4 Exercise #4: Monitors and Deadlocks

In this exercise you will implement different variants of monitors to be used as a library (in contrast to language support). These monitors will be able to prevent or detect deadlocks and to be used recursively.

4.1 Simple Monitor

Use the Pthread library to implement a simple monitor type. These 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 a wait or signal operations.

4.2 Recursive Monitor

Extend your monitor implementation, in a configurable manner, to allow for recursive use by its holder. It should be possible to wait on the monitor while being in a deep recursion. Note that you cannot use the `PTHREAD_MUTEX_RECURSIVE` attribute for `pthread_mutex_t` as the POSIX specification explicitly states that waiting on a condition variable may not completely unlock the mutex (see the `pthread_mutexattr_settype` manpage).

4.3 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 monitors. Extend the monitor interface with a function to return a unique id for a specific monitor object. Insert checks in the enter and leave function to prevent threads form locking and unlocking in the wrong order. Make this change to the monitor implementation configurable, too. This deadlock prevention strategy should also work with recursive monitors.

4.4 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 monitor and on which monitor a thread waits. When a monitor is already locked, check if a deadlock would happen when the current thread would wait on the monitor. Follow the path of the monitor-holding thread and check if it also waits on a different monitor 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 monitor is also locked again if a thread wakes up from a wait operation. To prevent false positives, detecting deadlocks must be atomic relative to other threads performing operations on monitors. Deadlock detection should be also a configurable feature of the monitor implementation and should also work with recursive monitors.

4.5 Tests

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

4.6 Submit

Submit your solutions by creating the directory /proj/i4cs/students/`your_login`/assignment4/ All files in this folder will be collected after the submission deadline. The file `comments.txt` will be created in this directory and contains comments from the tutors. Please create a file `group.txt` with your and your partner's login if you do your assignments in a group of 2 people.

Remarks:

- It is advisable to encapsulate the Pthread functions, types and the needed error-checking and handling.
- Thread-local variables can be created by either compiler support using `__thread` variables, or the Pthread functions `pthread_setspecific` and `pthread_getspecific`.

Submit until: 2015-01-13