RCU in Xv6

Semil Jain (u1417989)

Problem Statement

The focus of this project is to delve into the concept of Read-Copy-Update (RCU) and explore its efficacy in multi-threaded systems, particularly in scenarios with predominantly read-heavy workloads. The objective is to implement RCU within the xv6 operating system, drawing from the understanding gained from RCU's application in the Linux kernel. The primary goal is to assess the performance enhancements achievable through RCU by establishing a rudimentary system call structure and comparing its performance against conventional locks.

Tasks:

1) Implement Read-Copy-Update (RCU) in xv6

2) Implement a toy data structure and test this RCU

3) Modify the Adv DB class' reader-writer lock to use liburcu, benchmark, and compare the results with B+Tree.

4) Implement B+Tree based on the liburcu library.

RCU

There are a lot of Synchronization techniques between concurrent threads trying to access a shared pointer based structure that involves lock primitives. RCU is a synchronization mechanism integrated into the Linux kernel during the 2.5 development phase, tailored for read-dominant scenarios. This mechanism facilitates concurrent reading and updating of elements within shared data structures (e.g., linked lists, trees, hash tables) by multiple threads without relying on lock primitives.

Why is RCU exciting?

RCU enables concurrent reading without locks, minimizing overhead caused by lock contention in multi-threaded systems.
It facilitates non-blocking access, allowing threads to operate without waiting for exclusive access to data.
By reducing contention among threads, RCU enhances system scalability and performance.
RCU is specifically tailored for read-heavy workloads, ensuring efficient concurrent reads while effectively managing updates without compromising data integrity.

How does an RCU update typically works?

- create a new structure
- copy the data from the old structure into the new one, and save a pointer to the old structure,
- modify the new, copied, structure,
- update the global pointer to refer to the new structure,

- sleep until the operating system kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using synchronize_rcu(),
- once awakened by the kernel, deallocate the old structure.

RCU APIs

rcu_read_lock(): Marks an RCU-protected data structure so that it won't be reclaimed for the full duration of that critical section.

rcu_read_unlock(): Used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.

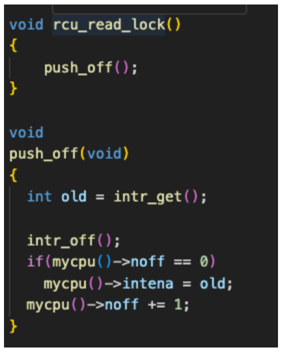
synchronize_rcu(): Blocks until all pre-existing RCU read-side critical sections on all CPUs have completed. Note that synchronize_rcu will not necessarily wait for any subsequent RCU read-side critical sections to complete.

Implementation

1) RCU in Xv6

Disabling interrupts for read locks and unlocks in xv6 could ensure a form of synchronization during reading operations, preventing interruptions from occurring when accessing shared data.

For the synchronize_rcu() API call, ensuring that each CPU allows the current process to run through once to eliminate readers might work in some scenarios. However, this method might potentially create a bottleneck or delay for other processes waiting to execute, impacting overall system performance.

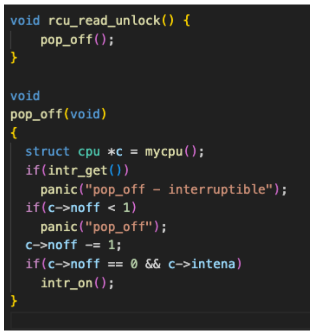


```
void rcu_read_lock()
{
    push_off();
}

void
push_off(void)
{
  int old = intr_get();

  intr_off();
  if(mycpu()->noff == 0)
    mycpu()->intena = old;
  mycpu()->noff += 1;
}
```

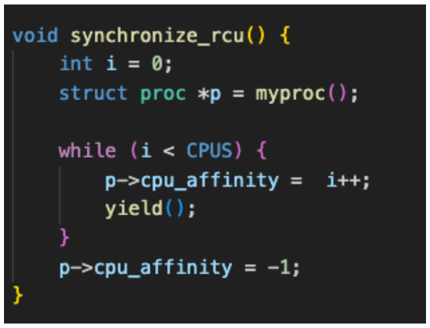
Interrupts disabled



```
void rcu_read_unlock() {
    pop_off();
}

void
pop_off(void)
{
  struct cpu *c = mycpu();
  if(intr_get())
    panic("pop_off - interruptible");
  if(c->noff < 1)
    panic("pop_off");
  c->noff -= 1;
  if(c->noff == 0 && c->intena)
    intr_on();
}
```

Interrupts enabled



```
void synchronize_rcu() {
    int i = 0;
    struct proc *p = myproc();

    while (i < CPUS) {
        p->cpu_affinity =  i++;
        yield();
    }
    p->cpu_affinity = -1;
}
```

Made changes to the proc structure and the scheduling logic to take into account cpu affinity

2) Using liburcu

liburcu is a LGPLv2.1 userspace RCU (read-copy-update) library. This data synchronization library provides read-side access which scales linearly with the number of cores.

Used the library in my Adv. Database Systems final project to implement synchronization while implementing concurrent B+Trees.

Testing and Benchmark Infrastructure

For testing in Xv6,

Implemented a LinkedList in the Kernel
Created two versions for concurrency testing:
- Reader-Writer Locks
- RCU Sync

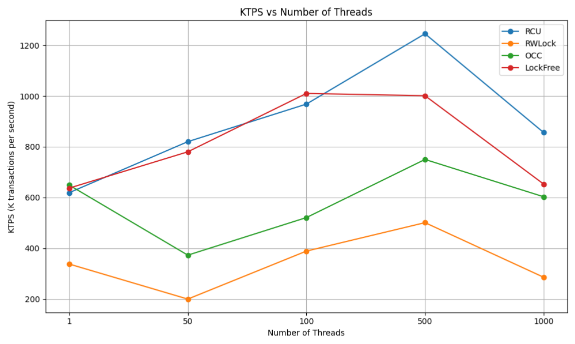
Exposed read and upserts methods as system calls
Tested both versions with multiple user processes calling the LinkedList methods

For testing liburcu,

Implemented the B+tree using liburcu.
In order to benchmark I used the YCSB read only workload with 90% reads and 10% writes on a 8 core intel machine on CloudLab.

Results

In Xv6 I wasn't able to get much difference between the two implementations. Almost got the same ticks for both Reader writer lock and RCU approach on the linked list. But got significant improvement using liburcu in B+Trees.



Challenges and Learnings

- I Was able to see real world improvements using RCU on database structures.
- Strategy for RCU synchronization: I had a number of approaches for synchronization strategy but they all came down to using variables that made it blocking. Was Finally able to decide upon a non-blocking strategy.

- Testing in Xv6: Was stuck for long on implementing multi-threading in Xv6
- Using rcd on data structures: Easier on Simple Structures like linked Lists, but challenging when implementing on something Complex like B+Trees

Future Work

In order to further improve RCU synchronization, I would like to explore some of the non blocking RCU apis like call_rcu(&callback), assign_pointer(), derefrence(), that might potentially be better than the current approach.

References and Code link

1. https://github.com/SemilJain/RCU-in-Xv6
2. https://en.wikipedia.org/wiki/Read-copy-update
3. https://www.kernel.org/doc/html/next/RCU/whatisRCU.html
4. https://www.kernel.org/doc/html/next/RCU/listRCU.html