Synchronization

RCU & Hazard Pointer C++26

2024 ATS Spring Summit

Masaori Koshiba (masaori@apache.org)

Papers for C++ 26

`<rcu>`

- Read-Copy Update (RCU) - P2545R4

`<hazard_pointer>`

- Hazard Pointers for C++26 - P2530R3

History

Both of them were proposed for C++ Technical Specification "Concurrency TS 2". However, the target was changed to C++26.

Read-Copy-Update

> RCU achieves scalability improvements by allowing reads to occur concurrently with updates.

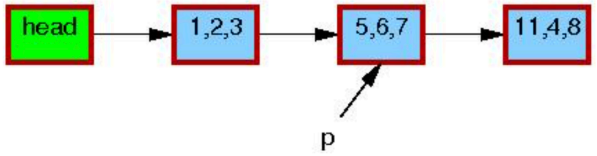
— What is RCU, Fundamentally?

Example 2: Maintaining Multiple Versions During Replacement

To start the replacement example, here are the last few lines of the example in the previous section:

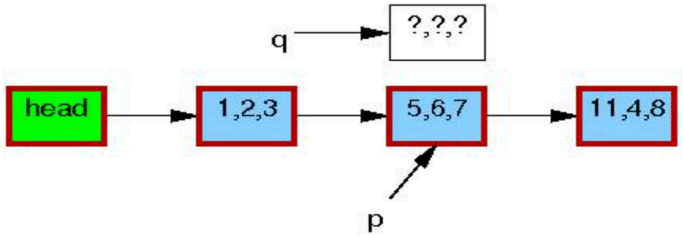
```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;
3 q->b = 2;
4 q->c = 3;
5 list_replace_rcu(&p->list, &q->list);
6 synchronize_rcu();
7 kfree(p);
```

The initial state of the list, including the pointer `p`, is the same as for the deletion example:

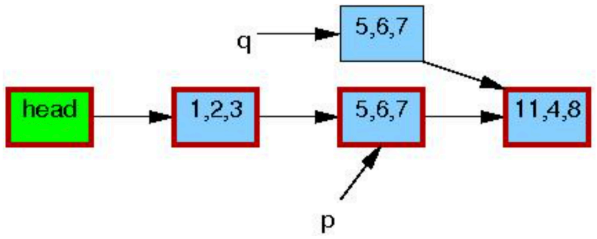


As before, the triples in each element represent the values of fields `a`, `b`, and `c`, respectively. The red borders on each element indicate that readers might be holding references to them, and because readers do not synchronize directly with updaters, readers might run concurrently with this entire replacement process. Please note that we again omit the backwards pointers and the link from the tail of the list to the head for clarity.

Line 1 `kmalloc()`s a replacement element, as follows:



Line 2 copies the old element to the new one:



Line 3 updates `q->b` to the value "2":

Implementations

C

- Linux Kernel (2002)
- liburcu - Userspace RCU
- OpenSSL (v3.3.x - d0e1a0ae)

C++

- folly/synchronization/Rcu.h
- paulmckrcu/RCUCPPbindings (C++ Binding of liburcu)

Example (Non-Intrusive)

```
#include <rcu>

struct Data : std::rcu_obj_base<Data> /* members */;
std::atomic<Data*> data_;

template <typename Func>
Result reader_op(Func fn) {
  std::scoped_lock l(std::rcu_default_domain());
  Data* p = data_;
  // fn should not block too long or call rcu_synchronize(),
  // rcu_barrier(), or rcu_retire(), directly or indirectly
  return fn(p); return
}

// May be called concurrently with reader_op
void update(Data* newdata) {
  Data* olddata = data_.exchange(newdata);
  std::rcu_retire(olddata);  // reclaim *olddata when safe
}
```

Hazard Pointer

> Each hazard pointer can be written only by its owner thread, but can be read by other threads.

— Safe memory reclamation for lock-free objects (2004)

Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects

Maged M. Michael

Abstract—Lock-free objects offer significant performance and reliability advantages over conventional lock-based objects. However, the lack of an efficient portable lock-free method for the reclamation of the memory occupied by dynamic nodes removed from such objects is a major obstacle to their wide use in practice. This paper presents *hazard pointers*, a memory management methodology that allows memory reclamation for arbitrary reuse. It is very efficient, as demonstrated by our experimental results. It is suitable for user-level applications—as well as system programs—without dependence on special kernel or scheduler support. It is wait-free. It requires only single-word reads and writes for memory access in its core operations. It allows reclaimed memory to be returned to the operating system. In addition, it offers a lock-free solution for the ABA problem using only practical single-word instructions. Our experimental results on a multiprocessor system show that the new methodology offers equal and, more often, significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support. We also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability, even in the presence of thread failures and arbitrary delays.

Index Terms—Lock-free, synchronization, concurrent programming, memory management, multiprogramming, dynamic data structures.

1 Introduction

A shared object is *lock-free* (also called nonblocking) if it guarantees that whenever a thread executes some finite number of steps toward an operation on the object, *some* thread (possibly a different one) must have made progress toward completing an operation on the object, during the execution of these steps. Thus, unlike conventional lock-based objects, lock-free objects are immune to deadlock when faced with thread failures, and offer robust performance, even when faced with arbitrary thread delays.

Many algorithms for lock-free dynamic objects have been developed, e.g., [11], [9], [23], [21], [4], [5], [16], [7], [25], [18]. However, a major concern regarding these objects is the reclamation of the memory occupied by removed nodes. In the case of a lock-based object, when a thread removes a node from the object, it is easy to guarantee that no other thread will subsequently access the memory of that node, before it is reused or reallocated. Consequently, it is usually safe for the removing thread to reclaim the memory occupied by the removed node (e.g., using `free`) for arbitrary future reuse by the same or other threads (e.g., using `malloc`).

This is not the case for a typical lock-free dynamic object, when running in programming environments without support for automatic garbage collection. In order to guarantee lock-free progress, each thread must have unrestricted opportunity to operate on the object, at any time. When a thread removes a node, it is possible that some other contending thread—in the course of its lock-free operation—has earlier read a reference to that node, and is about to access its contents. If the removing thread were to reclaim the removed node for arbitrary reuse, the contending thread might corrupt the object or some other object that happens to occupy the space of the freed node, return the wrong result, or suffer an access error by dereferencing an invalid pointer value. Furthermore, if reclaimed memory is returned to the operating system (e.g., using `munmap`), access to such memory locations can result in fatal access violation errors. Simply put, the memory reclamation problem is how to allow the memory of removed nodes to be freed (i.e., reused arbitrarily or returned to the OS), while guaranteeing that no thread accesses free memory, and how to do so in a lock-free manner.

Prior methods for allowing node reuse in dynamic lock-free objects fall into three main categories. 1) The IBM tag (update counter) method [11], which hinders memory reclamation for arbitrary reuse and requires double-width instructions that are not available on 64-bit processors. 2) Lock-free reference counting methods [29], [3], which are inefficient and use unavailable strong multiaddress atomic primitives in order to allow memory reclamation. 3) Methods that depend on aggregate reference counters or per-thread timestamps [13], [4], [5]. Without special scheduler support, these methods are blocking. That is, the failure or delay of even one thread can prevent an aggregate reference counter from reaching zero or a timestamp from advancing and, hence, preventing the reuse of unbounded memory.

This paper presents *hazard pointers*, a methodology for memory reclamation for lock-free dynamic objects. It is efficient; it takes constant expected amortized time per retired node (i.e., a removed node that is no longer needed by the removing thread). It offers an upper bound on the total number of retired nodes that are not yet eligible for reuse, regardless of thread failures or delays. That is, the failure or delay of any number of threads can prevent only a

• *The author is with the IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. E-mail: magedm@us.ibm.com.*

Manuscript received 15 Apr. 2003; revised 22 Oct. 2003; accepted 2 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0058-0403.

1045-9219/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

Implementations

C

- Concurrency Kit - ck/src/ck_hp.c

C++

- folly/synchronization/Hazptr.h

Example (Read-Mostly Shared Data)

```
#include <hazard_pointer>

struct Data : hazard_pointer_obj_base<Data> { /* members */ };
atomic<Data*> pdata_;

// Called frequently and in parallel
template <typename U, typename Func>
U reader_op(Func userFn) {
  hazard_pointer h = make_hazard_pointer();
  Data* p = h.protect(pdata_);
  // The user function userFn is allowed to block and call the writer function because hazard
  // pointer protection does not block writers or prevent the reclamation of unprotected objects.
  return userFn(p); // Safe to access *p.
} // RAII end of protection.

// May be called concurrently with readers
void writer(Data* newdata) {
  Data* old = pdata_.exchange(newdata);
  old->retire(); // Reclaim old when unprotected.
}
```

Cppcon 2023 The C++ Conference

October 01 - 06

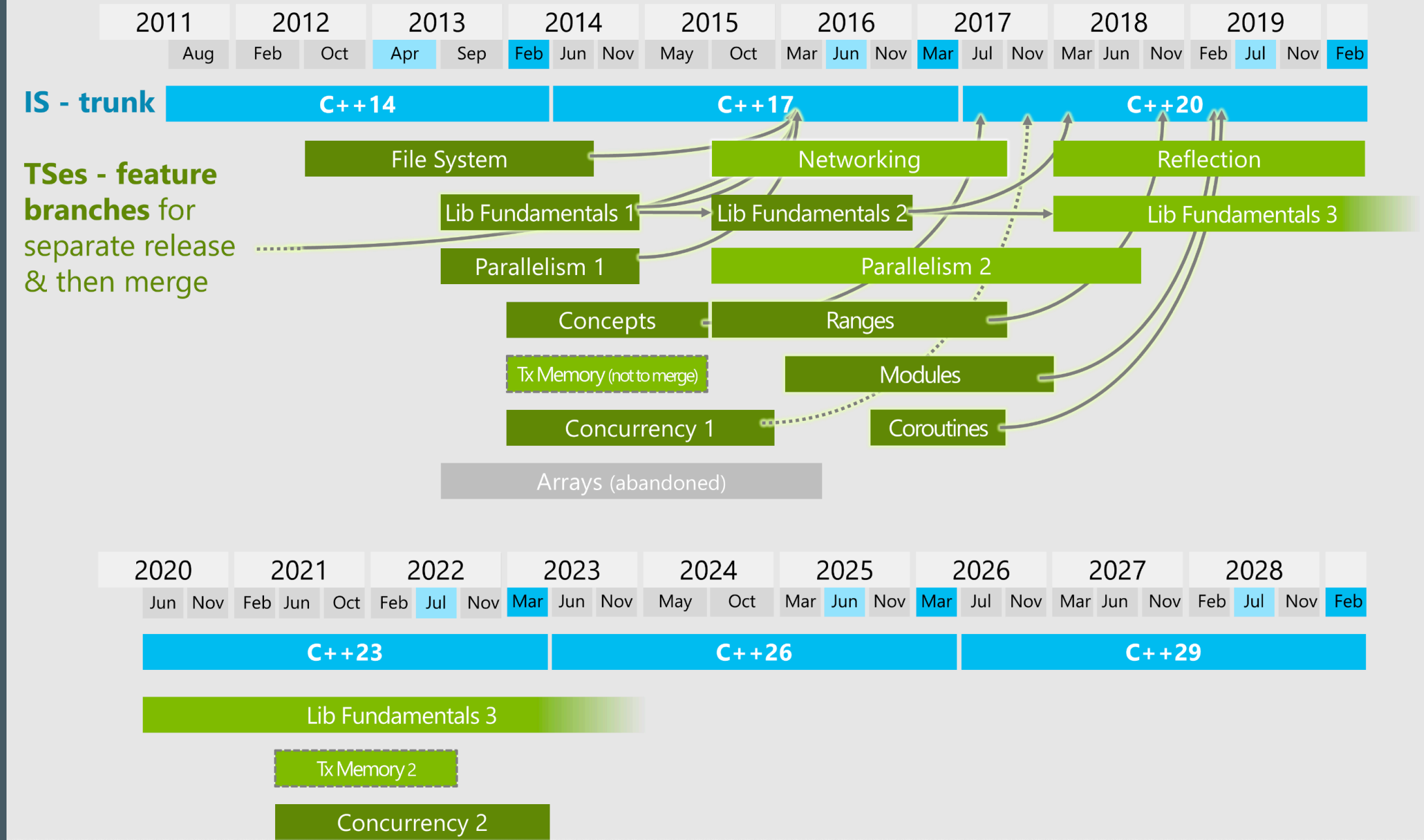
Maged Michael & Michael Wong

Concurrency TS2:
At Last Going to Geneva

Safe Reclamation Tradeoffs

- RW Locks and RCU protect many objects but are not suitable for long protection.
- Hazard pointers and reference counting are suitable for long protection but need to be specific about protected objects.
- Hazard pointers and RCU defer reclamation. RW-locks and reference counting reclaim immediately.

	RW Locks	Reference Counting	RCU	Hazard Pointers
Latency	High	High	Low	Low
Contention	Yes	Yes	No	No
Protection duration	Cannot be long	Can be long	Cannot be long	Can be long
Objects not yet reclaimed	Zero	Zero	Unbounded	Bounded
Irregular protection	General protection	Specific protection	General protection	Specific protection


2011 Aug
2012 Feb Oct
2013 Apr Sep
2014 Feb Jun Nov
2015 May Oct
2016 Mar Jun Nov
2017 Mar Jul Nov
2018 Mar Jun Nov
2019 Feb Jul Nov
2020 Jun Nov
2021 Feb Jun Oct
2022 Feb Jul Nov
2023 Mar Jun Nov
2024 May Oct
2025 Mar Jun Nov
2026 Mar Jul Nov
2027 Mar Jun Nov
2028 Feb Jul Nov Feb
IS - trunk
C++14
C++17
C++20
C++23
C++26
C++29
TSes - feature branches for separate release & then merge
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
Lib Fundamentals 3
Tx Memory 2
Concurrency 2

Ref.

Linux Kernel

- What is RCU?

Papers for C++26

- Read-Copy Update (RCU) - P2545R4
- Hazard Pointers for C++26 - P2530R3

Talk at CppCon 2023

- Concurrency TS2: Improved C++ Concurrency and Lock-free Programming

cppreference.com

- https://en.cppreference.com/w/cpp/header/rcu
- https://en.cppreference.com/w/cpp/header/hazard_pointer

ISO C++

- Current Status

C++ Concurrency in Action

Chapter 7. Designing lock-free concurrent data structures