3 Exercise #3: Mutual Exclusion and Deadlocks

In this exercise you will implement different variants of mutexes in LWT. These primitives will be able to prevent or detect deadlocks and to be used recursively. Monitors and semaphores will be implemented with the help of different mutex variants.

3.1 Mutex and Condition Variable

Implement simple mutexes and condition variables in LWT. For this first task you can ignore the `config` parameter to `lwt_mutex_init` and the `lwt_mutex_id` function. See the `lwt_mutex...` and `lwt_cond...` functions documentation for more details.

3.2 Recursive Mutex

Extend your mutex implementation in a configurable manner to allow for recursive use by its holder. It should be possible to wait on a condition variable while being in a deep 'recursion'. The `config` parameter to `lwt_mutex_init` indicates which type of mutex it creates.

3.3 Deadlock Prevention

One possibility to prevent deadlocks is to enumerate all locks and to force threads to acquire locks in the increasing order defined by their enumeration. Implement this deadlock-prevention strategy for your mutex implementation. Insert checks to prevent threads form locking and unlocking in the wrong order. Make this change configurable, too. This deadlock prevention strategy should also work with recursive mutexes (`ORDERED` and `RECURSIVE` flags).

3.4 Deadlock Detection

Sometimes it is not convenient to use a deadlock-prevention strategy. In this case deadlock detection can help debugging parallel code that causes deadlocks. Keep track of the holder of a mutex and on which mutex a thread waits. When a mutex is already locked, check if a deadlock would happen when the current thread would wait on the mutex. Follow the path of the mutex-holding thread and check if it also waits on a different mutex and so on. If a cycle is detected, print this cycle and the participating threads and monitors in-order and exit the program. Note that the mutex is also locked again if a thread wakes up from a condition variable. To prevent false positives, detecting deadlocks must be atomic relative to other threads performing operations on mutexes. Deadlock detection should be also a configurable feature of the mutex implementation and should also work with recursive mutexes.

3.5 Recursive Monitor

Use the LWT mutex and condition variable functions to implement a recursive monitor type. Write a `lwt-monitor.h` header and design your own monitor interface functions. Monitors should have functions to **enter** and **leave** a critical section. Also, a thread should be able to **wait** on a monitor object that it currently holds until is is **signal**ed again. Threads waiting on a monitor should release the critical section. Only the holder of a monitor should be able to leave it again and to perform wait or signal operations. Again, it should be possible to wait on the monitor while being in a deep recursion. (Hint: once you have the mutex and condition variable types in place, this is a very simple task);

3.6 Semaphore

Use the LWT mutex and condition variable functions to implement a counting semaphore type. Semaphores should not be used recursively and different threads will use the `signal` and `wait` functions, see `lwt_sem...` functions. Show how you can implement a simple mutex and condition variable using semaphores. What could be the drawbacks of implementing one blocking type with the help of the other? What are the advantages?

3.7 Tests

Write tests for your implementations and configurations. Show that deadlock prevention and detection work by detecting faulty programs.