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?

Solution. A model implementation is provided in the archive of solution code. Compare this with your solution.

The difference between an implementation of `AddAll` that first acquires the recursive mutex and one that does not is that the implementation that acquires the recursive mutex will behave *atomically*: all elements will be added to the collection in an indivisible step. If `AddAll` does not acquire the recursive mutex and simply calls `Add` repeatedly then (a) an observer of the collection may see the elements provided to an `AddAll` operation appear in the collection gradually, and (b) the calls to `Add` made by concurrent calls to `AddAll` may interleave.

2. 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.

- `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).
- `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?

Solution. A model implementation is provided in the archive of solution code. Compare this with your solution. The `SharedMutexFair` class in the model solution includes a `ghost_num_readers_` field that relates to Question 3 and can otherwise be ignored.

When I run the demonstration code in `demo_shared_mutexes.cc` five times on my 8-core Linux laptop (in release mode), I observe the following execution times, in milliseconds:

Mutex kind	Median	Mean	Min	Max
Stupid	10813	8401	1745	12118
Simple	2029	2022	1976	2053
Fair	4	3	1	4
Native	1665	1665	1645	1686

I observe the following read counts:

Mutex kind	Median	Mean	Min	Max
Stupid	674555	532556	108486	800000
Simple	800000	800000	800000	800000
Fair	198	179	84	211
Native	799969	799901	799728	800000

The demo code features a single writer and 8 readers. The maximum number of read attempts is set to $100,000 \times N$ where N is the number of readers, so to $800,000$.

The data shows that the fair shared mutex performs extremely well from the writer's point of view: the writer succeeds in incrementing the shared counter in a matter of milliseconds, with in the order of hundreds of read attempts taking place concurrently.

In contrast, the simple shared mutex is extremely unfair to writers: a writer only completes at all because readers give up when the maximum number of read attempts is reached.

Performance of the stupid shared mutex (that does not actually use readers-writers locks) is both poor and rather variable, but is at least reasonably fair: the limit of $800,000$ read attempts is rarely reached.

I was surprised that using a native `std::shared_mutex` led to poor writer throughput, with readers getting close to or reaching the maximum number of read attempts.

For a good blog post on more advanced readers-writers lock implementations, see:

`https://eli.thegreenplace.net/2019/implementing-reader-writer-locks/`

3. 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.

Solution. The book does not justify why the pair of fields are necessary, and indeed they do not appear to be necessary. In my sample solution for `SharedMutexFair` I have included a `ghost_num_readers_` field. I have also introduced an `Invariant()` method that asserts the invariant:

```
ghost_num_readers_ == read_acquires_ - read_releases_
```

Whenever `read_releases_` is incremented, `ghost_num_readers_` is decremented, and whenever `read_acquires_` is incremented, `ghost_num_readers_` is incremented. Assertions are used throughout the code to confirm that the invariant holds, and on careful manual inspection of the code it is clear that this invariant is indeed guaranteed to hold.

Thus the pair of fields, `read_acquires_` and `read_releases_`, could be replaced with a single `num_readers_` field.

I find it frustrating that the book does not justify the use of a pair of fields. I expect the motivation may be performance-related: if two fields are used then there may be less cache contention because `LockShared()` only needs to access `read_acquires_` and not both fields. However, this would require the fields to be stored in different cache lines, and even then I expect the performance benefit would be small.

If a member of the class finds out more about this I would be really interested to hear.

4. 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?

Solution. A potential use case for `std::call_one` is when threads *may* require access to a library that requires a global setup routine to be invoked once on behalf of all threads in the process.

If the threads executing an application are guaranteed to require access to the library then the setup routine should simply be called before the threads are launched, rather than via `std::call_once`.

However, if the threads might not need access to the library at all it would be wasteful to needlessly initialise the library. In this case the library can be lazily initialised by the first thread that needs it by having that thread invoke `std::call_once`, passing a function that will invoke the necessary setup routine.

An example of global setup routines in a practical library is the `curl_global_init` routine from libcurl (`https://curl.se/libcurl/c/curl_global_init.html`).

The `std::once_flag` struct could be implemented via a boolean field, initialised to *false*, and a mutex to protect the field. A thread executing `std::call_once` would then acquire the mutex and test the boolean field. If the field is *false* the thread would execute the target function, set the boolean to *true* and release the mutex. Otherwise the thread would simply release the mutex.

It might be more efficient to make the boolean flag an *atomic* boolean. A thread calling `std::call_once` could then test the value of this boolean and only acquire the mutex in the event that the boolean is *false*. (It would still be necessary for the thread to acquire the mutex and re-test the boolean field in this case, since another thread may be concurrently executing `std::call_once`.)

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.

5. 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?

Solution. A model implementation is provided in the archive of solution code. Compare this with your solution.

On my 8-core Linux laptop, the `Relaxed` version runs 2-3 times faster than the `SC` version. This is because the use of relaxed ordering allows store buffering effects: by removing the requirement for distinct threads to observe updates to distinct array elements in the same order, the compiler is able to emit plain stores to memory, rather than stores that enforce a memory ordering. In this use case, this relaxation has no effect on the sets that are generated. To confirm this, try changing both statements of the form:

```
std::mt19937 generator(device());
```

to use a fixed integer seed (e.g. 0) instead of `device()`. You should find that after such a change, the `Relaxed` and `SC` versions consistently produce identically-sized sets.

6. 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 5, 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?

Solution. A model implementation is provided in the archive of solution code. Compare this with your solution.

On an x86 architecture you won't find any performance difference between the `Relaxed` and `SC` versions. This is because on x86 there is no way to perform a `fetch_add` operation without issuing a full memory barrier: regardless of the memory ordering used in the source code, the `fetch_add` operation compiles to a locked `add` in the generated assembly, and locked operations implicitly issue a memory barrier.

If you have access to an ARM architecture (e.g. on a Mac with an M1 chip set) then you might see a performance improvement. Using the Compiler Explorer (`https://godbolt.org/`), you can check that, when targeting the ARM V8 architecture, Clang compiles a `fetch_add` instruction to a loop of the form:

```
.label:
        ldxr    w1, [x]
        add     w1, w1, w2
        stxr    w3, w1, [x]
        cbnz    w3, .label
```

when relaxed ordering is used, but a loop of the form:

```
.label:
        ldaxr    w1, [x]
        add     w1, w1, w2
        stlxr    w3, w1, [x]
        cbnz    w3, .label
```

when sequentially consistent ordering is used.

The differences are that `ldaxr` is a *load acquire* instruction, while `ldxr` is a plain load instruction, and `stlxr` is a *store release* instruction, while `stxr` is a plain store instruction. The acquire load and release store provide stronger memory ordering guarantees than the plain variants, but are more expensive to execute.

For interest, see here for details of data transfer instructions on 64-bit ARM platforms:

`https://developer.arm.com/documentation/dui0802/b/A64-Data-Transfer-Instructions/A64-data-transfer-instructions-in-alphabetical-order`

Leaving choice of architecture aside, the histogram function can be optimised very effectively by having each thread compute its own local histogram, which it then merges into the shared histogram after it has computed all of its local random values. This is illustrated by the `RandomHistogramLocal` function in the sample solution. You should find that this function runs considerably faster than (either of the) non-local variants.

Even on a platform (such as ARM) where using relaxed ordering originally provided a speedup, the speedup is unlikely to be visible when using this local histogram optimisation. This is because the vast majority of memory accesses will be local and non-atomic, and only 10 `fetch_add` instructions per thread will be necessary, for merging of local histograms. Even though these instructions will be faster when relaxed ordering is used, the percentage of the computation that they represent is very small, so the overall speedup will be negligible.

This is an example of Amdahl's law.