Theory & Practice of Concurrent Programming (COMP60007) Coursework 1

This coursework should be undertaken in groups of up to four. A single submission should be made per group. Each member of a group will thus be awarded the same mark for the associated submission.

The practical part has an associated data file that you should download from Scientia, containing skeleton code and a PDF excerpt from *The Art of Multiprocessor Programming*.

Your task is to write a series of implementations of a hash set class using C++. The implementations of this class will feature increasingly fine-grained concurrency. They are based on Java implementations described in *The Art of Multiprocessor Programming*. The purpose of this is to give you hands-on experience writing concurrent C++ code, working with the various synchronisation and atomic primitives that C++ offers, and using the ThreadSanitizer tool to debug data races in your code.

You will be graded based on:

- the correctness of the code you submit, which will be read carefully and subjected to intensive testing;
- whether you have used the concurrency features of C++ in an appropriately efficient manner in each part of the question;
- the quality of explanatory comments in your code base, explaining *why* you have used the concurrency features of C++ in the manner that you have, and any decisions you have made that deviate from the guidance that you will follow in *The Art of Multiprocessor Programming*.

What to submit: You should submit an archive, `practice.zip`, which should contain the contents of your `src` directory. When your `practice.zip` file is unzipped, this should lead to the `.cc` and `.h` files from your solution being unpacked straight into the current directory, with no enclosing folders. To create this archive, navigate into your `src` directory and run the command:

```
zip -r practice.zip *
```

1 Your high level task

Your task is to write four implementations of a hash set class. A percentage breakdown of marks associated with these implementations is as follows:

- A *sequential* hash set, i.e. one that is not thread safe (**10%**).
- A *coarse-grained* hash set, which achieves thread safety via a "big global lock" that protects the whole hash set (**20%**).
- A *striped* hash set, which achieves thread safety at a finer level of granularity via a separate lock for each *initial* bucket of the hash set, but which keeps the set of locks fixed for the lifetime of the hash set (**45%**).

- A *refinable* hash set, which is like a striped hash set except that there is always a separate lock per bucket of the hash set. That is, when the hash set is resized to use a larger number of buckets, the set of locks is resized correspondingly (**25%**).

Each team member should spend an estimated 10 hours in order to do well on the coursework, but this will vary according to programming ability. The refinable hash set is the most challenging part of the coursework, and getting it completely working could end up taking a lot of time. For this reason it is only worth 25% of the marks – do have a look at it, but if you are finding that the coursework is taking too long you might decide to skip it, or to submit a valiant but incomplete attempt at it, without losing too many marks.

Algorithms for very similar hash set implementations in Java are described in Chapter 13 of *The Art of Multiprocessor Programming*:

- Pages 305–315 of the second edition
- Pages 299–308 of the first edition

Remember that the first edition of the book is available online here:

`http://cs.ipm.ac.ir/asoc2016/Resources/Theartofmulticore.pdf`

The relevant pages of the first edition are included in the data file for the coursework as a PDF.

You should read the details of these implementations carefully, and have your C++ implementations follow a similar approach, with two exceptions:

- Your hash set classes should provide a `Size` method that yields the number of elements in a hash set; this is not one of the public methods described in *The Art of Multiprocessor Programming* (but it is easy to add).
- The methods of your hash set classes should *not* support re-entrancy. For example, it should not be possible for a call to `Add` to lead to another public method of the hash set being called.[1] Furthermore, you should exploit the fact that re-entrancy is not permitted to make your implementations simpler than those described in the book.

2 Where to write your code

You are strongly encouraged to create a Git repository to manage your code, starting from the initially-provided files.

You are provided with the following files:

- `src/hash_set_base.h`: A `HashSetBase<T>` "interface" (C++ class featuring only abstract methods) for a hash set, generic with respect to a template parameter `T`. This interface specifies the operations that your hash sets should support. You should *not* modify this file.

[1] Skip this footnote on a first reading, but once you have oriented yourself with the coursework, note that the code shown in the PDF extract from the book uses an `initializeFrom()` method that is called from `resize()`. The code for this method is not shown, but the accompanying text says: "Because the `initializeFrom()` method calls `add()`, it may trigger nested calls to `resize()`." Because the hash sets that you will built will not support re-entrancy, you must adapt your approach to avoid this.

- `src/hash_set_{...}.h`: Stub C++ classes for the hash sets that you will implement. Each is a subclass of `HashSetBase<T>`. The new code that you write should be contained in these header files.

- `src/benchmark.h/cc`: Utility functions for benchmarking a particular hash set implementation on a synthetic workload. Do *not* modify these files. Use `src/playground.cc` (see below) to experiment with your own workloads.

- `src/demo_{...}.cc`: Stand-alone programs that use the benchmarking utility functions to run and provide performance numbers on each hash set implementation. An exception to this is that `src/demo_sequential.cc` merely checks that a number of method calls on a sequential hash set implementation work as expected. Do *not* modify these files.

- `src/checks/`: a variety of C++ files that check that each of your hash set implementations can be built in a standalone fashion, and that they can be build when all included simultaneously. Do *not* modify these files. **Note:** do make sure that, when you build your solution as described below, these files all compile successfully. If they do not, it probably means that one of your hash set implementations depends on a header file that you have not explicitly included in the `.h` file associated with the hash set.

- `src/playground.cc`: An initially empty program in which you can write code to experiment with your hash set implementations. The build system for the coursework will build this target using the build configurations detailed below.

3 Building and running your solution

Build files and scripts are provided to support building your solution under Linux using the Clang compiler, making use of the ThreadSanitizer and AddressSanitizer tools.

The project is configured to build with very pedantic compiler warnings, and with warnings treated as errors. This is good practice when writing C++ code: compiler warnings often indicate a lack of portability of the potential for undefined or erroneous behaviour.

The scripts assume that the `cmake`, `make`, `clang++-18` and `clang-format-18` tools are on your path. This is the case on a standard lab machine. If you prefer to develop on a non-Linux system then feel free to edit your local copies of the build files and scripts appropriately.

3.1 Building from scratch

To build your solution from scratch (deleting the results of any previous build), run:

`./scripts/check_clean_build.sh`

This will cause the code to be built in four different configurations:

- `./temp/build-release`: a release build, with full optimisation and no debug information. This is the version to use to test how well your code is performing.

- `./temp/build-debug`: a debug build, with no sanitisers. Ideal if you wish to debug your code using a tool such as GDB without sanitiser support.

- `./temp/build-asan`: a debug build, with AddressSanitizer enabled. This is useful for finding various C++-related gotchas such as buffer overflows.

- `./temp/build-tsan`: a debug build, with ThreadSanitizer enabled. This is invaluable for finding data races in your code.

You can use `scripts/check_all.sh` to perform a clean build *and* check that your source code is correctly formatted (as discussed below).

3.2 Building incrementally

The `scripts/check_build.sh` script will build your solution incrementally, based on any recent changes you have made. This is useful for rapid development.

3.3 Building using an IDE

If you wish to use an IDE that supports CMake (the CLion IDE from JetBrains is an excellent choice) then you should be able to create a new IDE project based on the *CMakeLists.txt* file, and then configure CMake in your IDE by looking at the way CMake is invoked in the `scripts/build.sh` file.

3.4 Running the built programs

In a `./temp/build-*` directory you will find a `playground` executable, corresponding to the main function in `playground.cc`, and a `demo_*` executable corresponding to each `demo_*.cc` file.

As mentioned above, `playground` is where you can try out your hash set implementations in any manner you wish.

The `demo` programs all take the following arguments (except `demo_sequential`, which does not take an argument for the number of threads):

- The number of threads that should execute the benchmark
- The initial capacity of the hash set
- A chunk size that controls how much work each thread will do

Look at `benchmark.h/cc` to see how these arguments are used.

The `scripts/run_benchmark.sh` script runs the `demo` programs in turn on some example workloads.

4 Code style

Your code should be written using the concurrency features available in C++20 – this is the version of C++ covered in the lectures.

You should make reasonable efforts to ensure that your code adheres to the Google C++ style, e.g. with respect to naming conventions:

`https://google.github.io/styleguide/cppguide.html`

Style will largely be ignored during marking, but submissions with particularly bad coding style will be subject to a small penalty.

The project is configured to use `clang-format-18` to check whether your code is at least *formatted* according to the Google style. Use `scripts/check_format.sh` to check whether your code is correctly formatted, and `scripts/fix_format.sh` to automatically fix formatting problems.

Within a single hash set implementation you should make reasonable efforts to follow good coding style, including minimising duplicate code.

However, do *not* aim to share code between your hash set implementations – make them all self contained. The task of writing many variations on the same algorithm is an artificial one, thus it would be artificial to attempt to refactor your solution to share code between variations. Having each implementation be self-contained will also make your code easier to assess.

Note that you will be writing all your new code in header files. This is typical when writing templated C++ classes, because when a templated class or method is instantiated, the compiler needs to recompile the template for the particular types used in the instantiation. To do so, it needs access to all of the code. It will not have the required code access if the templated method or class is split out into a header file and an implementation file.

5 Guidance

Be sure to write detailed comments in your code describing the *intent* of the particular concurrency-related features of C++ that you use. Make sure your comments do not parrot your code (e.g.: "This acquires a lock." would not be a good comment!) Instead, focus on the reason why concurrency-related code is *necessary* (e.g. "Acquiring this lock ensures that ...")

If you have exploited non-re-entrancy to good effect, use code comments to explain how your solution differs to the approach proposed in the book, and justify why it is correct and whether it has the potential to be more efficient.

Here are a few tips to help guide you towards a good solution.

- A good representation for a table storing elements of type `T` is a vector of vectors of `T`s.
- The `<functional>` header file provides a class, `std::hash<T>`, which returns a `size_t` hash code for many types. You may assume that `std::hash<T>` is available for any type `T` with which your hash sets are instantiated. To obtain the hash code of an expression e, do: `std::hash<T>()(`e`)`. The two pairs of parentheses are needed because `std::hash<T>` is a *class* for which the parentheses operator has been defined. The first `()` invokes the class's zero-argument constructor, after which `(`e`)` applies the parentheses operator to yield a hash code for e.
- Manage mutexes using built-in RAII classes where possible. For example, in the coarse-grained hash set implementation, a `std::scoped_lock` should work well.
- When built-in RAII classes are not sufficient for resource management, implement your own simple RAII classes for resource management where possible.
- Certain concurrency-related C++ classes are not *copyable*; this means that they cannot be directly stored in a `std::vector`. In such cases, consider using a vector of *unique pointers* to objects of the desired types.
- Exploiting the fact that your hash set classes should not be re-entrant is most relevant when implementing the refinable hash set.
- Working out what to do about new vs. old locks when resizing a refinable hash set is the hardest part of this coursework. Part of this blog post on locking in WebKit offers insights that might help you.

  `https://webkit.org/blog/6161/locking-in-webkit/`

  It is a fascinating blog post, so I encourage you to read all of this, but you might want to skim it first to identify the most relevant part.

6 Optional extensions

- See whether you can improve the performance of your hash set implementations for workloads that make a lot of `Contains` calls compared with `Add` and `Remove` calls by using reader-writer locks.
- Implement a *lock-free* version of the hash set, as explained in Section 13.3 of *The Art of Multiprocessor Programming*.