NC State University

Department of Electrical and Computer Engineering

ECE 565: Fall 2023

Project 3

by

RAGHUL SRINIVASAN

UnityID: rsriniv7

(200483357)

Part 4 (Deadlock Detection)

Active Lock Structure (al_lock_t):

- **flag:** A boolean flag indicating whether the lock is currently acquired.
- **guard:** A boolean guard flag to protect critical sections of code.
- **q:** A queue to hold processes waiting for the lock.
- **pid:** Process ID of the process holding the lock.

Lock Initialization (al_initlock):

- Initializes the active lock structure.
- Ensures that the maximum number of active locks (NALOCKS) is not exceeded.

Lock Acquisition (al_lock):

- Implements a spinlock (test_and_set) for the guard to prevent race conditions.
- If the lock is not acquired (flag is FALSE), sets the flag, assigns the current process as the lock holder (pid), and releases the guard.
- If the lock is already acquired, enqueues the current process, parks it, and releases the guard.

Lock Release (al_unlock):

- Implements a spinlock for the guard.
- If the waiting queue is empty, releases the lock (flag is set to FALSE).
- If the waiting queue is not empty, dequeues a process, unparks it, and releases the guard.

Lock Attempt (al_trylock):

- Attempts to acquire the lock without waiting.
- Returns TRUE if successful; otherwise, returns FALSE.

Park and Unpark (al_park, al_unpark):

- **al_park:** Parks a process, setting its state to PR_PARK and invoking deadlock detection.
- **al_unpark:** Unparks a process, setting its state to PR_READY and placing it in the ready list.

Deadlock Detection (deadlock_detection):

***Deadlock Tree Array (deadlock_tree)*:**

- It is an array of size NPROC used to store the process IDs in the order of their acquisition of locks.
- The array is initialized with zeros.

Loop Iterations:

- The outer loop (LoopIteration) iterates over all possible process IDs in the system (NPROC).
- The inner loop (LoopIteration1) checks for cycles in the acquisition chain.

Process ID Assignment:

- In each iteration of the outer loop, the current process ID (pid) is assigned to the deadlock_tree array.

Deadlock Flag Check:

- Checks if the current process has already been marked with the deadlock_flag. If not, proceeds with cycle detection.

Cycle Detection:

- The inner loop checks if the current process ID (pid) is already present in the deadlock_tree array.
- If a cycle is detected, it prints a message indicating a deadlock, sorts and prints the involved processes, and returns.

***Updating Process ID for Next Iteration*:**

- Updates the process ID (pid) to be the process ID of the process that acquired the lock (al_lock_acquired_pid).
- If the -out code within the deadlock detection section seems to have been an alternative approach to marking processes involved in the deadlock.

- The function uses a simple array (deadlock_tree) to track the acquisition chain, and the outer loop covers all possible processes.
- The inner loop checks for cycles by comparing the current process ID with those already in the array.
- When a cycle is detected, it prints a message and marks the processes involved with the deadlock_flag.
- The commented-out code within the deadlock detection section seems to have been an alternative approach to marking processes involved in the deadlock.

Part 5 (Testcase for deadlock detection)

Test Case 1: Deadlock Detection for 3 Threads
- Three processes (tc1_Deadlock) attempt to acquire locks in a circular dependency, leading to a potential deadlock.
- The deadlock_detection function is used to detect the deadlock.
- Deadlock is detected with deadlock_flag in PCB.

Test Case 2: Deadlock Prevention by Avoiding Hold-and-Wait
- Three processes (tc2_HoldAndWait) attempt to acquire locks but do not hold a lock while waiting for another.
- The mainmutex is introduced to ensure that a process must acquire the main mutex before acquiring the other locks.
- This prevents the hold-and-wait condition, avoiding a potential deadlock.

Test Case 3: Deadlock Prevention by Preemption of the Thread Owning the Lock
- Three processes (tc3_Preemption) use al_trylock to attempt lock acquisition and handle preemption if the lock is not available.
- This mechanism prevents a process from holding a lock indefinitely while waiting for another, avoiding a potential deadlock.

Test Case 4: Deadlock Prevention by Avoiding Circular Wait
- Three processes (tc4_CircularWait) attempt to acquire locks in a circular dependency.
- The order of lock acquisition is determined to prevent circular wait conditions.
- This prevents the circular wait condition, avoiding a potential deadlock.

VALIDATION:
- SharedVariable is incremented by each process in critical section after acquiring two locks.
- It should be incremented by 3 (1 by each process) after Testcase2, testcase3 and testcase4.
- SharedVariable should be equal to 3,6, 9 after Testcase2, testcase3, testcase4 respectively as SharedVariable is not reset to 0 after the end of each testcase.

Part 6 (Priority Inversion)

Lock Initialization (pi_initlock):
- Initializes a priority inheritance lock.
- Tracks the number of locks (pi_lock_t_count) to prevent exceeding the maximum allowed locks (NPILOCKS).
- Uses a flag, guard, and a queue for locking and unlocking processes.

Lock Acquisition (pi_lock):
- Uses a guard to avoid race conditions during lock acquisition.
- If the lock is not held, it is acquired, and the process is set as the lock owner.
- If the lock is held, the process is enqueued and parked until the lock is available.

Lock Release (pi_unlock):
- Uses a guard to avoid race conditions during lock release.
- If the lock queue is empty, the lock is released, and the guard is cleared.
- If the lock queue is not empty, a process is dequeued, unparked, and its priority is potentially upgraded.

Priority Inheritance (pi_setpark and pi_park):
- pi_setpark sets the park flag for a process.
- pi_park checks the park flag and upgrades the priority of the waiting process to the priority of the lock owner.
- The waiting process is enqueued with its original priority, and the park flag is cleared.

Priority Downgrade (pi_unpark, priority_downgrade):
- pi_unpark unparks a process and downgrades its priority based on priority inheritance.
- priority_downgrade handles the priority downgrade logic for a process and other potentially affected processes.

Priority Upgrade (priority_upgrade):
- Handles priority upgrades for a process based on the priority of the process holding the lock.
- Iterates through the chain of processes holding the lock, upgrading priorities accordingly.

Priority Modification (readylist_modify):
- Modifies the ready list to ensure proper ordering after a process's priority is changed.