Name: Rakesh Kannan
Student ID: 200535445

4. Implementation Approach for Active Lock System with Deadlock Detection
The active_lock.c file implements a locking system for managing lock acquisition and release while detecting potential deadlocks among processes.
Core Functions:
Deadlock Detection (check_deadlock):
- Detects circular dependencies by tracking each process's locking chain and checking for cycles.
- Tracks involved processes and sorts them to provide a clear deadlock chain, printing the sequence as deadlock_detected=P1-P2-....
- Each time a process requests a lock already held by another process, check_deadlock verifies if a cycle exists, signaling a deadlock.

Function Implementations
1. Lock Initialization (*al_initlock*):
- Sets up a lock with an ID and initializes its guard and flag fields.
- Limits total active locks to a predefined maximum (NALOCKS), ensuring system resources are managed.

2. Lock Acquisition (*al_lock*):
- Uses test_and_set to acquire the lock guard in a non-blocking way.
- If the lock is free, the requesting process immediately gains it, updates the lock's state, and logs acquisition.
- If the lock is already held, al_lock:
  - Checks for potential deadlocks.
  - Parks the requesting process, adding it to the lock's queue until it can safely acquire the lock.

3. Lock Release (*al_unlock*):
- Releases a lock and either frees it or passes it to the next waiting process in the queue.
- Ensures lock states are correctly updated, and waiting processes are unparked in a priority order.

4. Try Lock (*al_trylock*):
- A non-blocking attempt to acquire the lock that returns immediately.
- Returns TRUE if the lock is acquired and FALSE if another process holds the lock.

5. Park and Unpark (*al_park* and *al_unpark*):
- Manages the parked (waiting) state of a process and restores it when it is ready to proceed.
- al_setpark and al_park put processes into a waiting state until they can safely acquire a lock, and al_unpark reactivates them when the lock becomes available.

Deadlock Detection (check_deadlock):
The *check_deadlock* function detects circular dependencies by examining each process lock acquisition chain for cycles. When a process requests a lock that's already held, *check_deadlock* is called to verify if

the new request creates a circular dependency. If a cycle is detected, this indicates a deadlock, and the function immediately notifies the user by printing the involved processes in ascending order of process IDs (e.g.,*deadlock_detected=P1-P2-..*). This proactive, real-time deadlock detection ensures that any circular dependencies are identified promptly upon lock acquisition attempts, allowing users to see which processes are involved in the deadlock.

5. Implementation Approach for Testing Deadlock Prevention Methods
This implementation explores four methods to test for deadlock detection and prevention. The primary objective is to verify how well the locking mechanism handles potential deadlocks in different scenarios.

Part 1 – Trigger Deadlock Detection
<u>Process Creation and Locking:</u>
Each process is created with the *trigger_deadlock* function as its entry point, which requires two lock parameters to simulate staggered execution times.
Processes are configured to acquire two locks each in a circular dependency, creating a potential deadlock.
For instance:

- Process 1 acquires Lock A then Lock B.
- Process 2 acquires Lock B then Lock C.
- Process 3 acquires Lock C then Lock A.

<u>Execution and Verification:</u>
Each process is resumed, attempting to lock its respective locks sequentially. This configuration forces the processes into a deadlock.
Deadlock detection is verified if a notification appears, identifying the deadlocked process chain.

Part 2 – Prevent Deadlock by Avoiding Hold-and-Wait
<u>Process Creation with avoid_hold_wait:</u>
Each process attempts to acquire its first lock with a retry mechanism using *al_trylock*.
If the second lock is unavailable, the first lock is released, preventing a hold-and-wait scenario.
<u>Execution and Verification:</u>
This method prevents deadlock by avoiding holding a lock while waiting for another. Each process retries lock acquisition in a non-blocking manner.
Successful execution without deadlock confirms the effectiveness of this method.

Part 3 – Allow Preemption to Avoid Deadlock
<u>Process Creation and Lock Preemption:</u>
The *allow_preemption* function lets processes acquire their first lock, but preemptively release it before acquiring the second lock.
This approach allows other processes to gain access to the first lock, preventing a circular dependency.
<u>Execution and Verification:</u>
Processes release locks in intervals, giving each one a chance to progress and thereby avoiding deadlock.
Successful execution without deadlock detection confirms that preemption helps prevent deadlock.

Part 4 – Enforce Strict Lock Ordering to Avoid Circular Wait
Lock Ordering Mechanism:
Each process acquires locks in a predetermined order using the *avoid_circular_wait* function.
Locks are always acquired based on an ascending order of their identifiers, ensuring that no process holds a lower-ordered lock while waiting for a higher-ordered one.
Execution and Verification:
By enforcing a strict order, circular wait conditions are avoided, thereby preventing deadlock.
Processes proceed without entering a deadlock, confirming that lock ordering effectively prevents deadlock.

6. Implementation Approach for Priority Inheritance Lock system to avoid Priority Inversion
Priority Inheritance Lock (pi_lock.c)
This code implements a priority inheritance lock to mitigate priority inversion, allowing high-priority processes to access resources held by lower-priority processes without unnecessary delays.

Initialization (*pi_initlock*):
Sets up a lock with default flags, initializes a guard, and creates a wait queue. A lock count (pi_count) limits the number of active locks.

Lock Acquisition (*pi_lock*):
If the lock is free, it is acquired immediately by the current process.
If the lock is held, the requesting process is enqueued and parked, and priority inheritance is applied to elevate the lock-holder's priority if a higher-priority process is waiting.

Lock Release (*pi_unlock*):
Releases the lock if no processes are waiting.
If processes are in the queue, it unparks the next waiting process, transferring the lock and adjusting priorities as needed.

Priority Inheritance (*update_priority*):
Temporarily raises the lock-holder's priority to match any higher-priority waiting process, preventing priority inversion.

Priority Restoration (*restore_inheritance*):
When the lock is released, this function reverts the priority of the lock-holder to its original state or to the highest priority of any remaining dependent processes.

Try-Lock (*pi_trylock*):
Attempts immediate lock acquisition without blocking. Returns success or failure based on lock availability.

The lock system dynamically adjusts priorities to prevent blocking high-priority tasks on lower-priority ones, ensuring fair and efficient resource access.