**CSE-316**

**SIMULATION BASED ASSIGNMENT**

**Topic**: Banker’s Algorithm in multithreaded programming

**Section: - K22UN**

### A black and orange logo Description automatically generated

Lovely Professional University

Jalandhar, Punjab, India.

|  |  |
| --- | --- |
| **Submitted by:** | **Submitted To:** |
| Name of the Student: Swayam Swarup Barik  Reg. No.: 12205790  Roll No.: 22  Signature: | Name of the Faculty: Mrs. Cherry Khosla  UID: 13436  Signature: |

**Question no :3**

Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

**Ensure:**

1. The program should be dynamic such that the threads are created at run time based on the input from the user.

2. The resources must be displaced after each allocation.

3. The system state should be visible after each allocation.

1. **INTRODUCTION**

Deadlock, a state where two or more processes are stuck and unable to make progress due to a circular waiting condition, is a paramount concern within the realm of operating systems. To combat this issue, a range of deadlock avoidance algorithms have been devised, and one such solution is the Banker's Algorithm.

Banker's Algorithm stands as a resource allocation and deadlock avoidance algorithm designed to ensure the safe execution of processes by continuously monitoring resource availability. This algorithm is widely applied in operating systems to prevent deadlocks and uphold system reliability.

In a recent project, I have taken the initiative to implement the Banker's Algorithm in the C programming language. My approach leverages multi-threading to simulate the concurrent execution of processes. This software solution prompts the user for input regarding the number of processes, the available resources, and their initial allocation. It then dynamically allocates memory and allows user input for both resource allocation and maximum need. The algorithm subsequently determines if there exists a safe sequence of processes that can be executed, and if so, it proceeds to execute them.

My implemented algorithm serves as a concrete manifestation of theoretical concepts within the field of operating systems. It offers a unique opportunity for hands-on exploration and experimentation with process synchronization and deadlock avoidance strategies. By employing threads, my solution provides a realistic simulation of concurrent processes competing for resources.

1. **REQUIREMENT**

**2.1. User Input:** : The program needs the number of processes, resources, and initial resource availability.

**2.2. Resource Allocation:** We must specify the resources allocated and the maximum required for each process.

**2.3. Maximum Required Resources:** For each process, we should provide the maximum resources that the process may need.

**2.4. Need Matrix Calculation:** This program should calculate the need 2D-matrix, which represents the remaining resources a process needs to complete its execution.

**2.5. Safety Check:** The program should use the Banker's Algorithm to see if it's safe to run processes without deadlocks.

**2.6. Simulation:** If safe, the program simulates process execution using threads.

**2.7. Resource Management**: It should allow resource allocation and release during simulation.

**2.8. Output:** The program displays process progress, including allocation, execution, and release.

**2.9. Error Handling:** It should handle invalid inputs and situations where a safe sequence can't be found.

1. **LOGIC BEHIND THE SOLUTION**

**3.1 Resource Management:** The program gathers data about how resources are allocated and the maximum needs for each process. It calculates what resources are still needed.

**3.2 Safety Check (Banker's Algorithm) :** It employs the Banker's Algorithm to verify if processes can operate without deadlock risks.

**3.3 Process Emulation:** If it's safe, the program uses threads to emulate process execution. These threads mimic resource allocation and release**.**

* 1. **Progress Updates:** The program provides status updates for each process, including details on allocated and needed resources**.**
  2. **Resource Handling**: During the simulation, processes perform resource allocation and release actions, simulating a real-world operating system environment.

1. **PSEUDO CODE**

**Initialize Global Variables**

nResources, nProcesses, resources, allocated, maxRequired, need, safeSeq, nProcessRan

**Initialize Lock and Condition Variable**

lockResources, condition

**Function getSafeSeq():**

Initialize tempRes

Initialize finished

while nfinished < nProcesses:

safe = false

for i = 0 to nProcesses - 1:

if not finished[i]:

possible = true

for j = 0 to nResources - 1:

if need[i][j] > tempRes[j]:

possible = false

break

if possible:

for j = 0 to nResources - 1:

tempRes[j] += allocated[i][j]

safeSeq[nfinished] = i

finished[i] = true

nfinished++

safe = true

if not safe:

return false

return true

**Function processCode(arg):**

p = arg

Lock resources using lockResources

while p != safeSeq[nProcessRan]:

Wait on condition using condition

Print process information (Allocated, Needed, Available)

Simulate process execution (sleep for random time)

Release allocated resources

Unlock resources

Increment nProcessRan

Signal all threads using condition

**Main Function:**

Prompt for nProcesses, nResources

Allocate memory for resources, allocated, maxRequired, need, safeSeq

Prompt for currently available resources

Prompt for resource allocation and maximum requirements

Calculate need matrix

Call getSafeSeq():

If return value is false:

Print "Unsafe State!" and exit

Print safe sequence

Create threads for processes

Join all threads

Print "All Processes Finished"

Free allocated memory

**End of Main**

**\*\* flowchart of the pseudo code has been added for better understanding**

**Start**

**|**

**|-- Initialize Global Variables**

**| |-- nResources, nProcesses, resources, allocated, maxRequired, need, safeSeq, nProcessRan**

**|**

**|-- Initialize Lock and Condition Variable**

**| |-- lockResources, condition**

**|**

**|-- GetSafeSeq() Function**

**| |-- Initialize tempRes**

**| |-- Initialize finished**

**| |-- While nfinished < nProcesses:**

**| |-- safe = false**

**| |-- For i = 0 to nProcesses - 1:**

**| |-- If not finished[i]:**

**| |-- possible = true**

**| |-- For j = 0 to nResources - 1:**

**| |-- If need[i][j] > tempRes[j]:**

**| |-- possible = false**

**| |-- Break**

**| |-- If possible:**

**| |-- For j = 0 to nResources - 1:**

**| |-- tempRes[j] += allocated[i][j]**

**| |-- safeSeq[nfinished] = i**

**| |-- finished[i] = true**

**| |-- nfinished++**

**| |-- safe = true**

**| |-- If not safe:**

**| |-- Return false**

**| |-- Return true**

**|**

**|-- ProcessCode() Function**

**| |-- p = arg**

**| |-- Lock resources using lockResources**

**| |-- While p != safeSeq[nProcessRan]:**

**| |-- Wait on condition using condition**

**| |-- Print process information (Allocated, Needed, Available)**

**| |-- Simulate process execution (sleep for random time)**

**| |-- Release allocated resources**

**| |-- Unlock resources**

**| |-- Increment nProcessRan**

**| |-- Signal all threads using condition**

**|**

**|-- Main Function**

**| |-- Prompt for nProcesses, nResources**

**| |-- Allocate memory for resources, allocated, maxRequired, need, safeSeq**

**| |-- Prompt for currently available resources**

**| |-- Prompt for resource allocation and maximum requirements**

**| |-- Calculate need matrix**

**| |-- Call GetSafeSeq():**

**| |-- If return value is false:**

**| |-- Print "Unsafe State!" and exit**

**| |-- Print safe sequence**

**| |-- Create threads for processes**

**| |-- Join all threads**

**| |-- Print "All Processes Finished"**

**| |-- Free allocated memory**

**|**

**End**

1. **GitHub link**

The GitHub link has been attached for the code and the test cases.

[**https://github.com/swayam-learning/Project\_OS.git**](https://github.com/swayam-learning/Project_OS.git)

1. **OUTPUT**

**A black and purple background

Description automatically generated with medium confidence**

A black and purple background

Description automatically generated with medium confidence