Document number: P0940R0
Date: 20180212 (Jacksonville)
Project: Programming Language C++, WG21, SG1, LEWG, LWG
Authors: Michael Wong, Olivier Giroux
Email: michael@codeplay.com, ogiroux@gmail.com
Reply to: michael@codeplay.com

Concurrency TS is growing: Concurrent Utilities and Data Structures

Introduction

This is a proposal for a draft new section to the C++ Standard to support SG1 Concurrency features. We foresee a number of upcoming features for inclusion. We also foresee some features in existing sections that are concurrency related that is worth moving into this new section.

There is no wording yet until we agree on the structure of the reorganization.

Organization Proposal

A large number of Concurrency features are coming for C++20. This is because Concurrency TS1 was not added to C++17. However some of the features in it has changed. There are also many new features aiming for Concurrency TS2. Lets us recap.

Concurrency TS1 was published in Jan 19, 2016[P0159] but still too late for C++17. It contains

- atomic_shared_ptr and atomic_weak_ptr class templates
- Latches and barriers
- Improvements to std::future<T> and Related APIs

Since its publication and through usage feedback, several of these facilities have been rethought. In a recent SG1 meeting in Toronto, Atomic_shared_ptr is now atomic<shared<ptr>>. Latches and barriers is undergoing a partial redesign to split the arrive/wait facilities. Even futures is being redesigned to serve the needs of executors, TLS, and other facilities better.

Concurrency TS2 is an ongoing WIP but should contain the following which has been making its way through WG21/SG1:

- Executors that links concurrency and parallelism constructs with different execution resources. There is a possibility that this may split off into its own TS.
- Constructs that deal with contention such as apply(), latch, basic_barrier<F>, typedef barrier, binary_semaphore, counting_semaphore, synchronic
- Data structures such as Concurrent queues, counters, Synchronized<T>, Atomic_ref<T>
- Several synchronization primitives for locked-free programming on concurrent data structures. These are cell, hazard ptr and RCU. These extends the existing shared_ptr and the proposed atomic_shared_ptr which all have safe reclamation facilities. As such we also propose moving shared_ptr and atomic<shared<ptr>> to this new location. We suspect this part may be controversial, so would ask for discussion on this topic.

Given the proliferation of these and other facilities, as Concurrency TS editor, and before we move sections and inject new wordings, we propose the following new chapter to handle these concurrency utilities for Concurrency TS2 and TS1.

At this point, there is no plan to change or update Concurrency TS1. However, not all may agree with that. We would also invite a discussion on this in the upcoming meeting.

Proposed Structure

Clause 34: Concurrency Utilities Library

- **34.1 Concepts**
  - 34.1.1 Future
  - 34.1.2 Executor
- **34.2 Execution**
  - 34.2.1 Executors
  - 34.2.2 require, prefer
  - 34.2.3 {concrete executors}
- **34.3 Contention**
  - 34.3.1 apply()
  - 34.3.2 latch
  - 34.3.3 basic_barrier<F>
  - 34.3.4 typedef barrier
  - 34.3.5 binary_semaphore
  - 34.3.6 counting_semaphore
  - 34.3.7 synchronic
- **34.4 Data structures**
  - 34.4.1 Concurrent queue
  - 34.4.2 Concurrent counters
  - 34.4.3 Synchronized<T>
  - 34.4.4 Atomic_ref<T>
- **34.5 Safe Reclamation**
  - 34.5.1 cell
  - 34.5.2 RCU
  - 34.5.3 Hazard Pointers
  - 34.5.4 shared_ptr
  - 34.5.5 atomic<shared<ptr>>

The reason I am interested in moving the latter two into the section on concurrency in some order with Safe Reclamation is that they are actually shared concurrency structures. Shared_ptr exists where it does not (Clause 20 Smart Pointer) because at the time, it was delivered with the Boost Smart pointer as a package. In this paper [P0233], the authors illustrate in the table in Section 7 a comparison of the capabilities between the various facilities for Reclamation. Reference Counting is the implementation behind shared_ptr and Split reference Counting (or Reference Counting with DCAS) is the implementation behind atomic_shared_ptr. These have many capabilities similar to Hazard Pointers, Cell and RCU differing only in the performance and lock-free implications.

We would ask SG1 to give guidance on this structure reorganization at the next meeting.

Acknowledgement

The author wishes to thank Maged Michael and Paul Mckenney for the comparison.

References

[P0159] Programming Languages — Technical Specification for C++ Extensions for Concurrency
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html
[P0233] Hazard Pointers: Safe Reclamation for Optimistic Concurrency
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0233r6.pdf