Problem 3: Bank Simulation:

1. **What are semaphores and what is its usefulness? What type are the synchronization problems that a semaphore may resolve but the simple lock cannot?**

A semaphore is a counter that can be manipulated by two types of operations (wait and signal) and is used to solve a synchronization problem. The first operation is the *down or wait* that decreases the counter and the second operation is referred to as the *up, signal, or post* where the counter is increased. A semaphore is initialized depending on the type of problem the semaphore is being used to solve. Lets consider the case if a semaphore is initialized to a value of one. The first thread that wants to enter the critical section calls wait is able to go right through without having to wait. This changes the semaphore value from one to zero. Now, if another thread calls the wait operation, the thread will be blocked and added to the waiting queue. After the first thread finishes using the critical section, it will call signal and wake one of the threads in the waiting queue without increasing the semaphore value. If the second thread calls signal the semaphore value increases from 0 to 1. If there are no threads in the waiting queue, then the next thread that calls wait would not have to wait in the queue and can go through because the signal is remembered. If a semaphore is initialized to values greater than one, then more threads that call wait can go through without actually having to wait. For example, if the semaphore value is initialized to two, the first thread that calls wait will go through and the semaphore value will go from 2 to 1. Similarly, the next thread that calls the wait operation will continue and the semaphore value will change from 1 to 0. When the third thread calls the wait operation, then it is blocked because the semaphore value is 0. It can be thought of as how many threads one want to hold the semaphore at one time. Semaphores are useful for allocating a number of resources. For example, if there are a specific number of shared buffers that threads can use, a thread can call wait to see if a shared buffer is available for it to use. If there are no shared buffers left, the semaphore will be zero and the wait operation will block. As soon as a thread has finished with the shared buffer and it becomes available, it posts the semaphore it either increases the value of the counter or allows the blocked thread to use it.

A semaphore resolves some problems that a simple lock cannot. A lock emphasizes mutually exclusive access. If a thread wants to use a shared data object, it must lock the mutex and then unlock after it so other threads can use the shared data object. Since only one thread is able to access the shared data object at a time, the other threads that want to use it have to wait until it is unlocked. A semaphore allows more than one thread to enter the critical section. A semaphore can be a binary semaphore or a counting semaphore. A lock can only unlocked by the thread that locked it and it does not have to be initialized. In semaphore, there is use of wait and signal operations, and it needs to be initialized, and gives more threads access to critical sections at one time depending on the initialized value, unlike a lock.

1. **Provide the definition of a semaphore by Diikstra. The implementation based on this definition leads to the issue of “busy waiting.” What is this problem and why is it undesirable?**

Dijkstra defined a semaphore as a variable whose value can be manipulated by the two operations P and V. P represented the wait or down operation and V is the signal or up operation. Dijkstra’s definition considered a semaphore to be atomic. This means that once the operations of a semaphore have begun, other processes cannot access them until they are completed. He also thought that one process can only continue at once and the other processes would have to wait. One issue that semaphores cause is the “busy waiting” problem. “Busy waiting” is when processes that are waiting on semaphores, they have to continuously check if the semaphores values are zero or not. This causes a problem because it utilizes unnecessary CPU cycles and is referred to as spinlock.

1. **Provide an implementation of the semaphore that avoids the busy wait. In which places is now the waiting restricted?**

wait(Semaphore s){

s=s-1; **if** (s<0) { *// add process to queue* block(); } }

signal(Semaphore s){ s=s+1; **if** (s>=0) { *// remove process p from queue* wakeup(p); } }

Init(Semaphore s , Int v){ s=v; }