3 Exercise #3: Mutual Exclusion

In this exercise you will implement various algorithms for mutual exclusion. You will also evaluate them under varying conditions to find out which one is the best solution in which case.

3.1 Spinlocks

Implement a library with multiple spin-lock types. Use `C/C++` atomic types and operations where necessary. Make sure that all lock enter and leave operations provide at least acquire/release consistency. Implement at least the following lock algorithms:

- The most simple spin-lock uses the `atomic_flag` type. The lock uses an atomic `TAS` operation to gain exclusive access to a critical section. The lock performs this atomic operations repeatedly, until it finally succeeds. Combine this lock with multiple back-off strategies.
- The `CAS` operation can be used for a spin-lock that only tries to enter the critical section when it appears to be free. As long as the lock is taken, this lock does not try to modify lock variables. This optimization reduces cache contention. Again, combine this lock with multiple back-off strategies.
- Ticket spin-locks guarantee starvation-freedom for the threads contending on the lock. A FIFO order of the threads is ensured. Experiment by placing member variables of this lock on different cache lines.
- The lecture presents lock algorithms that only rely on atomic load and store algorithms. Implement at least one of those algorithms. Make sure your implementation supports more than two threads.
- At least one 'scalable lock' of MCS or K42 type. A good summary of the algorithms can be found online[1] but you can also read the original paper[2] for MCS locks. Note that the K42 lock has only been published through a US patent.

Write micro-benchmarks to compare the different spin-lock types under high and low contention. What happens when there are more threads than CPUs? Why? What effect does the back-off strategy have on lock performance? You can also compare the lock performance to `pthread_mutex` in your benchmarks.

3.2 Validate a Lock Algorithm

Use `CDSChecker`[3] to validate at least one lock algorithm. Implement a suitable unit test and compile it with the `CDSChecker` headers and library. Add assertions to validate that two threads can never be inside a critical section simultaneously. Make sure you keep the test case simple to make the validation process fast.

3.3 Actor Library

Implement a simple actor library. Actors are entities that send each other requests asynchronously. Each actor is associated to a worker thread that handles these requests. To manage data dependencies, the worker thread signals termination of requests.
Actors simplify synchronization because the worker threads inherently serialize requests. As long as only one actor can access a variable, no race conditions occur. Furthermore, worker threads are often independent and can thus increase the amount of parallelism in the application.
Write a test case for your actor implementation. Also evaluate the actor library and compare it to lock algorithms. Under which condition is an actor better than a lock?

Remarks:

- Intel suggests placing a `pause` instruction in busy loops to mark them as such (for the instruction fetch of the CPU pipeline). You can also experiment with this instruction, use the gcc intrinsic function `__builtin_ia32_pause()`[4].
- You can find documentation for the `C/C++` atomic standard library online[5].
- To verify code with a busy loop, call `thrd_yield()` in the loop body and pass the `-y` flag to `CDSChecker`.
- The `CDSChecker` is installed in the CIP pools in `/local/cdschecker/`.
- Submission deadline: 2018-12-17, 14:00

[1] `https://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html`
[2] `http://dl.acm.org/citation.cfm?id=106999`
[3] `http://plrg.eecs.uci.edu/software_page/42-2/`
[4] `https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html`
[5] `http://en.cppreference.com/w/c/atomic` and `http://en.cppreference.com/w/cpp/atomic`