Concurrent data structures using RCU

Rolfe Bozier

19-Oct-2014

Agenda

- Wait-free data structures
- How does the RCU mechanism work?
- A hash table with wait-free readers
- Conclusion
- References

RCU – Read-Copy-Update

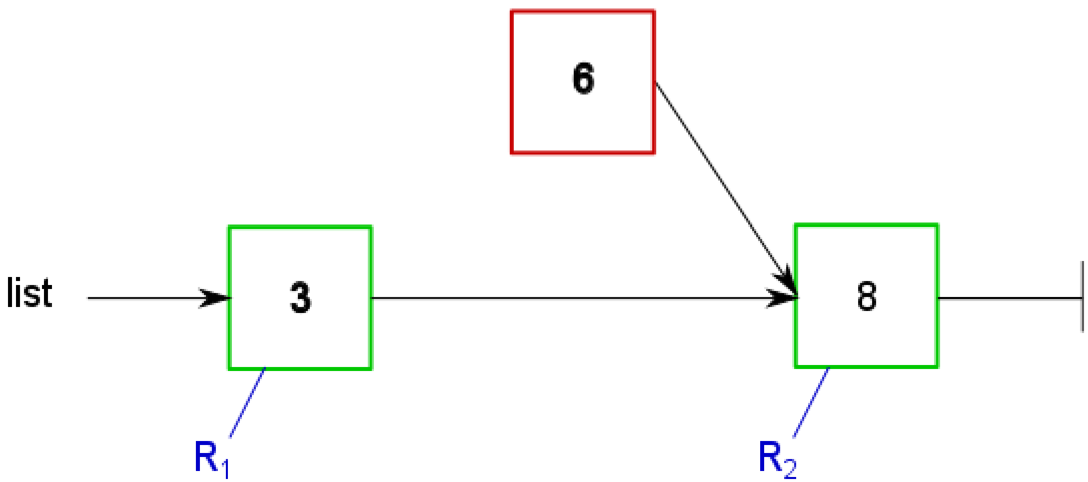
- What is RCU?
  - A synchronisation method that can often be used in place of a traditional reader-write lock
  - Used to protect data structures that are written and read at the same time
  - Extremely low overhead for readers – wait-free
  - Readers cannot block writers
  - Multiple writes are serialised
- What are the drawbacks then?
  - Multiple versions of the data structure can be present at any one time
  - Concurrent readers can see different [valid] versions
  - Some memory overhead in keeping older versions of the structure around (for a period)
- RCU has been used in the Linux kernel for over 10 years
- RCU requires 2 pieces of functionality:
  - Publish-subscribe mechanism
  - Grace periods

Publish-subscribe

- Consider the following stages in adding an element to a shared linked list
  - There are two readers, currently at the first node

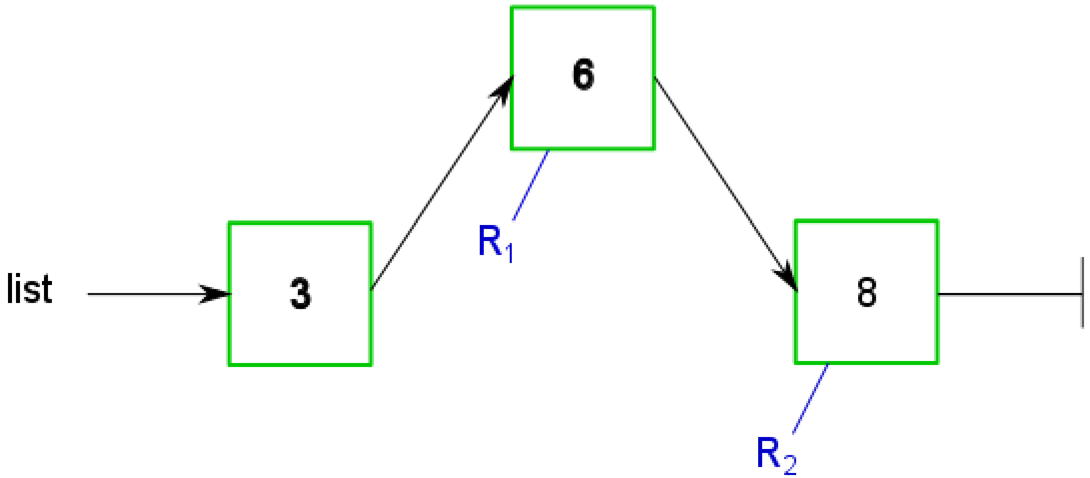
Publish-subscribe

- We create a new node and set its next pointer to the correct destination
  - No reader can see the new node yet
  - Meanwhile, the second reader has moved to the last node



Publish-subscribe

- Finally, we *atomically* update the previous nodes next pointer to include the new node
  - The first reader follows the pointer to the new node (and then finally to the last node)



Publish-subscribe

- OK, this seems pretty simple, but…
  - Don't forget about compiler and CPU re-ordering of instructions!

Writer

```
prev = …
p = malloc(sizeof(node));
p->value = 6;
p->next = prev->next;
rcu_assign_pointer(prev, p);
```

Add a write barrier before setting `prev->next`

Reader

```
list_for_each_entry_rcu(p, head, next)
{
    do_something_with(p->value);
}
```

Add a read barrier for each step through the list

Publish-subscribe

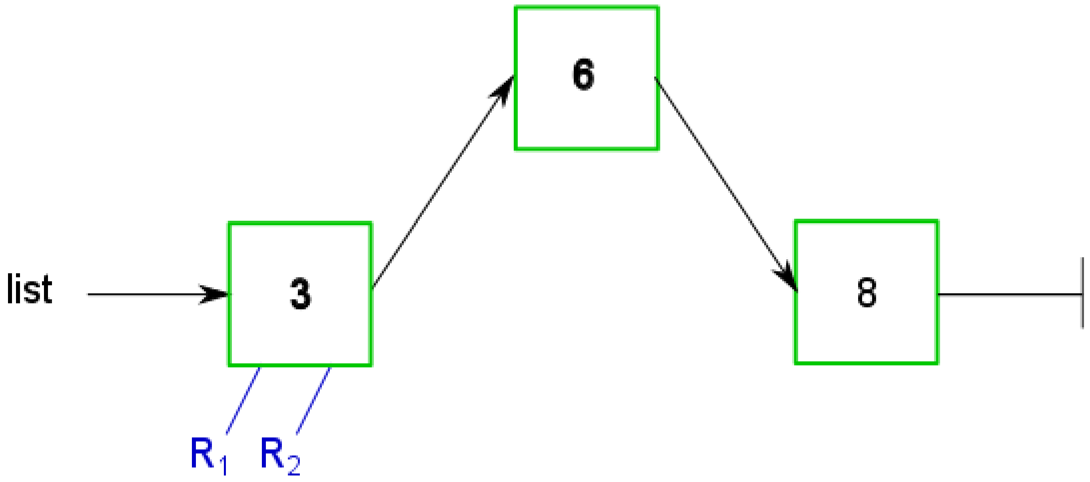
- Different readers can see different views of the list at the "same time"
  - Reader 1 saw: 3, 6, 8
  - Reader 2 saw: 3, 8
- Publish semantics:
  - Writer updates the data structure atomically
  - Writer uses memory barriers to prevent store re-ordering
- Subscribe semantics
  - Readers are prepared for things to change on the fly
  - Readers use memory barriers to prevent load re-ordering

Grace periods

- Deleting an item from an RCU list is not quite so straightforward…
  - Remember that readers can be looking at any node in the list while we are deleting
  - … including the node we want to remove!
- To allow for this, we preserve the existing data for a grace period, to allow readers to complete
  - After the grace period has expired, we can delete the old node
  - Obviously we need something a bit more rigorous than a “timeout”

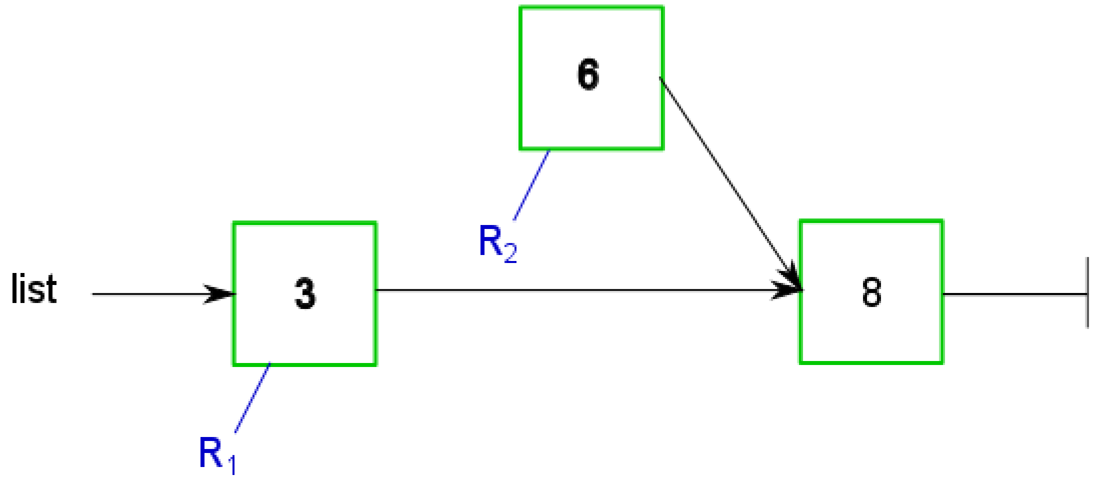
Grace periods

- Consider the following stages in deleting an element in a shared linked list



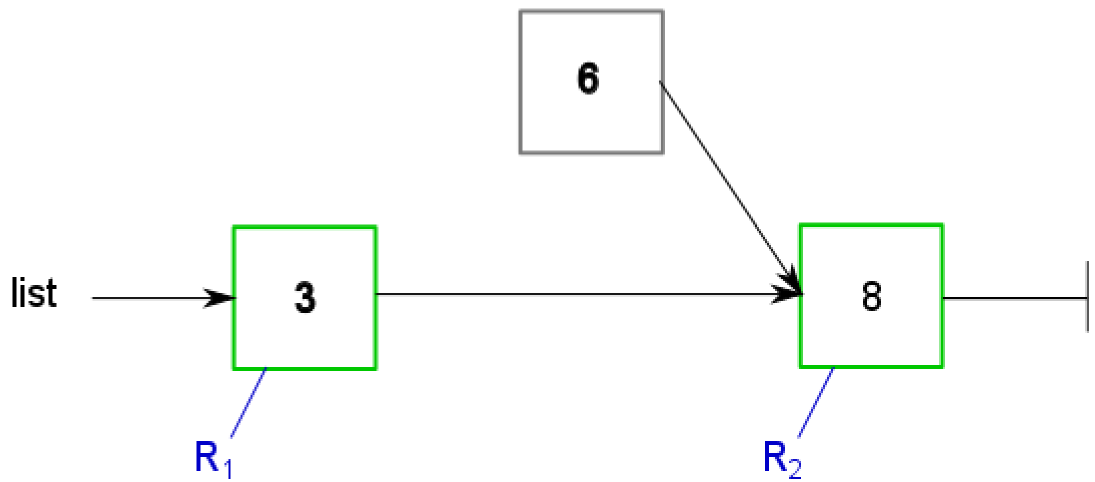
Grace periods

- The next pointer in the previous node is updated to bypass the target node
  - But before this, reader 2 has traversed to the target node



Grace periods

- The writer waits for a "grace period"
- After the grace period has expired, there can be no readers still referencing the target node
  - Reader 2 has moved to the final node



Grace periods

- The target node is freed
  - Meanwhile the first reader traverses the list to the final node

Grace periods

- Different readers can see different views of the list at the "same time"
  - Reader 1 saw: 3, 8
  - Reader 2 saw: 3, 6, 8
- The grace period allows the writer to safely perform garbage collection of the deleted node
  - This needs to be long enough for all readers to no longer reference the deleted data
  - It should not be so long as to hold up the writer or cause excess memory usage
- If we want to *update* a node, we generally can't do this atomically (unless the data is atomic)
  - **Read** the node to be replaced
  - **Copy** it to a new node with the modified values
  - **Update** the list atomically to use the replacement node

Grace periods

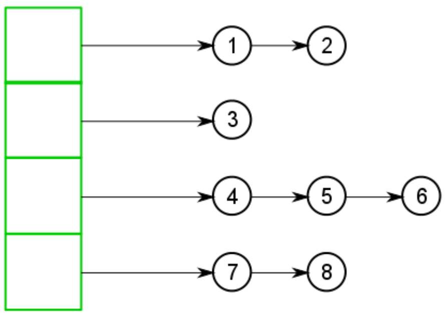
- In the Linux kernel, grace periods are managed as follows:
  - Readers enclose their operations in `rcu_read_lock()` and `rcu_read_unlock()` "functions" - in the simple case they do nothing
  - Code must *not* block between these two functions
  - The write thread schedules itself to run on each CPU in turn
  - At the end, it is guaranteed that no reader is still holding on to a deleted node
  - The node can be freed
- Obviously something different needs to be done for userspace implementations of RCU
- There are now a number of userspace libraries that can manage RCU operations in various ways, e.g.
  - A global 64-bit grace period counter that can be atomically incremented, plus
  - Per-thread grace period counter references to indicate where the reader is
- See the references for more information

An RCU Hash Table

- Consider the example of an RCU Hash Table
  - In open hashing, collisions are resolved by adding a linked list to the bucket
  - We can see that RCU lets us add, modify and delete entries to a hash table, including the bucket lists
- But what about resizing?
  - We can resize a hash table without blocking readers with the following conditions:
    - The table grows or shrinks by a factor of two
    - Bucket lists can be *imprecise* – they can contain extra entries (from different hash values)

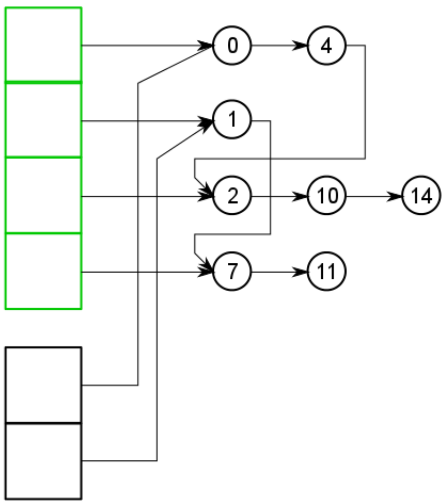
An RCU Hash Table

- Consider a hash table with 4 buckets
- We want to shrink it to size 2



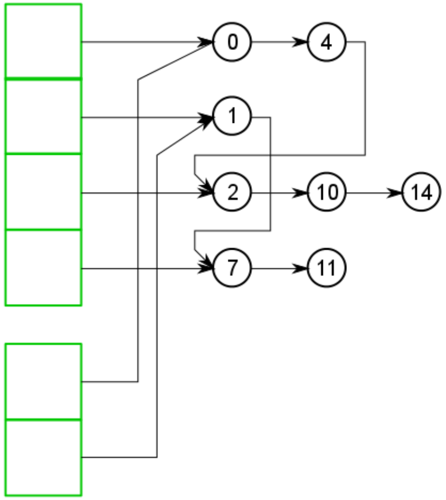
An RCU Hash Table

- Create the new bucket array
- Chain the bucket lists according to their new hash value



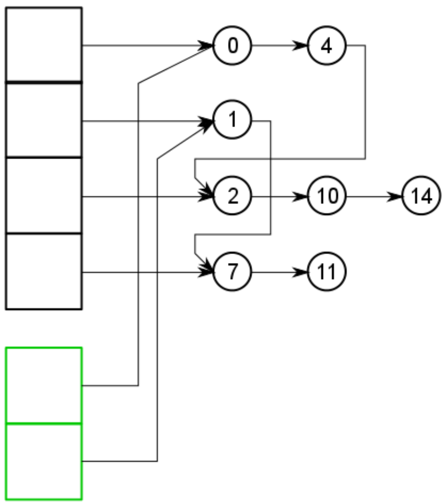
An RCU Hash Table

- Publish the new table and wait for readers



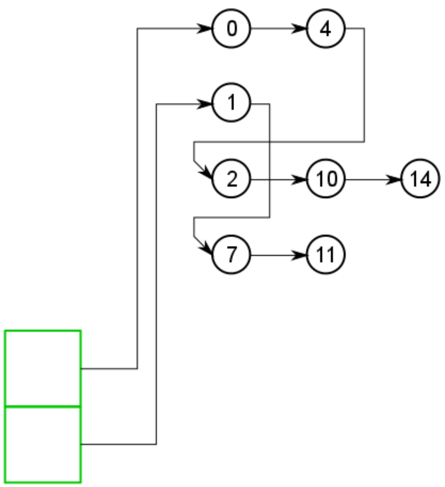
An RCU Hash Table

- The old bucket array can now be freed



An RCU Hash Table

- The final version



- Growing is essentially the reverse process
  - It is a bit more involved due to the need to unzip the bucket lists and wait for readers for each step

Conclusion

- RCU is a synchronisation mechanism for managing data structures with wait-free readers
  - Most suitable where reads >> writes
  - The data structure semantics may need some adjustment (e.g. imprecise bucket lists)
- It has been used in Linux kernels for over 10 years
- To implement, you need a few things
  - Atomic operations
  - Memory barriers
  - A suitable grace period mechanism

References

- What is RCU, Fundamentally? [3 parts]
  - http://lwn.net/Articles/262464/
  - http://lwn.net/Articles/263130/
  - http://lwn.net/Articles/264090/
- “User-Level Implementations of Read-Copy Update”
  - https://www.efficios.com/pub/rcu/urcu-main.pdf
  - https://www.efficios.com/pub/rcu/urcu-supp.pdf
- Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming
  - http://www.usenix.org/events/atc11/tech/final_files/Triplett.pdf
  - http://works.bepress.com/jonathan_walpole/12/