Advances in Lock-Free Programming

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

What we have seen so far …

- Locks in OS
    - What are locks and why are they required?
    - Different types of Locks
    - Why are Locks Bad?

- Lock-Free Programming
    - Definition, Different Lock Free Primitives
    - Examples of lockless Data Structures
    - Advantages
    - Problems ~ ABA Problem

Outline

- Lock-Free Primitives
  - Hardware
  - Software

- Read-Copy-Update(RCU) in Linux
- Lock-Free APIs in Programming Languages
  - Java
  - C/C++

- Problems with lock-free programming
- Uses of lock-free programming

Lock-Free Primitives

Lock-Free Primitives in Hardware

Ever wondered… How **CAS** instruction is implemented in hardware ?

- Using a special compare and exchange instruction (x86)
  - `CMPXCHG`

> same/similar instructions exists in other architectures. Read more

Instruction Format:

CMPXCHG reg/mem32, reg32

reg/mem32: destination register/memory

reg32: source register

How does it work ?

The CMPXCHG instruction

```
accumulator = %eax
TEMP = DEST

IF accumulator = TEMP
    THEN
        ZF := 1;
        DEST := SRC;
    ELSE
        ZF := 0;
        accumulator := TEMP;
        DEST := TEMP;
FI;
```

ZF : Zero Flag (also known as EFLAG) is a status flag in the FLAGS register. Read more

Similar instructions exist for 8 bit, 16 bit and 64 bit architectures. Read more

The CMPXCHG instruction

- The `CMPXCHG` instruction is not completely atomic !!
- It is atomic, but only on single core, not for *multiple cores*
- To make it atomic on multiple cores a special prefix is used

LOCK CMPXCHG reg/mem32, reg32

↑ prefix

Note: Don't confuse it with the lock that you have studied. It is just a part of the instruction to make it atomic across multiple cores

Lock-Free Primitives in Software

- In software, such primitives are provided by compilers
- Underlying these functions use hardware primitives
- For eg: GCC provides some atomic builtins :
  - `__atomic_compare_exchange`
  - `__atomic_fetch_add`
  - `__atomic_test_and_set`
  - `__atomic_is_lock_free`

  … and many more

Read more about them here:
https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/_005f_005fatomic-Builtins.html

We will see more such standard APIs later

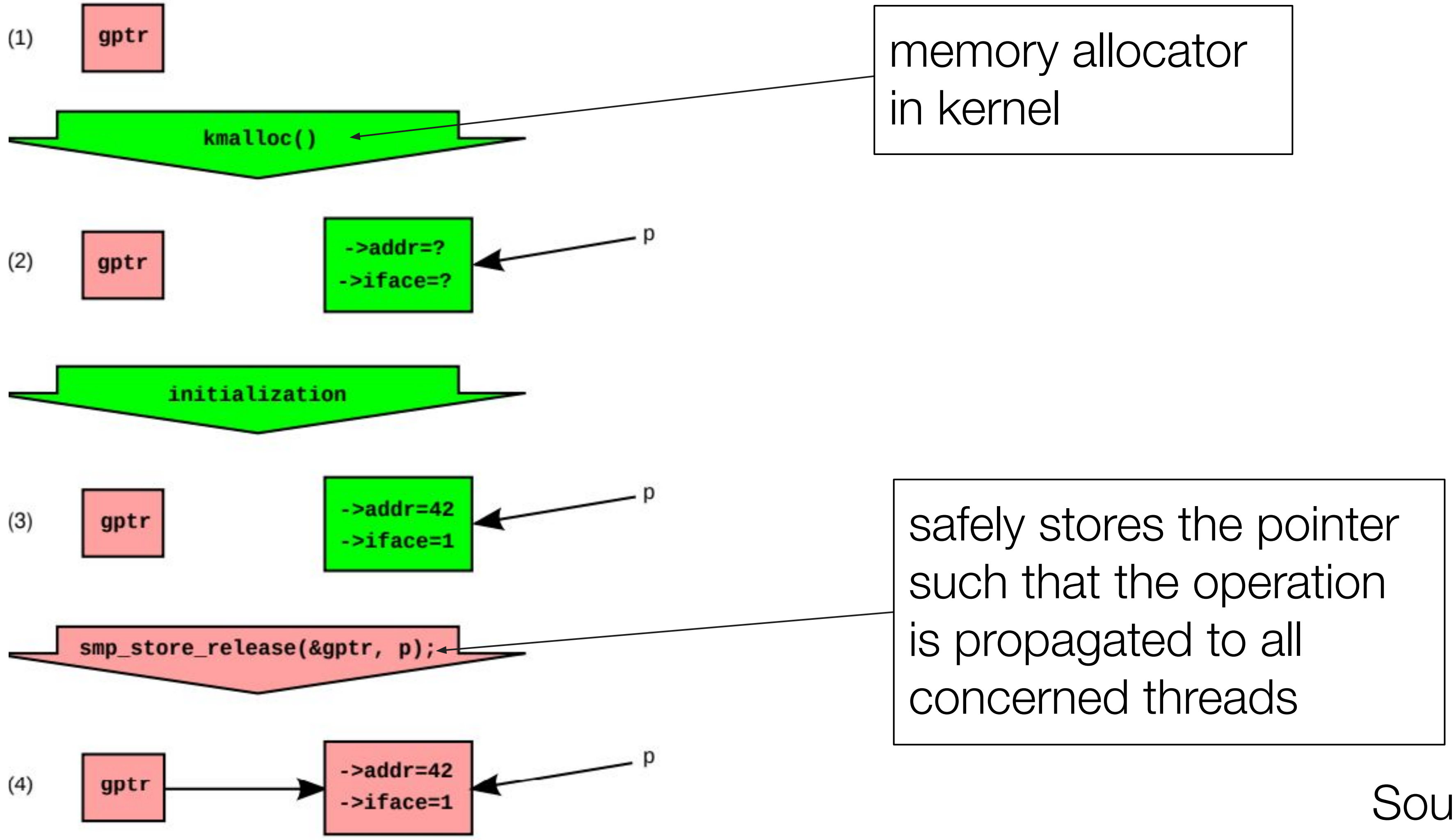
Read-Copy-Update(RCU) in Linux

Read-Copy-Update(RCU) in Linux

- Another synchronization mechanism, added to linux kernel in 2002
- Supports concurrency between multiple reader and a single updater
- no over-head from read-side primitive
- considered as one of the safest data-structures
- uses cache-line and memory very efficiently
- provides lock-free read critical section
- locks when a writer is compatible with readers
- preemption is not allowed in the read critical section

RCU : Motivation

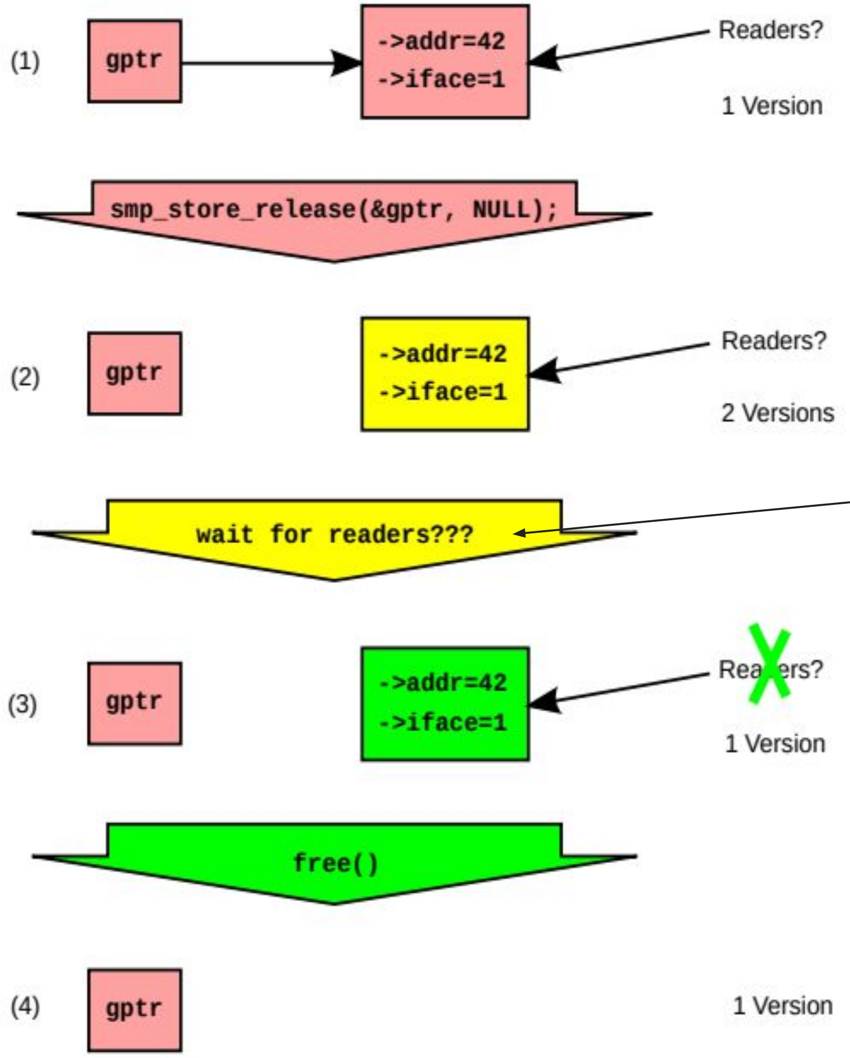
Lets see an example of simple pointer update



Source: Link

RCU : Motivation

Freeing the pointer



How do you do that ?
read sec 9.5.1.3 : Link

RCU Core APIs

	Primitive	Purpose
Readers	`rcu_read_lock()`	Start an RCU read-side critical section.
	`rcu_read_unlock()`	End an RCU read-side critical section.
	`rcu_dereference()`	Safely load an RCU-protected pointer.
Updaters	`synchronize_rcu()`	Wait for all pre-existing RCU read-side critical sections to complete.
	`call_rcu()`	Invoke the specified function after all pre-existing RCU read-side critical sections complete.
	`rcu_assign_pointer()`	Safely update an RCU-protected pointer.

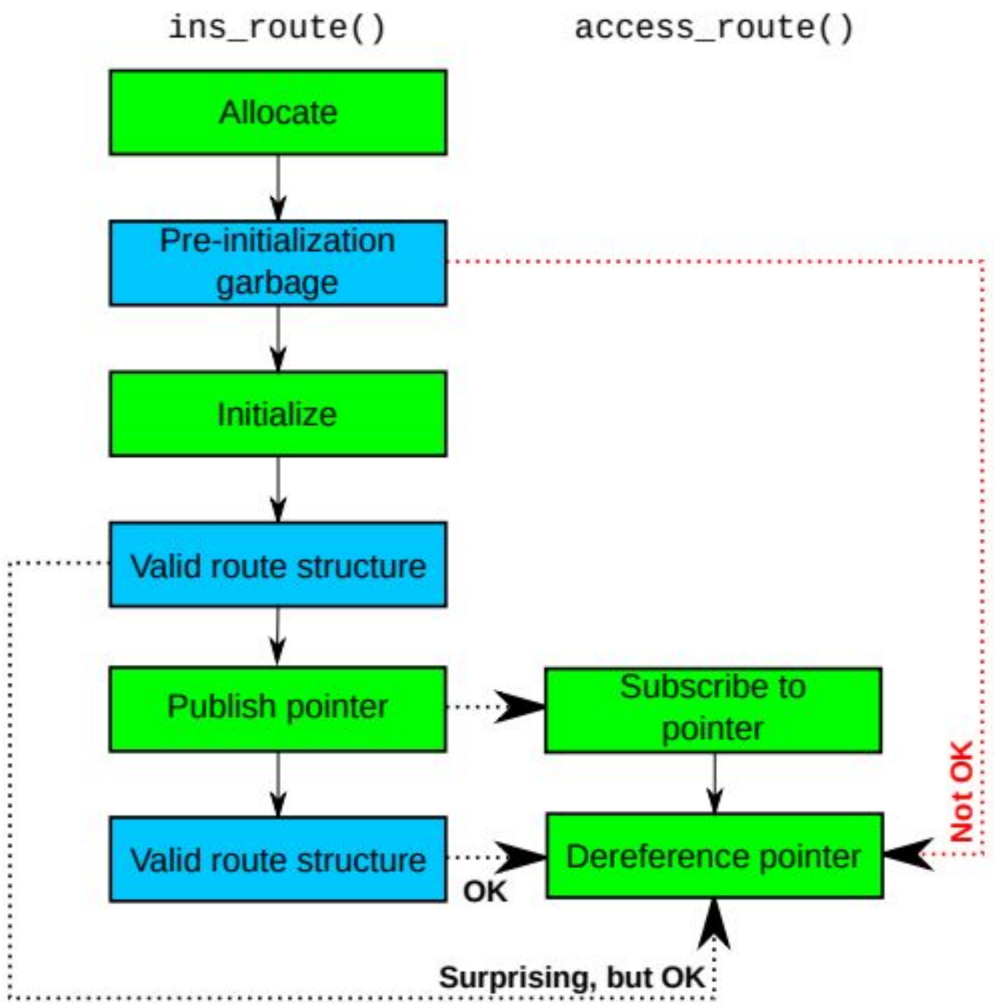
Quick Question: Can you use RCU in user space ?

Properties of RCU

- Reads need not wait for updates
  - provides low-cost/no-cost readers leading to low overhead and great scalability
  - allows RCU readers and updaters to make useful concurrent forward progress.
- Each reader has a coherent view of each object
  - Ensured by:
    - maintaining multiple versions of objects
    - using update-side primitives like `synchronize_rcu()` to ensure objects are not freed until all readers have completed

RCU Fundamentals

- Publish-Subscribe Mechanism
  - Readers are subscribers → subscribing to the current version of the RCU-protected data item
  - Updaters are publishers



RCU Fundamentals

- Wait For Pre-Existing RCU Readers

"The great advantage of RCU is that it can **wait for each of (say) 20,000 different things without having to explicitly track each and every one of them,** and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking"

- Maintain Multiple Versions of Recently Updated Objects

Outline

- Lock-Free Primitives
  - Hardware
  - Software
- Read-Copy-Update(RCU) in Linux
- **Lock-Free APIs in Programming Languages**
  - Java
  - C/C++
- Problems with lock-free programming
- Uses of lock-free programming

Lock-Free APIs in Programming Languages

Lock-Free APIs in Java

- numerous packages consisting of lock-free primitives
- one such is `java.util.concurrent.atomic`
  - provides various features like atomic integers, booleans, references etc.
  - Examples:

    i) `AtomicInteger`

    Provides atomic operations on an integer, such as `get()`, `set()`, `incrementAndGet()`, and `compareAndSet()`

```
AtomicInteger atomicInt = new AtomicInteger(0);
int currentValue = atomicInt.get();
atomicInt.incrementAndGet();
boolean success = atomicInt.compareAndSet(0, 1);
```

Lock-Free APIs in Java (contd.)

ii) `AtomicReference`

Provides atomic operations on objects or references to objects. This is useful for lock-free linked data structures

```
AtomicReference<String> atomicRef = new AtomicReference<>("initial");
boolean success = atomicRef.compareAndSet("initial", "updated");
```

Similarly, there are multiple functionalities like AtomicBoolean, AtomicLong etc. Read more

Lock-Free APIs in C/C++

- in C++ we have the `atomic` library for lock-free programming
- `std::atomic` provides various atomic data types like:
  - `atomic_int`
  - `atomic_bool`
  - `atomic_char`
  - `atomic_intptr_t`

    ...
- provides templatized access to atomic primitives
  - i.e. `std::atomic<T>`
  - can use with user defined data types (UDT) !!

Lock-Free APIs in C/C++ (contd.)

What ??

Member functions

(constructor)	constructs an atomic object (public member function)
operator=	stores a value into an atomic object (public member function)
is_lock_free	checks if the atomic object is lock-free (public member function)
store	atomically replaces the value of the atomic object with a non-atomic argument (public member function)
load	atomically obtains the value of the atomic object (public member function)
operator T	loads a value from an atomic object (public member function)
exchange	atomically replaces the value of the atomic object and obtains the value held previously (public member function)
compare_exchange_weak compare_exchange_strong	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (public member function)
wait (C++20)	blocks the thread until notified and the atomic value changes (public member function)
notify_one (C++20)	notifies at least one thread waiting on the atomic object (public member function)
notify_all (C++20)	notifies all threads blocked waiting on the atomic object (public member function)

Lock-Free APIs in C/C++ (contd.)

Specialized member functions

Specialized for integral, floating-point(since C++20) and pointer types

`fetch_add`	atomically adds the argument to the value stored in the atomic object and obtains the value held previously (public member function)
`fetch_sub`	atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously (public member function)
`operator+=` `operator-=`	adds to or subtracts from the atomic value (public member function)

Specialized for integral and pointer types only

`fetch_max` (C++26)	atomically performs `std::max` between the argument and the value of the atomic object and obtains the value held previously (public member function)
`fetch_min` (C++26)	atomically performs `std::min` between the argument and the value of the atomic object and obtains the value held previously (public member function)

Lock-Free APIs in C/C++ : Example

```
#include <atomic>
std::atomic<int> counter(0);  // Atomic shared counter
void increment() {
    for (int i = 0; i<1000; i++)
        ++counter;  // Atomic increment (lock-free)
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 100; ++i) {
        threads.push_back(std::thread(increment));
    }
    for (auto& t : threads) {
        t.join();
    }
    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

Problems with Lock-Free Programming

- Usually, it is hard to write lock-free code
- It is even harder to write correct lock-free code
- Spin-Locks causes heavy memory usage
- Overall-performance can go down in some cases compared to mutexes
- ABA Problem may occur in some cases

How to decide between lock-based and lock-free?

Lock-Based vs Lock-Free

- Observe the number of threads:
  - If number of threads are very high then lock-free may be better

- Observe the contention period:
  - If low then opt for lock-free otherwise, use lock-based (why?)

Rule of Thumb: Test your code and measure the performance

References

- Is Parallel Programming Hard, And, If So, What Can You Do About It? By Paul Mckenney
- C++ Concurrency in Action By Anthony Williams
- std::atomic - cppreference
- More about RCU - Link
- An awesome step-by-step guide to Lock-Free Programming - Link