CS4204 — Concurrency and Multi-Core Architectures

Practical 1 : Concurrent Data Structure Implementations

Christopher Brown & Susmit Sarkar

Deadline: 2021-03-03 (Wed Week 6) 21:00

Credits: 50% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification at least a week before the deadline. You must contact the lecturer(s) regarding any queries well in advance of the deadline.

1 Purpose

This practical will help practice and develop your skills in:

- programming low-level concurrent algorithms;
- understanding and analysing lock-free techniques;
- appreciating the performance and correctness issues for programming at this level.

2 Overview and Background

We would like to implement an abstract set data structure that can be safely accessed by multiple threads. This can be used as the basis of several algorithms.

The abstract type signature of a concurrent set looks like the following:

```
class CSet<T> {
  bool contains (T element);
  bool add (T element); /* return true if was not present before */
  bool remove (T element); /* return true if was present before */
}
```

If implementing in C, where generics are not syntactically present, feel free to use `void *` as the element type, or a particular type (e.g. `int`) as long as your algorithm is type-independent.

3 Tasks

Your task is to implement, using either C or C++, a few strategies for a concurrent set data structure. Correctness and performance are equally important. We recommend starting with a simple algorithm first. You would need to test the performance of your implementations, and ideally implement one or more high-performance implementations.

In C (using a relatively modern version of either gcc or clang), you will probably want to include the header file

```
#include <stdatomic.h>
```

In C++ the corresponding header is

```
#include <atomic>
```

For each implementation, you should argue for the correctness of your algorithm, in the concurrent shared-memory setting. This argument can be either theoretical, or experimental, or ideally both.

Analyse the performance of each implementation by running them on sequences of add, contains, and remove operations called concurrently from several threads. Exploring performance in different usage scenarios is encouraged.

Note that you may make reasonable simplifying assumptions if you wish, or implement any helper functions/locations, as long as you implement the set operations.

4 Hints

Chapter 9 of the textbook presents a few implementation strategies using a linked list as the underlying data structure, which you may wish to look at. Using linked lists is a good choice. However, we do not insist on using linked lists; arrays or hash tables (expandable when necessary) or binary search trees are all reasonable alternatives.

Regardless of underlying data structure, a single lock per set is the easiest implementation strategy, and we recommend implementing this first to get a good baseline (and get practice in proving correctness and benchmarking performance).

For high-performance implementations, you may wish to consider lazy strategies, or non-blocking algorithms. You will probably need a `CompareAndSwap` or similar primitive as discussed in lectures for these. Both C and C++ have such primitives: in C you can use the `atomic_compare_exchange_weak` function, and in C++ the `std::atomic_compare_exchange_weak` function.

5 Submission

You should hand in the sources of your implementations, together with any recipes needed to build them. In your report, briefly discuss the key design decisions, any problems or unexpected features you encountered, the arguments for correctness, and the performance results of your implementations.

Make a **zip archive** of all of the above, and submit via MMS by the deadline.

6 Marking

I am looking for:

- a reasonable design and understandable code, with major decisions explained in comments or report;
- a convincing argument for correctness of the implementations;
- a study of the performance characteristics of the implementations; and
- a report giving a critical analysis of the design and the tradeoffs of the approaches implemented.

The standard mark descriptors in the School Student Handbook will apply:
https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

In particular, a very basic implementation (such as an array-based implementation with a single lock) would fit the definition of a reasonable attempt achieving some of the required functionality, and could get marks up to 10. More complex algorithms (correctly implemented) will get higher marks.

In all cases, but particularly for 17 and above, I will be looking at the quality of the theoretical and experimental arguments and analysis. A simpler algorithm excellently implemented and analysed is better than a very complex algorithm with no explanation.

7 Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

8 Good Academic Practice

The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/