Theory & Practice of Concurrent Programming (COMP60007)
Tutorial 1: C++ Concurrency

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

Note: As indicated in bold below, the final questions of this sheet rely on theoretical material that Azalea will teach and that Ally will follow up on in due course, so no need to try those questions until later in the term.

1. This question involves implementing a *recursive* mutex (similar to `std::recursive_mutex`, but implemented from scratch), and then implementing a container class whose methods are protected by a recursive mutex.

   The file `recursive_mutex/recursive_mutex.h` provides the skeleton of a `RecursiveMutex` class. A recursive mutex instance should keep track of which thread (if any) has locked the mutex, and the number of times they have locked the mutex.

   The `Lock` and `Unlock` methods should behave as follows:

   - If `Lock` is called by a thread and no thread holds the recursive mutex, the caller should succeed in locking the recursive mutex, and the lock count should be set to 1.
   - If `Lock` is called by a thread that already holds the recursive mutex, the lock count should be incremented.
   - If `Lock` is called by a thread but some other thread holds the recursive mutex, the thread should wait until the recursive mutex becomes free.
   - When a thread that holds the recursive mutex calls `Unlock`, the lock count should be decremented. If it reaches zero, the field tracking which thread holds the recursive mutex should be cleared, and any threads waiting to lock the recursive mutex should be notified.

   Where possible, use assertions to check (a) that the recursive mutex is used correctly (e.g. only a thread that holds the recursive mutex should unlock it), and (b) that the internal state of the recursive mutex is consistent.

   You can use a regular mutex to protect the state of the recursive mutex during calls to `Lock` and `Unlock`, and a condition variable to support waiting and notification. The regular mutex should only be held during these method calls, and should be released before each call returns.

   A recursive mutex implementation is presented in Section 8.4 of *The Art of Multiprocessor Programming*. Try to avoid looking at this implementation until you have thought hard about how to solve the problem independently. If you cannot solve it independently, try reading the text of Section 8.4 without looking at the associated Java code example and see whether that provides sufficient detail for you to work out how to implement a solution. Eventually, do inspect the Java code in the book to see how closely it mirrors your C++ implementation.

   The `recursive_mutex/container.h` file contains the skeleton of a templated `Container` class. Implement this class as follows:

- A `std::vector<T>` field should represent the contents of the container
- A `RecursiveMutex` field should be available to protect the methods of the container
- The `Add` method should push its argument on to the back of the vector, while holding the recursive mutex
- The `AddAll` method should acquire the recursive mutex for its duration, and should repeatedly call `Add` to add elements of the given vector to the container (this will lead to the mutex being locked multiple times by the thread that executes `AddAll`)
- The `Show` method should print the contents of the container to standard output, in the form `[a, b, c, ...]`.

In `recursive_mutex/demo_recursive_mutexes.cc`, write some code demonstrating that your container works properly when manipulated by multiple threads.

What is the difference between the implementation of `AddAll` proposed above, and an implementation that would repeatedly call `Add` *without* first acquiring the recursive mutex?

2. Recall the FIFO queue example shown in the lectures. The queue used two condition variables: one to allow threads to wait for the queue to become non-empty, and one to allow threads to wait for the queue to become non-full.

   Consider an alternative design where just *one* condition variable is used, so that all threads must use this condition variable to be notified about any updates to the queue.

   Does changing the design to use this single condition variable, but making no other changes, lead to a correct FIFO queue? If so, why is this the case?

   If not, explain why this approach would not be correct, and what could be done to rectify the problem.

3. Consider again the FIFO queue studied in the lectures, with its two conditions, `not_full_` and `not_empty_`. Recall that whenever a thread would successfully enqueue an element to the queue, the thread would call `notify_one` on the "non-empty" condition variable.

   It might seem to be wasteful to notify on this condition variable if the queue was *already* non-empty before the thread enqueued its element. Instead, it might appear to be both correct and more efficient to *only* notify on this condition variable when the size of the queue has changed from 0 to 1—i.e. when the queue has changed from empty to non-empty.

   Consider this proposal, both by thinking about it, and implementing it and experimenting with it. Is the optimisation correct? If not, what changes would you need to make to the design in order for it to be correct?

   Hint: This question is based on Section 8.2 of "The Art of Multiprocessor Programming", in particular on the example shown in Figure 8.6 of the book.

4. A *readers-writers* lock is a lock that can be held either by a single *writer*, or by one or more *readers*. Readers-writers locks are useful for protecting concurrent objects that have read-only methods, especially when those methods are likely to be invoked significantly more frequently than methods that modify the object. This is because readers-writers locks allow multiple threads to execute read-only methods concurrently, reducing contention. A readers-writers lock is sometimes referred to as a *shared mutex*. C++ provides a readers-writers lock via the `std::shared_mutex` class. We will look at implementing shared mutexes from scratch in terms of regular mutexes.

The tutorial data files contain a `SharedMutexBase` class, in `shared_mutexes/shared_mutex_base.h`, and skeletons for a number of other classes.

Populate the following classes:

- `SharedMutexStupid` (in `shared_mutexes/shared_mutex_stupid.h`): This class should implement the methods of `SharedMutexBase` using a *regular* mutex, so that it will not actually allow for multiple readers.
- `SharedMutexSimple` (in `shared_mutexes/shared_mutex_simple.h`): This class should have members to represent whether a writer holds the mutex, and the number of readers that hold the mutex. It should maintain the invariant that either there is no writer or zero readers. This should be achieved by having each method use a regular mutex and associated condition variable to block until the required locking condition becomes true (in the case of the lock methods), or to notify waiting threads that the state of the shared mutex has changed (in the case of the unlock methods).

  You can adapt the algorithm shown in Section 8.3.1 of *The Art of Multiprocessor Programming* to C++, but try to work on an independent implementation first.

  Note that there is an error in Figure 8.9 of the first edition of *The Art of Multiprocessor Programming*. The `while` loop at line 48 should have the condition:

  `readers > 0 || writer`.
- `SharedMutexFair` (in `shared_mutexes/shared_mutex_fair.h`): A problem with the `SharedMutexSimple` implementation is that it can lead to writer starvation. Think about why this is the case.

  Study the "Fair readers-writers lock" in section 8.3.2 of *The Art of Multiprocessor Programming* and adapt the given algorithm to C++ (making sure that you understand it).

  Unfortunately there are some errors in the figures that describe this algorithm in the first edition of the book, so take account of the relevant errata from this link before studying it if you do not have the second edition: `https://studylib.net/doc/8915596/errata`
- `SharedMutexNative` (in `shared_mutexes/shared_mutex_native.h`): This class should delegate calls to corresponding methods of `std::shared_mutex`. It allows you to benchmark your shared mutex implementations against the C++ standard library implementation using a common interface.

The `shared_mutexes/demo_shared_mutexes.cc` file contains a `main` function with some example code for timing a computation. In this file, write a benchmark that compares both performance and fairness of these shared mutex implementations on synthetic workloads. For each mutex implementation in turn, your benchmark should launch N reader threads and a single writer thread. Each thread should have access to a shared (non-atomic) integer and a readers-writers mutex.

The *writer* should perform the following `max_value` times (where `max_value` is an integer limit that should be passed to the writer):

- Acquire the writer lock
- Increment the shared variable
- Release the writer lock

A *reader* thread should repeat the following actions until the shared variable reaches `max_value`:

- Acquire a reader lock
- Do some "work" by spinning or sleeping for a while (so that the reader lock is held for some length of time)
- Release the reader lock

Furthermore, the readers should all share an atomic counter which a reader should increment whenever it attempts to acquire a reader lock. A reader should exit early if the value of this shared counter exceeds `max_read_attempts`, a constant that readers can take as a parameter.

Have your benchmark print:

- The total time for execution
- The final value of the shared integer
- The number of read attempts

Experiment with different numbers of reader threads, and different values for `max_value` and `max_read_attempts`. Also experiment with performance and fairness when the shared variable is incremented by multiple writers.

What do the values you observe tell you about the performance and fairness of your `SharedMutexBase` implementations?

5. The fair readers-writers lock from *The Art of Multiprocessor Programming* uses a pair of integer fields to track the number of read acquires and the number of read releases. Do you think it is actually necessary to have both of these fields? If not, (a) instrument your `SharedMutexFair` class with an alternative field and assertions to show that your alternative field does just as good a job as the pair of integer fields, and (b) write a simpler version of the shared mutex that uses this simpler implementation.

6. Read about the `std::call_once` function from `<mutex>`. Can you think of potential use cases for this function? The function depends on a special struct, `std::once_flag`. How might `std::once_flag` be implemented?

You can stop here until Ally's second block of teaching commences. The remaining questions are related to *relaxed memory*, a concept that Azalea will first cover in the theory part of the course, and that Ally will follow up on, in the context of C++, later in the term. Feel free to try have a look at these questions now, but don't worry if you haven't yet been taught about relaxed memory and memory orderings.

7. This question involves writing a synthetic example to illustrate the *relaxed* memory ordering.

   The file `random_sets/demo_random_sets.cc` contains a placeholder `main` method showing how to use the C++ Mersenne Twister 19937 engine to generate random integers. The main method also shows you how to use a high resolution clock for benchmarking. It also declares a constant, `kMaxValue`, set to 2^{24}.

   Write a function, `RandomSetSC`, with the following signature:

```
static void RandomSetSC(
    std::array<std::atomic<bool>, kMaxValue>& random_set,
    size_t iterations);
```

This function uses the `std::array` class, from the `<array>` header file, which supports defining fixed-size arrays. Here we are using an array of size 2^{24}.

The function should use its own local Mersenne Twister 19937 engine to generate `iterations` random numbers in the range $0–2^{24} - 1$. For each generated number n, *true* should be stored to `random_set` at index n (regardless of the current value at that position of the array). The store should use the *sequentially consistent* (default) memory ordering.

Assuming that the initial contents of `random_set` is uniformly *false*, this has the effect of generating a random set with up to `iterations` elements in the range $0–2^{24}$, where for a given value i in this range, the value of the boolean `random_set`$[i]$ indicates whether i is present in the set.

In `main`, write code that creates an array object via `std::make_unique`, initialises the contents of the array to *false*, then launches 8 threads each of which executes `RandomSetSC` with an iteration count of 2^{24}. Use a high resolution clock to benchmark how long the execution of these threads takes (i.e., do not include the initialisation code in your benchmarking). After the threads have finished executing, report the size of the generated set by counting the number of elements in the array that are *true* (this can be done sequentially).

Now write a function `RandomSetRelaxed` that is identical to `RandomSetSC` but uses the *relaxed* memory ordering. Call this function in a multi-threaded fashion from your `main` method (writing similar code to report on the size of the set) and compare the performance obtained from the `SC` and `Relaxed` versions.

You should find that the `Relaxed` version is significantly faster. Why is this the case?

8. This question is also about a use case for the *relaxed* memory ordering. You will write a series of functions for producing histograms of data. This is a common use case in a variety of application domains. However, to keep the example simple we will consider histograms that are generated at random. (A possible use case of this could be to assess whether a pseudo-random number generation yields a uniform distribution.)

   Similar to Question 7, the file `histograms/demo_histograms.cc` contains a placeholder `main` method showing how to use the C++ Mersenne Twister 19937 engine to generate random integers in the range 0–9. The main method also shows you how to use a high resolution clock for benchmarking.

   Write a function, `RandomHistogramSC`, with the following signature:

   ```
   static void RandomHistogramSC(
       std::array<std::atomic<int>, 10> &histogram,
       size_t iterations);
   ```

   Again, this function uses the `std::array` class from the `<array>` header file. Here we are using an array of size 10.

   The function should use its own local Mersenne Twister 19937 engine to generate `iterations` random numbers in the range 0–9, atomically incrementing `histogram` at the index associated with each generated number. The atomic increment should use the *sequentially consistent* (default) memory ordering.

In `main`, write code that creates an empty histogram, then launches 8 threads each of which executes `RandomHistogramSC` with an iteration count of 2^{24}. Use a high resolution clock to benchmark how long this takes.

Now write a function `RandomHistogramRelaxed` that is identical to `RandomHistogramSC` but uses the *relaxed* memory ordering. Call this function in a multi-threaded fashion from your `main` method and compare the performance obtained from the `SC` and `Relaxed` versions.

Do you observe a noticeable difference in the performance of the `Relaxed` version? If so, argue why this is the case. Either way, inspect the assembly code generated by the compiler to see what is going on. You can get an assembly dump in `demo_histograms.cc` by running this command:

```
clang++ -S src/histograms/demo_histograms.cc -O3 -std=c++17
```

Can you think of a way of optimising each of the functions you have written to dramatically reduce the number of read-modify-write operations that they perform, without changing what they compute?

Supposing that the *Relaxed* version of your original function did significantly outperform the `SC` version, do you think this would still be the case with your optimisation? If not, then whose "law" is this an example of?

Can you think of a scenario involving histogram computation where your optimised version would not be appropriate?