Comparing Read Copy Update Mechanism with Read Log Update Mechanism

A Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
by

Pijush Chakraborty
(153050015)

Supervisor:
Prof. Sriram Srinivasan

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

Acknowledgements

I would like to thank **Prof. Sriram Srinivasan** for helping me out throughout the semester and helping me build a good foundation on synchronization problems. During the period, he helped me gain a lot of information about synchronization mechanisms such as Read Copy Update mechanism and also helped me learn about how research should be done in the field of concurrency.

Acceptance Certificate

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

The seminar report entitled "Comparing Read Copy Update Mechanism with Read Log Update Mechanism" submitted by Pijush Chakraborty (153050015) may be accepted for being evaluated.

Date: 3 May 2016

Prof. Sriram Srinivasan

Abstract

In this era of multiprocessor systems, multithreaded programming is common and is used everywhere. To support synchronization between them while accessing shared memory regions, various mechanisms has been build. In this paper, we will discuss two such mechanisms, **Read-Copy-Update(RCU)** mechanism and **Read-Log-Update(RLU)** mechanism. The two mechanisms never blocks readers and allows them to iterate and not worry about data inconsistency. While RCU has been used over a decade in the Linux Kernel, RLU is new and came into existence just a few months ago. We will discuss how RLU can eventually improve performance by implementing a more scalable reader-writer synchronization.

Table of Contents

List of Figures

Chapter 1

The Main Problem

In this new age of processors, multicore processors and also multiprocessor(SMP) systems have made **multithreaded processes** run efficiently and fast. Threads can now run in parallel on different processors allowing them to finish a task efficiently. To allow such processes to work, the treads must communicate with each other to ensure proper synchronization between them so that and the end result is consistent and as good as a single thread execution of the process. Multithreaded programming offers a significant advantage in terms of efficiency but other methods for proper communication is required. The need for such synchronization and a particular problem with multithreaded programming can be described below.

1.1 Reader Writer Problem

In most of the cases, multiple threads that run in parallel, accesses a shared memory region. Suppose, there is a memory region **(Memory-A)** and two threads are simultaneously accessing it in order to make changes or read from it. Now, one of the threads **(Thread-A)** is trying to read data from the region while the other thread **(Thread-B)** is trying to update the same region as shown in Figure 1.1.

It will lead to chaos, if Thread-B (writer) updates the region while Thread-A (reader) is still reading. This may result in Thread-A having inconsistent data and may lead to improper execution of the process. It may get even worse if Thread-B removes the region while Thread-A is still accessing it. To save the threads from such problems various synchronization methods have been developed. This methods ensure that the threads get a consistent final result even if all the threads run in parallel. A simple solution to the

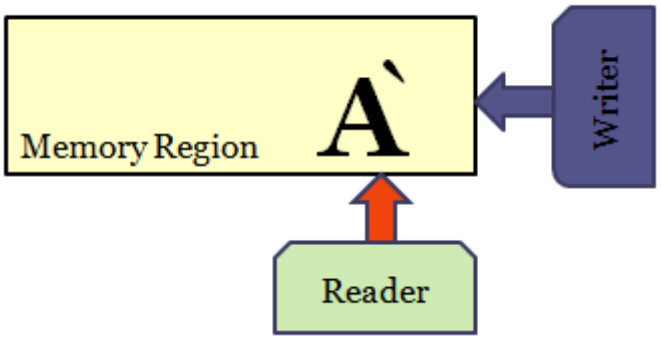

Figure 1.1: Reader Writer Problem

problem is the use of Reader Writer locks. The locks work simply by blocking threads accessing the memory region as described below.

1.2 Reader Writer locks

To make things simpler for threads working together and to find a simple solution to the Reader-Writer problem, a simple locking mechanism is usually implemented. This locking mechanism uses a shared exclusive lock for readers and writers. Since readers doesn't harm other parallel readers, they can run in parallel without making the data inconsistent. To ensure this readers get a read lock and increment the lock count on the memory region. Writers that try to access the memory regions waits for all the readers to release their lock and in the meantime waits in a write queue as shown in Figure 1.2.

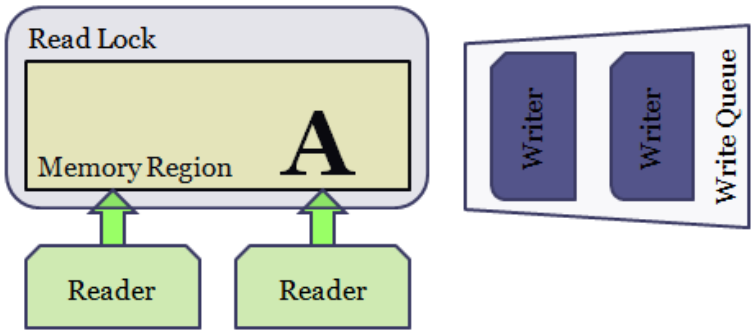

Figure 1.2: Reader Writer Locks

When all readers exit, a single writer from the queue gains a write-lock on the object to ensure that no other threads can access the same region until it releases the lock. A writer

must block other writers too so that it can update the memory region without any worry about parallel updates which may again make the data inconsistent. So a writer gains an exclusive lock on the memory region so that other writers and also readers waits for it to release the lock as shown.

Thus, a reader block only writers waiting to update the region while a single writers blocks all other threads. This simple mechanism provides a small and trivial solution to solve the reader-writer problem. But, this mechanism is not sufficient in some scenarios which will be described in the next section and a need for some more efficient solution will arise.

1.3 Issues with Reader-Writer Locks

The locking mechanism described above is quite a simple solution to the reader-writer problem but doesn't quite provide a elegant solution for the following cases:

- It doesn't prove to be efficient for read mostly data structures as a single writer may block multiple readers. Data structures such as the per process information list used by the kernel is accessed often and the simple locking mechanism may prove to be inefficient in terms of performance.
- As a reader blocks all writers, a data structure update take a lot of time to finish in order to maintain consistent data.

To address the above problems, various synchronization primitives have been designed and developed. The next chapter discusses once such important design, **Read-Copy-Update(RCU)** mechanism that enables readers and writers to run in parallel and still provide consistent data. While reader-writer locks tries to lock the same memory region, RCU writers doesn't change the original memory region and maintains **multiple versions** of the same memory region or the RCU protected data structure.

Chapter 2

Read Copy Update

Read-Copy-Update McKenney and Slingwine (1998) is another synchronization mechanism that solves the reader-writer problem with certain advantages in place. RCU favors readers over writers and allows multiple readers to access the protected data structure with no worry about other parallel writers. Readers are never blocked by any threads and nor do readers block writers as described in the below sections. RCU also provides a suitable way for memory reclamation to avoid any possible memory leak. Lets move onto how the RCU works.

2.1 Introducing RCU

RCU provides an effective and efficient solution to the reader writer problem allowing multiple readers and writers to access the data structure. The mechanism never blocks the readers and is efficient for protecting read mostly data structures. RCU readers normally accesses the critical section without worry.The RCU mechanism mainly offers two main primitives for readers.

The first primitive, **read-lock** is one which a reader is supposed to use this lock before accessing the data structure. Normally, as will be described later, this primitive doesn't do anything except for updating the read count. The second primitive, **read-unlock** is the one using which a reader should unlock the data structure or decrements the read count and start doing other important work. Again, this will simply help the writer make some important decision as we will see in later sections.

As for the RCU writers, RCU protected data structures must be updated so that readers are not harmed and must not be able to see any inconsistent data. To make this work, Classical RCU approaches a simple mechanism of three steps:

- **Read:**
  Read or iterate over the data structure to find the memory region for updating it.
- **Copy:**
  Copy the memory region to a different location and then fiddle with it. This allows readers to work on the old memory region and cannot see any inconsistent data.
- **Update:**
  Update the old memory region atomically to the new memory region so that any new readers that comes along will see the new memory region and will again see consistent data.

In simple words, the writers copies the memory region, fiddles with it and finishes off by atomically replacing the old memory region. The main point to be noted is that, RCU maintains multiple versions of the same data structure at a particular point of time. The RCU writer update is done in a way to ensure readers are not blocked and it only provides one extra primitive, **synchronze-rcu**. This primitive is another important writer primitive that deals with removing old unused memory regions after all the old readers releases their lock on the data structure. This primitive will be discussed in detail in later sections.

2.2 RCU Phases

As described in the earlier section, the **RCU mechanism** maintains multiple versions of the data structure and ensures that new readers get to see the new updated data while old reader can still access the old data. This mechanism provides complete freedom to the readers to iterate and read a consistent data. To achieve this, the **RCU Writer** works in three notable phases for updating a datasrturcture as described below.

2.2.1 Three Major Phases

As discussed, RCU works in three major phases to provide readers the flexibility of working without worrying about consistency. The three major phases are as following.

- **Removal Phase:**
  This phase is where the old memory region is read, copied and atomically updated

so that new readers can see the updated version. The removal phase is simple and the completion of this phase makes multiple versions of the same data structure. Moreover, after this phase the readers are classified into old and new readers. The old readers are the one that started before the completion of the removal phase and the new readers are the threads that started accessing the data structure after the phase is completed. Figure 4 shows the removal phase and the classification between the readers.

- **Grace Period:**
  As soon as the removal phase is complete, new readers can see the updated version of the data structure. But, even now the old memory region may still be accessed by some old readers forcing the writer to wait for the old readers to exit before freeing the old memory region so that it can be reused. The grace period ends when the readers that started before the beginning of this phase(old readers) releases their lock. This phase is one of the important phase to prevent memory leak and the **Classical-RCU** approach to determine the period is discussed in later sections. This phase is dealt with the RCU primitive, **synchronize-rcu**.

- **Reclamation Phase:**
  This phase begins when the writer is sure that the old readers are no longer accessing the old memory region and it is safe to free the old memory region without any consistency problems. This phase simply frees the old memory region so that it can be reused later.

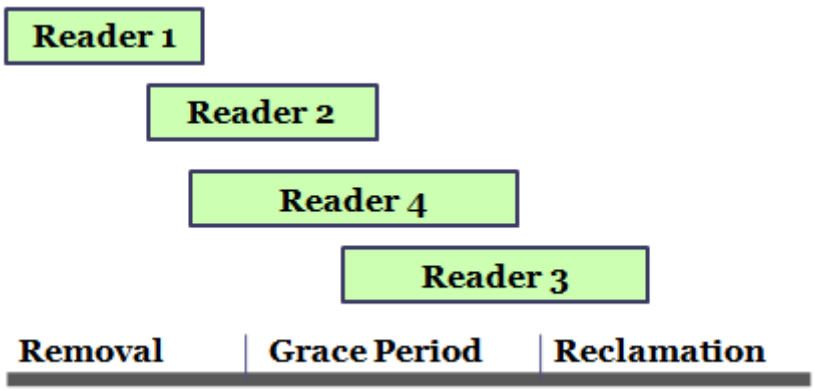

Figure 2.1: Phases in Read-Copy-Update Mechanism

As shown in Figure 2.1, the removal phase is where all the the functional work is carried out and after which the list is updated and is consistent for new readers to access.

Now, since C doesn't have any garbage collection methodology, it all boils down to the writer for preventing memory leak. The second and third phase takes care of the garbage collection and frees the old memory region to be reused later.

2.2.2 Understanding the RCU Phases

The RCU mechanism has been described in earlier sections and the RCU writer phases have also been described. This section shows how the mechanism protects a simple linked list data structure. Suppose, the linked list in Figure 2.2, is protected by RCU and has two readers reading and iterating the list.

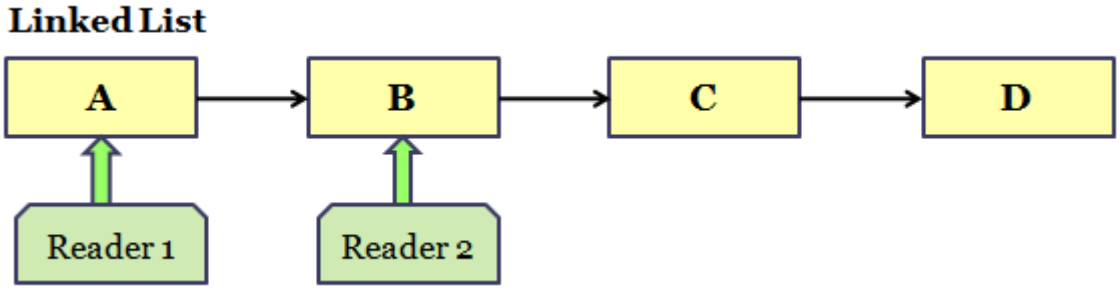

Figure 2.2: RCU Protected Linked List

A writer thread has just started and wants to update the third node(node 'C'). The thread simply copies the node, updates it and atomically replaces the old node from the entire data structure as shown in Figure 2.3.

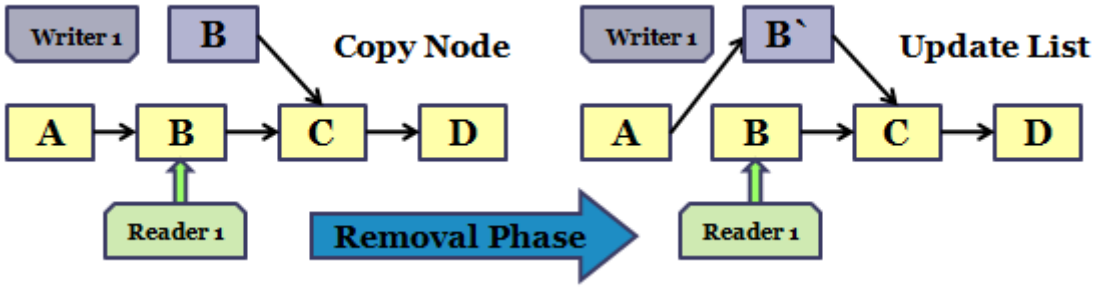

Figure 2.3: RCU Removal Phase

As shown in Figure 2.3, the simple atomic change of the next pointer of node 'B' atomically places the updated node 'C1' in the data structure. At this point there are two versions of the data structure starting from node 'A' and another version starting from node 'C'. Old readers still has access to node 'C' while new readers can access the original linked list having the updated version of the data(node 'C1'). This ends the

removal phase and marks the start of the grace period.

The grace period waits for the old readers to release their locks on the memory region as shown in Figure 2.4. After the grace period ends, the writer can be sure the old memory region no longer has any readers referring to it and then it can proceed to memory reclamation phase.

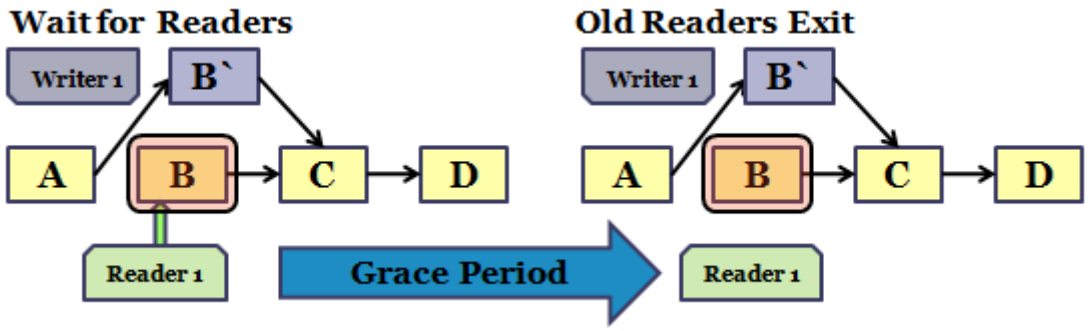

Figure 2.4: RCU Grace Period

The final phase or the reclamation phase is shown in Figure 2.5, where the old memory region is removed to ensure there is no memory leak.

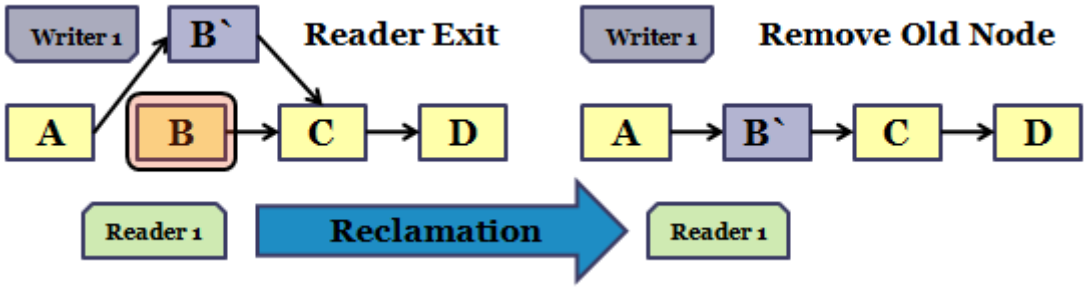

Figure 2.5: RCU Reclamation Phase

After the end of the last phase, the linked list remains consistent and the writer has finished the update. The old memory no longer can be accessed from the linked list and is no longer a part of the data structure. The next section describes how the grace period can be described and how the Classical-RCU determines it.

2.3 Calculating Grace Period

There are various implementations of calculating the grace period and it depends on the design of the **RCU synchronization mechanism** to provide a way for the implementation. Determining the grace period is one of the important part of the RCU mechanism to

prevent memory leak and the original paper 'Classical-RCU' had a unique way to do the same.

2.3.1 Classical RCU Implementation

The classical RCU way of determining grace period depends mostly on the way the readers work and is significant in most cases. In Classical RCU, readers are non-preemptive. This forces the readers to unlock the data structure when exiting the critical section. So a reader read side critical section implementation can be shown as follows:

read-lock // For Classical-RCU, it disables preemption on the current CPU
...Access critical section
read-unlock // It simply enables preemption on the CPU

The above **non-preemptive** nature is utilized by the writer thread in an unique way to determine the end of grace period. It can be easily said that since the reader threads are non preemptive, a context switch on the CPU means that the reader thread has released its lock. If a context switch occurs on all the CPUs present in the SMP system after the start of the Grace Period, it can said that the old readers are no longer present and the old memory region is no more accessible and is safe to free the region. This is the strategy used by the writer thread for determining the end of the grace period. The writer thread basically tries to run itself on all the CPUs and succeeding which, it can end the grace period and move on to the reclamation phase. The writer thread can be summarized as follows:

Removal Phase // The main RCU update part
Start of Grace Period
for-all-cpu
run-on(CPU) // Try to make a context switch on all CPUs
End of Grace Period
Reclamation Phase // Safe to free the old memory region

The Classical-RCU implementation is the first RCU implementation which prevents reader threads from preempting and the solution though unique doesn't scale when the no of readers becomes more. To solve this problem, another approach to determine the grace period is made in Sleepable-RCU McKenney (2006b) implementation which deals

with reference counter based mechanism to determine if a memory region is free for removal.

2.4 Issues with RCU Mechanism

The Read Copy Update Mechanism solves the reader-writer problem in an efficient way and favors the readers allowing them to access the data structure without any worry about consistency. To provide this mechanism, normally writer synchronization is left to the developers using the RCU mechanism and these causes various problems as seen below:

RCU is not an efficient solution for data structures with more that one pointers such as a doubly linked list(Figure 2.6). The list shows that the removal phase is in process and a writer has copied Node-B and updated it. It now need to replace this updated node in the original linked list and for this the writer has to make two pointer assignments for **Node-A->next** and **Node-C->prev**. Now, it must be noted that a single pointer assignment is atomic but assigning multiple pointers can not be atomic and leads to the state as shown in the below figure.

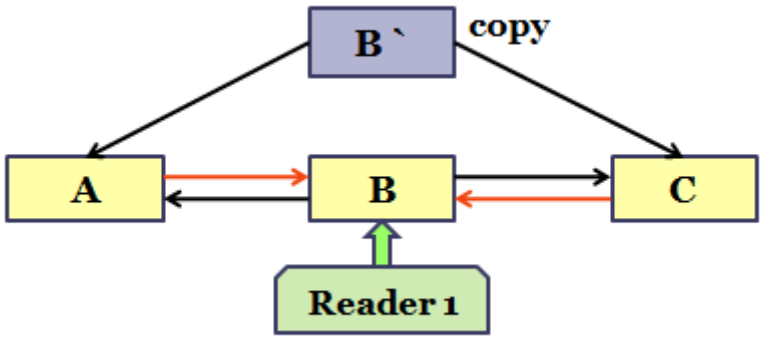

Figure 2.6: Issues with RCU

Here, To make the updated copy Node-B1 successfully replace the the old memory region Node-B, another pointer assignment must be done. At this point an old reader accessing Node-B has access to both the original and the updated version of the list which again provides inconsistency for the overall data structure.

The problems are solved by another new synchronization mechanism, Read-Log-Update(RLU) as described in the following chapters.

Chapter 3

Read Log Update Mechanism

Read-Log-Update mechanism(RLU) Matveev *et al.* (2015) provides another efficient method for maintaining synchronization in read mostly data structures. Much like RCU, it never blocks the readers and at the same time provides better flexibility to writers. RLU allows multiple writers to make updates using a log based mechanism and can commit multiple changes to the data structure atomically. It overcomes the problem that RCU faces and provides an efficient version of synchronization as can be seen in later sections.

3.1 Introducing RLU

RLU has a significant advantage over RCU in terms of performance as the no of operations performed on a particular time frame is particularly more on RLU as writers are allowed to work together with RLU readers on a RLU protected data structure. The most important advantage of RLU is that it can commit multiple objects atomically to provide readers accessing the data structure, consistent data over the entire data structure. As shown below in Figure 3.1, if RLU mechanism used the same concept of copying the memory region to update, then an old RLU reader will see **A->B->C-D** and a new RLU reader will see **A->B1->C1->D** as the protected list.

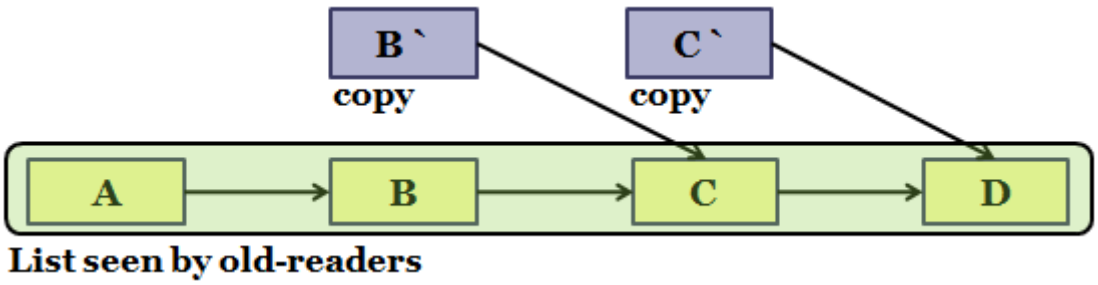

Figure 3.1: RLU Atomic List Update

This consistency is maintained using some lock based and clock mechanism described in later sections. Unlike RCU, where a reader may see any combination of the nodes, RLU presents a consistent data structure where a single operation allows multiple commits possible. Lets see the data structures used by RLU in order to achieve such consistency.

3.2 RLU Data Structure

RLU maintains various data structures in order to achieve the goals described in the earlier section. The data structures used by the RLU mechanism are given below:

- **Global Clock:**
  This is basically a software clock and it maintains a time-stamp denoting the current version of the data structure. Whenever a reader gets a lock on the region, it initially reads the current version of the time-stamp to later use it for iterating and finding the right memory region. The clock also denotes what each new thread should save as its version number.

- **Per Thread Data Structures:**
  RLU also has some per thread data structures as can be seen in Figure 3.2. The per thread data structures are useful for the thread to know how to use the data structure and accordingly make decisions.

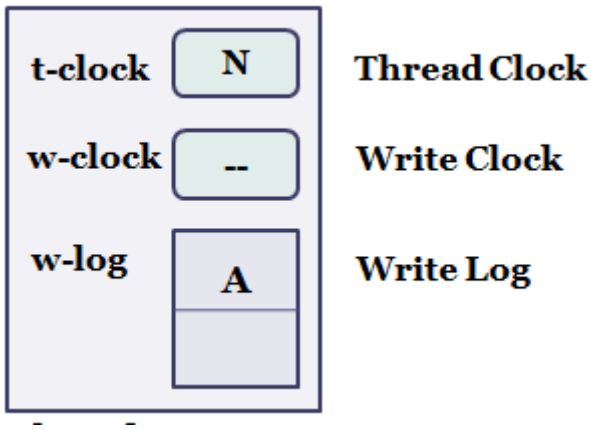

Figure 3.2: RLU Per Thread Data Structure

The per thread data structure is quite important for maintaining synchronization among threads. Each thread maintains a per **thread clock** which it initializes from the global clock. This thread clock validates the time when the thread has started reading and

according make decisions in the long run. The second per thread data is the **write clock** which is used by the thread when its updates a memory region. The write clock signifies the time stamp when the writer has issued a commit to signify that it has completed the update. Each thread also maintains a **write-log** which is simply holds a copy of the memory region to be updated much lick what RCU does. The log maintains copies of all memory region it wants to update.

Along with the above data structures, each memory region has an associated header that stores information related to the writer thread. The important information includes which writer-thread currently has a lock on the region and a pointer to the updated copy of the memory region in the log.

All the data structures described above is used by the threads to maintain synchronization and will be described in later sections. Another important point to note is that RLU objects are protected by fine grained locking which allows the multiple writers to work on the same data structure with fine grained locks on each node to allow multiple writers update the same shared data structure. The next section describes how threads use the data structures to maintain synchronization and consistency.

3.3 Putting Everything together

This section present how various readers and writers use the RLU data structures described to maintain synchronization and allow consistency. The section starts with how RLU readers access a particular memory object and then moves on to how writers updates the same region.

3.3.1 RLU Readers

RLU Readers are not blocked by other threads and are provided a consistent view of the entire data structure. The reader can be summarized as below:

- The reader when started initializes its per thread clock with the global clock. This is to determine when the thread began reading.
- It then proceeds to iterate the data structure such as the linked list. If it finds a memory region currently locked by ta writer thread, it checks if **Reader.t-clock>=Writer.w-clock** and if so reads the data from the writer log. The compari-

son basically determines if the reader thread started before the writer committed or after the commit.

So, as with RCU, old RLU readers too get to access the old memory region while new readers gets to see the new memory region. RLU Writers work in a different way as shown below.

3.3.2 RLU Writers:

RLU Writers work in a different way. The writer thread can also be summarized as shown below:

- It initially starts by initializing its per thread t-clock with the global clock and its w-clock to infinity(possibly a large value) and then iterates the data structure to the memory region which it wants to update.
- The writer updates the memory region header and updates the lock field to its own thread-id. It then copies the data to the write-log and updates it. The writer commits by setting the w-clock=g-clock+1 and the global clock(g-clock) is also incremented by 1. The commit clock update is done to ensure that any new readers that tries to read a locked object can read the updated data in the log as any new reader will have its per thread t-clock more than the locking thread's w-clock. Thus the writer maintains consistency and as RCU writers relies on atomic pointer assignments, RLU writers relies on atomic clock updates.

After the writer thread commits, the write-log consists of the updated data which must now be copied back to the original memory region. The writer now has to wait for all the readers to exit and this phase is similar to the RCU Grace Period. Moreover, the RLU Grace Period can be determined the same way as the RCU Grace Period. RLU Writers may defer the write back or delay the copying of the updated data to the original memory. It may be noted that as in **Hazard Pointers** Michael (2004), RLU uses the same strategy to use a reference to a log in order to defer its removal.

Chapter 4

Comparing RCU and RLU

Both RCU and RLU offers an efficient way for synchronization between readers and writers and favors readers in read mostly data structures. While RCU depends on atomic pointer assignments, RLU depends on atomic clock update which proves to be much more efficient as will be shown below. Moreover, RLU Writers can work in parallel due to fine grained locking per object. RCU Writers may not scale well with fine grained locks as described earlier and thus to enable multiple writers, proper synchronization must be provided by the developers using RCU mechanism.

As discussed, RLU writers can update multiple objects atomically with just a single clock update as shown below in Figure 4.1.

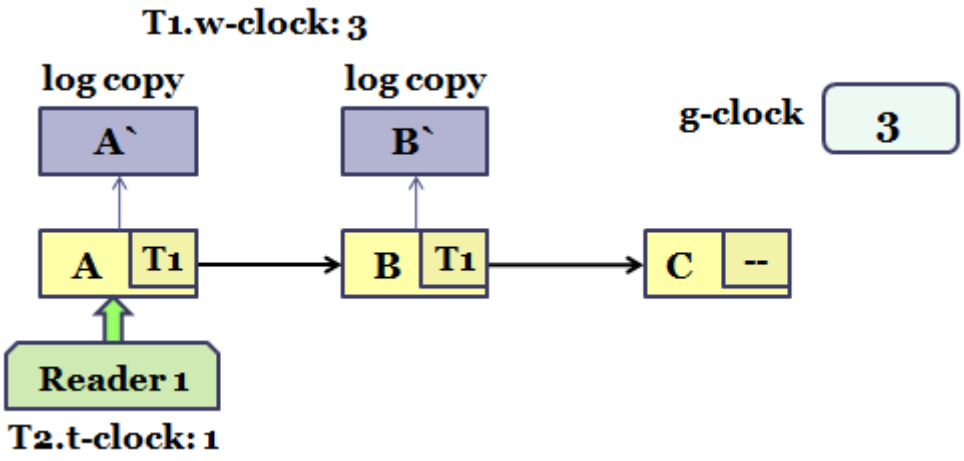

Figure 4.1: Multiple Object Commit

In the above figure, a single w-clock update enables new readers to access the updated memory regions while old readers still gets to use the old memory region.

RCU has been used in the Linux kernelMcKenney (2006a) and various implementation of RCU has been developed to support the same. Though RCU has been proved to be efficient in terms of implementation, it can be said that RLU provides much more advantage in terms of efficiency and performance as it enables both readers and writers to work in parallel without much complexity in implementation.

References

Matveev, A., Shavit, N., Felber, P., and Marlier, P., 2015, "Read-log-update: a lightweight synchronization mechanism for concurrent programming," in *SOSP*

McKenney, P. E., 2006 Octobera, "Read-copy update (RCU) usage in Linux kernel," available: `http://www.rdrop.com/users/paulmck/RCU/linuxusage/rculocktab.html`.

McKenney, P. E., 2006 Octoberb, "Sleepable RCU," available: `http://lwn.net/Articles/202847/`.

McKenney, P. E., and Slingwine, J. D., 1998 October, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems* (Las Vegas, NV). pp. 509–518.

Michael, M. M., 2004 June, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems* **15**, 491–504.