < processes - 1) {

printf(" -> ");

}

}

printf("\n");

}

int main() {

int allocation[5][3] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int max[5][3] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int available[3] = {3, 3, 2};

findSafeSequence(allocation, max, available);

return 0;

}

This C program demonstrates the Banker's Algorithm with a simple example. You can modify the `allocation`, `max`, and `available` arrays to simulate different scenarios.