3 Exercise #3: Locks

For this exercise you will implement different (busy) lock types. You will measure their performance in contended and uncontended cases.

3.1 Special Instruction Locks

Implement the following locks based on special CPU instructions. To use these special instructions, you can write assembly code or use intrinsic functions of your compiler[1].

- The most simple spinlock that uses an unconditional swap instruction to gain exclusive access to a critical section. The lock should perform this atomic swap as long as it does not succeed. On x86 the respective instruction is called `xchg` and is always atomic (also without a lock prefix).
- A read-spinlock that is very similar to the previous lock but reads the lock variable in a loop and only performs the atomic swap operation if the previous read suggests a possible success.
- Ticket spinlocks 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.
- Ticket spinlocks are well suited for backoff strategies, as each thread can calculate how many threads acquire the lock before it can succeed. Create another version of the ticket spinlock with at least one backoff strategy of your choice.

Feel free to implement more variations and combinations as you like.

3.2 Atomic Read/Write Locks

Read/write locks just assume atomic read and write access to variables to implement exclusive access, but require a previously known maximum number of threads. Be aware that especially for these type of locks unwanted reordering by the compiler and CPU may happen. Implement at least **two** the following algorithms for a configurable number $N \geq 2$ of threads:

- Dekker's algorithm (found in "Dijkstra: Cooperating Sequential Processes")
- Peterson's algorithm
- Kessels' algorithm
- Lamport's bakery algorithm

You will find all related papers in the materials to this assignment. Again if you find other algorithms feel free to implement them.

3.3 Test and Measure

For testing and measuring the implementations, use the `SimpleQueue` provided for the previous assignment. Synchronize its `enqueue` and `dequeue` operations with the locks you implemented. We will implement a non-blocking variant of this queue in the last exercise and compare it to the locked variants.
Write a test program that enqueues and dequeues in parallel with multiple threads. Verify that no item was lost or dequeued twice.
Write a benchmark for the contended case, increase the number of threads step by step and measure the execution time of the whole benchmark run. Pin (see remarks) the participating threads to CPUs in round robin fashion to see effects of the caching and memory hardware (especially interesting on `faui49big01`). Allocate all necessary memory before you start measuring time, else you most probably measure the performance of your memory allocator instead of the locking algorithms.
For the uncontended case just measure the time a successful lock-unlock pair takes. For this purpose an implementation that uses the x86 time stamp counter (TSC) will be provided in the materials. Do not forget to pin the thread that performs this measurement because the TSC is CPU local. Remember to do multiple measurements to compensate for fluctuations. You can filter out very large values that may come from an operating-system context-switch.
Plot interesting results and submit these plots in a single PDF file.

[1] `https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/_005f_005fsync-Builtins.html`

3.4 Submit

Submit your solutions by creating the directory /proj/i4cs/students/`your_login`/assignment3/ 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:

- For pinning threads to CPUs see the manpages: `sched_setaffinity` and `CPU_SET`. To determine the number of CPUs you can use `sysconf(_SC_NPROCESSORS_ONLN)`.
- You can also measure the performance of `pthread_mutex` in your benchmarks.
- 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: `asm volatile("pause\n\t");`.

Submit until: 2014-12-09