**OS Assignment**

**Problem statement**: 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.

**Description:** Herewe haveto write a multithreaded program that implements the banker's algorithm first let’s see what’s a **multithreaded program** is, a multithreaded program or application are those that run more than one thread in its process. Which, in the real world, are, basically, all out there. Now let’s talk about, **what is Banker’s Algorithm**? The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state”(safe-state) check to test for possible activities, before deciding whether allocation should be allowed to continue.  It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Here the banker will grant the request only if the request leaves the system in a safe state, a state is safe if the system can allocate all resources by all the processes (up to their stated maximums) without entering a deadlock state. More normally, a state is safe if there exists a ***safe sequence*** of processes {P0, P1, P2… PN} such that all of the resource requests for Pi can be granted using the resources currently allocated Pi and all the processes P­j where j < i. If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe state are deadlock free, but not all unsafe states lead to deadlock.)

It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

**Algorithm: Can be opened by double clicking.**

Following Data structures are used to implement the Banker’s Algorithm:

Let ‘n’ be the number of processes in the system and ‘m’ be the number of resources types.

**Available:**

* It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
* Available[j] = k means there are **‘k’** instances of resource type **Rj.**

**Max:**

* It is a 2-d array of size **‘n\*m’**that defines the maximum demand of each process in a system.
* Max[i, j] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj.**

**Allocation:**

* It is a 2-d array of size**‘n\*m’**that defines the number of resources of each type currently allocated to each process.
* Allocation[ i, j ] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj**

**Need:**

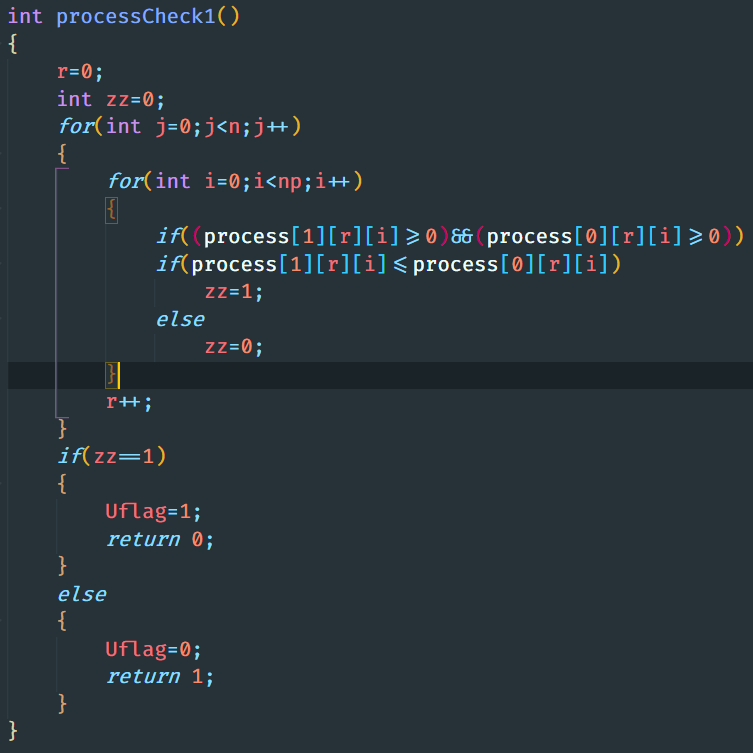
* It is a 2-d array of size **‘n\*m’** that indicates the remaining resource need of each process.
* Need [ i, j ] = k means process **Pi** currently allocated **‘k’** instances of resource type **Rj**
* Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

**Description (purpose of use):** Mostly the complexity of each line of code in the algorithm is O(n2) and the overall complexity of the program is O(n3).

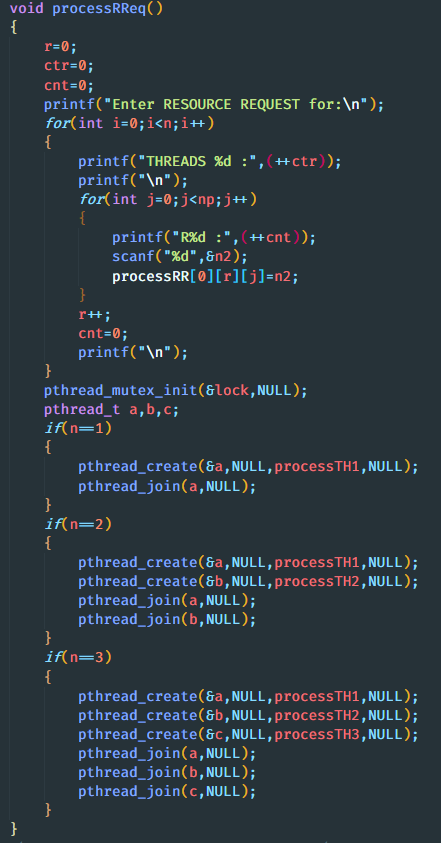
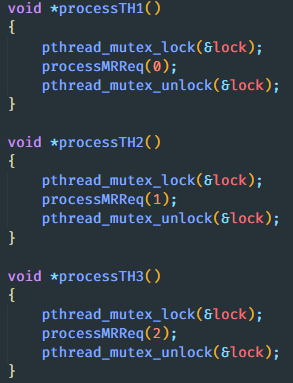
**Code snippet (All the constraints):** Our main constraint here is to ensure the safeness of the shared data which can be fulfilled by the **SAFETY ALGORITHM**

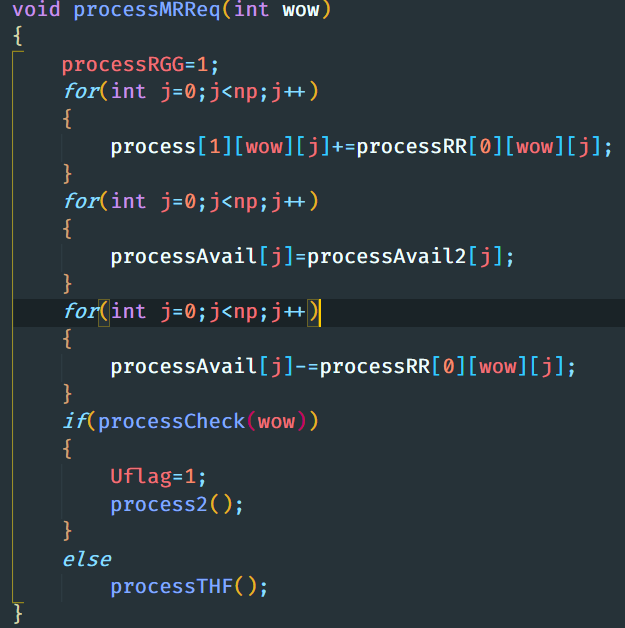
* In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
* This algorithm determines if the current state of a system is safe, according to the following steps:

1. Let Work and Finish be vectors of length m and n respectively.
   1. Work is a working copy of the available resources, which will be modified during the analysis.
   2. Finish is a vector of Booleans indicating whether a particular process can finish. ( or has finished so far in the analysis. )
   3. Initialize Work to Available, and Finish to false for all elements.
2. Find an index (i) such that both (A) Finish[ i ] == false, and (B) Need[ i ] < Work. This process has not finished, but could with the given available working set. If no such index (i) exists, go to step 4.
3. Set Work = Work + Allocation[ i ], and set Finish[ i ] to true. This corresponds to process index (i) finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[ i ] == true for all index (i), then the state is a safe state, because a safe sequence has been found.



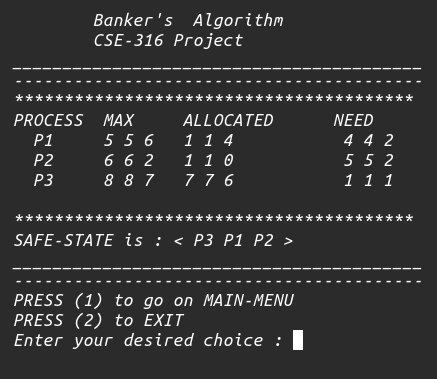
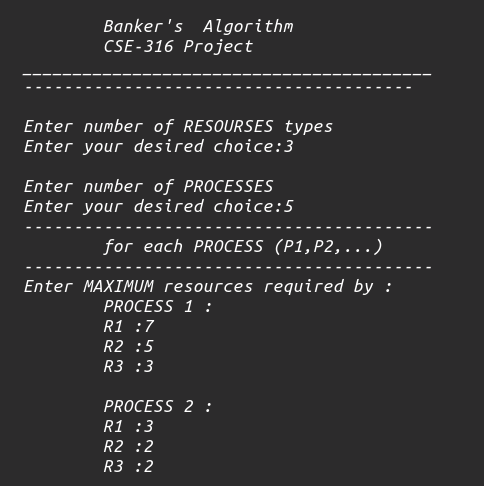
Here is the implementation of safety algorithm.

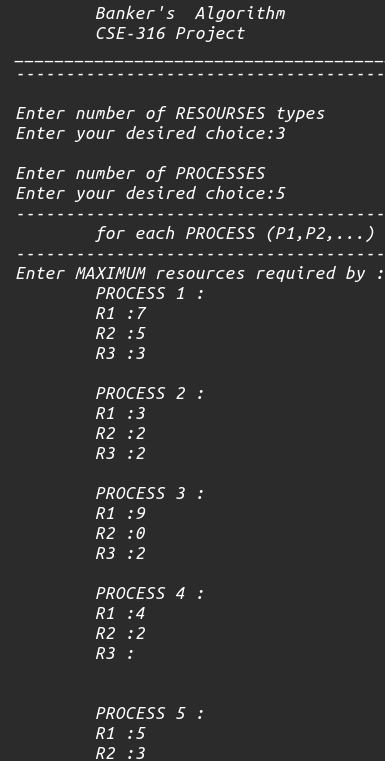
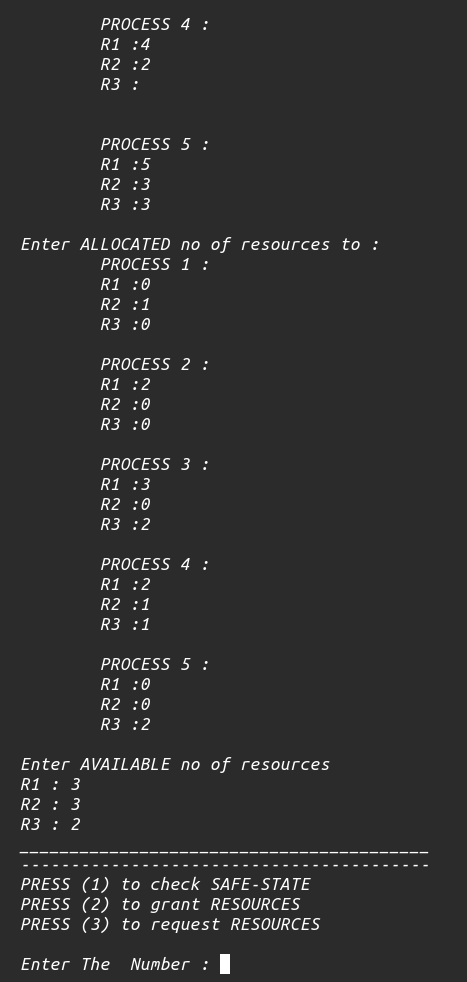
 

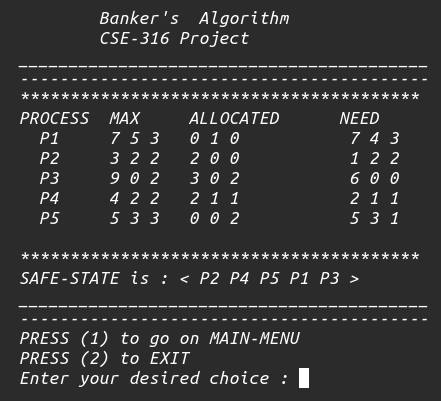
This is the implementation code of the mutex locks. It is code for processMRReq(). Hence all constraint satisfied.

**Boundary Condition:** Here the boundary condition of the code is when the Need[i] <= work which is when need of the resources should be less than the available resources if not then the process or thread will be put to **sleep** otherwise the process will allocate the resources.

**Test Cases (Description):**

** **

** **

****