Theory & Practice of Concurrent Programming (COMP60007)
Tutorial 2: Implementing Synchronisation Primitives

An archive of skeleton code associated with this tutorial is available online. Files referred to below are in the `src` directory of this archive.

Questions 1–5 involve writing additional spinlock implementations in the framework presented during the lectures. Under `src/spinlocks` you will find the code for the spinlocks studied during the lectures, together with skeleton header files for a number of additional locks.

Remember that the `check_build.sh` script builds various versions of your code. It is recommended that you test for *correctness* using the executables under `temp/build-tsan`, and for *performance* using those under `temp/build-release`.

1. The test-and-set locks that we studied in the lectures all use a *single* atomic memory location. The relaxed memory ordering ensure sequential consistency per location. If we change the memory orderings associated with all atomic operations in a test-and-set implementation, then:

   - Does it become possible for more than one thread to believe that it holds the lock?
   - Can the lock still be used to avoid races on shared data?

   Investigate this in practice by filling out the `SpinLockSimpleRelaxed` class and selecting it via the `simple_relaxed` argument to `demo_spinlocks`.

2. In the `SpinLockActiveBackoffWeakerOrderings` class, adapt the test-and-set lock with active backoff studied in the lectures so that its atomic operations use the weakest possible memory orders that you believe are valid.

   Argue why these memory orders suffice to ensure that the lock can be used to ensure race-free access to lock-protected data.

   Via the `active_backoff_weaker_orderings` argument to `demo_spinlocks`, compare the performance of your adapted lock compared with the original version using the *performance* benchmark shown in the lectures. Do you observe performance differences?

   Use the `-S` compiler flag, or the online Compiler Explorer, to investigate differences in the assembly code for the original lock implementation and the implementation that uses more relaxed orderings. Are there any differences? Use the Compiler Explorer to investigate this with respect to both x86 and ARM V8 assembly.

3. Consider the ticket lock implementation shown in the lectures.

   If you make the `now_serving_` field non-atomic (rewriting operations on the field in the obvious way) then is the ticket lock implementation correct in principle? Either way, does it appear work correctly on your system in practice?

   Try it out in practice in the `SpinLockTicketNonAtomic` class, which you can select via the `ticket_nonatomic` argument to `demo_spinlocks`.

   Does removing the `_mm_pause()` instruction make a difference to your results?

   If you make the `now_serving_` field non-atomic but declare it `volatile`, does this have any impact on the correctness of the lock implementation in principle, and/or whether it appears to work correctly in practice?

Try it out in practice in the `SpinLockTicketVolatile` class, which you can select via the `ticket_volatile` argument to `demo_spinlocks`.

4. The ticket lock implementation shown in the lectures uses `_mm_pause()` so that threads back off briefly between iterations while spinning on the value of `now_serving_`.

   Can you think of a smarter way to decide for how many iterations an `_mm_pause()` instruction should be executed? If you have access to it, a smarter method can be found in Section 4.2.2 of *Shared Memory Synchronization* by Michael L. Scott.

   Try it out in practice in the `SpinLockTicketOptimised` class, which you can select via the `ticket_optimised` argument to `demo_spinlocks`.

5. Implement two C++ version of the ALock array-based queue lock presented in Section 7.5.1 of *The Art of Multiprocessor Programming*.

   First read about the problem of *false* sharing that the book describes.

   The first version of your ALock should (for the sake of illustration) *maximise* false sharing. Implement this in `SpinLockALockFalseSharing`. The second should use padding to minimise false sharing—implement this in `SpinLockALockPadded`.

   Hints:

   - The ALock requires *thread-local storage*, so look online for details of how this works in C++. Unfortunately, C++ thread-local storage is *static*, so it is not easy to have multiple instances of a class such that every thread has separate local storage per instance of the class. Don't worry about this problem for the current exercise: just use a static thread-local field–this means that it is only valid to have one ALock instance alive at a time. When you have declared a static field `foo` of type `T` in a class `A`, it is necessary to include a definition of the form `T A::foo;` outside the class `A`. Normally this would be in the `.cc` file containing the implementation of `A`. Since your spinlocks are implemented purely in header files, feel free to place the additional definitions required for static thread-local storage fields in `src/spinlocks/demo_spinlocks.cc` (this is not good practice, but it doesn't matter for this tutorial).
   - To achieve padding, consider using a struct annotated with the `__attribute__((aligned(N)))`. This indicates that the memory associated with an instance of the struct should always start at an address that is a multiple of `N` bytes.

   Using the `alock_false_sharing` and `alock_padded` arguments to `demo_spinlocks`, benchmark the performance and fairness of these lock implementations, both against each other and against the lock implementations studied in the lectures. How do they compare? Does padding lead to a performance win over the version that suffers from false sharing?

   If you have your own machine and are willing to enable the necessary privileges to run the `perf` tool then try using `perf` to investigate the cache-related behaviour of your ALock implementations.

   What feature of ALocks means that they are unlikely to be useful as general-purpose locks?

6. Thanks to Prof Tyler Sorensen at University of California Santa Cruz (and my former PhD student at Imperial!) for suggesting Questions 6 and 7, which is similar to one that he uses in his concurrency course.

Take a look at the `RepeatedBlurSequential` function in `src/repeated_blur/demo_repeated_blur.cc`. This function takes an array of doubles and iterates over it `num_blur_iterations` times. On each iteration the vector is *blurred* by replacing element i with the mean of elements $i-1$, i and $i+1$ (the first and last values of the vector are left unchanged).

There is scope for parallelising a single blur pass: if the result of the blur is written into a different vector then multiple threads can blur chunks of the vector simultaneously.

Although it is sequential, the `RepeatedBlurSequential` has been written with this in mind: a temporary buffer is declared at the top of the function, and then two pointers, `current` and `next` are maintained so that on each loop iteration the contents of the vector pointed to by `current` are blurred into the vector pointed to by `next`. At the end of the function there is some logic to ensure that `data` holds the fully blurred result.

Unfortunately, there is *not* scope for executing distinct blur iterations in parallel: the second blur iteration depends on the result of the first blur iteration.

Your task is to implement a parallel version of repeated blurring by filling out the `RepeatedBlurForkJoin` function.

You may assume that `num_threads` divides the size of `data`. Your implementation should loop `num_blur_iterations` times. Like the `RepeatedBlurSequential` implementation, this function should ping pong data between two buffers on successive blurring iterations. On each blur iteration it should launch `num_threads` threads, each of which should take care of blurring a chunk of `current` into `next`. At the end of each iteration the threads should be joined. That is, a fresh set of threads should be launched on each blur iteration.

Look at the command line options of `demo_repeated_blur`, and study its main function. The program populates a vector of a user-specified size with random doubles in the range [0, 100]. It then executes a repeated blur function for a given number of iterations. It is likely that the average value of the numbers in the vector is very close to 50, and after sufficiently many blurring iterations it is highly probable that all numbers in the vector will converge to values reasonably close to 50; the program reports success if the values are in the range (48, 52). If the program succeeds, the runtime for the blurring computation is reported.

Use this program to check that your fork-join blur is working correctly, and benchmark it against the sequential blur algorithm.

You will likely find that, unless you use very large arrays, you do not see much performance improvement when using your fork-join version with multiple threads, and a single-threaded version of your fork-join program will likely exhibit a slow-down.

Why do you think this is the case?

7. Your fork-join implementation in Question 6 forked and joined n threads on every blur iteration.

   An alternative to this is to use an *execution barrier* (not to be confused with a memory barrier) to fully synchronise all threads without actually joining the threads. A barrier can be used to ensure that all threads reach a particular point in a computation before any proceed past that point.

   In the `SenseReversingBarrier` class, found in `src/repeated_blur/sense_reversing_barrier.h`, implement a C++ version of the *sense reversing barrier* described in Section 18.3 of *The Art of Multiprocessor Programming*, 2nd edition, or Section 17.3 of the 1st edition of the

book. This barrier is also described in Section 5.2.1 of the *Shared Memory Synchronization* book.

As with the ALock (Question 5), the descriptions of this barrier algorithm found in these book requires using thread-local storage, so again you will need to research how to use thread-local storage in C++. It is fine (for this exercise) to make the assumption that only one barrier object will ever exist, so the limitation that C++ thread local storage variables and fields must be static should not be a problem. Remember that a static class member needs to be re-declared outside the class; feel free to put the out-of-class declaration in `demo_repeated_blur.cc`.

In `RepeatedBlurBarrier`, implement a version of repeated blur that:

- Sets up a temporary buffer to be used for ping-ponging, and declares a `SenseReversingBarrier`
- Launches `num_threads` threads, each with access to the data buffers and the barrier
- Has each thread perform `num_blur_iterations` iterations, where on each iteration a thread takes care of blurring a chunk of one buffer into the other
- Uses the barrier to synchronise the threads between loop iterations
- Only joins the threads after all blurring has been completed

Benchmark this against the sequential and fork-join implementations via the `barrier` argument to `demo_repeated_blur`. You should find that it performs a lot better than the fork-join implementation.

Why do you think this is the case?

8. Study the sense reversing barrier algorithm closely. Do you think that it is truly necessary to use thread-local storage? Is it possible to implement this algorithm correctly *without* thread-local storage?

   Use the skeleton class `SenseReversingBarrierNoTls` (where `Tls` stands for thread-local storage), and the `RepeatedBlurBarrierNoTls` method in `repeated_blur_demo.cc` to experiment with the correctness and performance of your ideas here.

9. Consider the smart futex-based mutex implementation shown in the lectures.

   Suppose the line:

   ```
   old_value = cmpxchg(kFree, kLockedWaiters);
   ```

   at the end of the `do...while` loop were changed to:

   ```
   old_value = cmpxchg(kFree, kLockedNoWaiters);
   ```

   What impact would this have on the correctness of the mutex implementation? If you think it would still be correct, argue why. If you think it would be incorrect, describe a scenario involving threads using the mutex that would lead to problems.

10. Consider the smart futex-based mutex again. Supposed you are tasked with rewriting the `Unlock` method so that none of the following read-modify-write (RMW) operations are used (neither directly, nor via a helper function such as `cmpxchg`):

    - `exchange`
    - `compare_exchange_strong`

- `compare_exchange_weak`
- `fetch_and`

Can you find a way to implement a working version of `Unlock` that does not use these RMW operations?

Hint: You may wish to consider exploiting the fact that the three value the lock state can take are consecutive.