Chapter 1

Lockless radix-tree

One of the requirements of a lockless pagecache is a pagecache data structure that does not require locking. A lockless [1] radix-tree is presented in this chapter.

1.1 Read-Copy Update (RCU) overview

Read-Copy Update (RCU) [2] is an algorithm that was primarily developed to permit a shared data structure to be read without using locks. In this capacity, RCU's main objective is to provide an existence guarantee to the reader: that is, that the item or node currently under examination by the reader is not concurrently freed.

> Read-copy update provides a grace period to concurrent accesses by performing destructive updates in two phases: 1) carrying out enough of each update for new operations to see the new state, while allowing pre-existing operations to proceed on the old state, then 2) completing the update after the grace period expires, so that all pre-existing operations have completed.
> citeMcKenney01a

In most cases, especially with a non-trivial data structure, RCU cannot be simply "dropped in" to make a data structure suitable for a lockless read-side. The usual difficulty is to ensure that the update-side will transform the data structure from one valid state to the next, atomically, from the point of view of the read-side. A lockless radix-tree (for use by the Linux pagecache) will now be put forward.

[1] Read-side operations, not including tag lookups, are made lockless

[2] Read-Copy Update was invented by Dr Paul McKenney and John Slingwine, McKenney was also a member of the team who implemented the RCU infrastructure in Linux 2.6. McKenney has a wide range of papers on the research and application of RCU, with a specific focus on Linux. `http://www.rdrop.com/users/paulmck/RCU/`

1.2 RCU radix-tree

In Linux, the per-inode tree_lock protects the integrity of the radix-tree pagecache data structure associated with that inode. Not only the internal integrity of the structure, but also the integrity of the results it delivers.

It is possible to remove the requirement of the tree_lock for lookup while maintaining the integrity of the radix-tree structure and the results delivered. Integrity of the data structure itself will continue to be maintained with the use of the tree_lock for the write-side operations on the radix-tree (modifications to the tree). What remains is to allow the lookup operations to run concurrently with the write-side, while giving acceptable results regardless of the possible concurrent interleaving of operations.

1.2.1 Radix-tree description

For ease of explanation, a simpler radix-tree structure will be described than exists in Linux; in particular, "tags" will be ignored. The concepts put forward here will be extended to cover the Linux radix tree in Section 1.3.

A radix-tree is a tree of nodes, where the leaf nodes are the actual data items. Each node has a fixed, power of 2 number of pointers to nodes (known as slots), and there is a pointer to the root node.

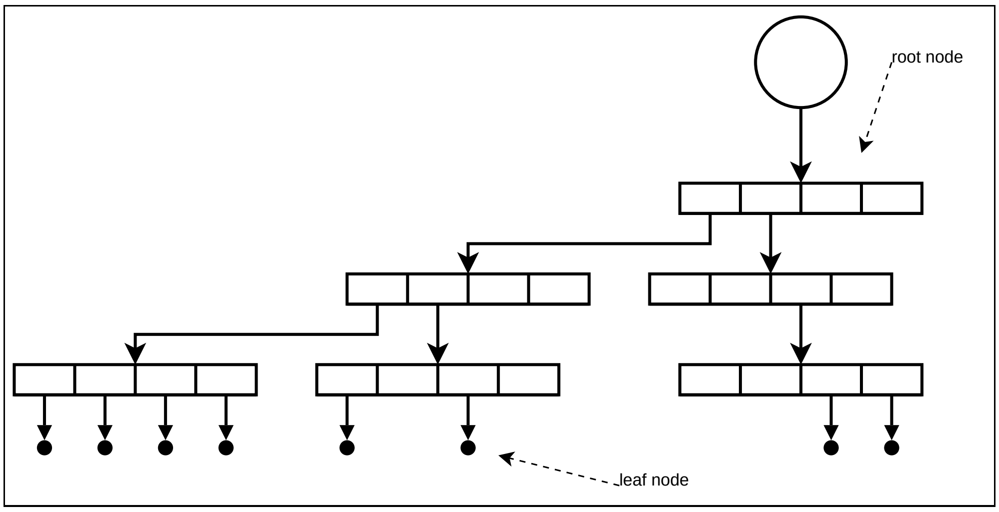

Figure 1.1: Radix-tree of height 3

1.2.2 Radix-tree height

In Linux, the height of a radix-tree is defined as the number of traversals from the root node required to reach a leaf node.

Figure 1.1 illustrates the structure of a radix tree with 4 slots per node, and a root node 3 traversals away from the leaf nodes. Slots which do not have arrows coming down from them are empty (pointer contains NULL).

The Linux radix-tree is of variable height, depending on the index keys stored. The radix-tree lookup code must know the height of the the tree it is traversing, so that it may determine whether its slots are pointers are to leaf nodes or to another layer of radix-tree nodes. A radix-tree in Linux has an associated "height" field to store this information.

The tree height is insufficient for lockless lookups, because a concurrent write-side operation may change the actual height of the tree, or the value of the "height" field at any time.

The solution to this problem relies on the observation that *changes to the tree height are only performed by inserting a new root node or removing the existing root node, leaving the height of any sub-tree the same.* Thus the height of a radix-tree node can be defined as the height of the sub-tree rooted at that node, and this height is invariant for the lifetime of the node. Figures 1.2 and 1.3 illustrate the node height invariant under modifications to tree height.

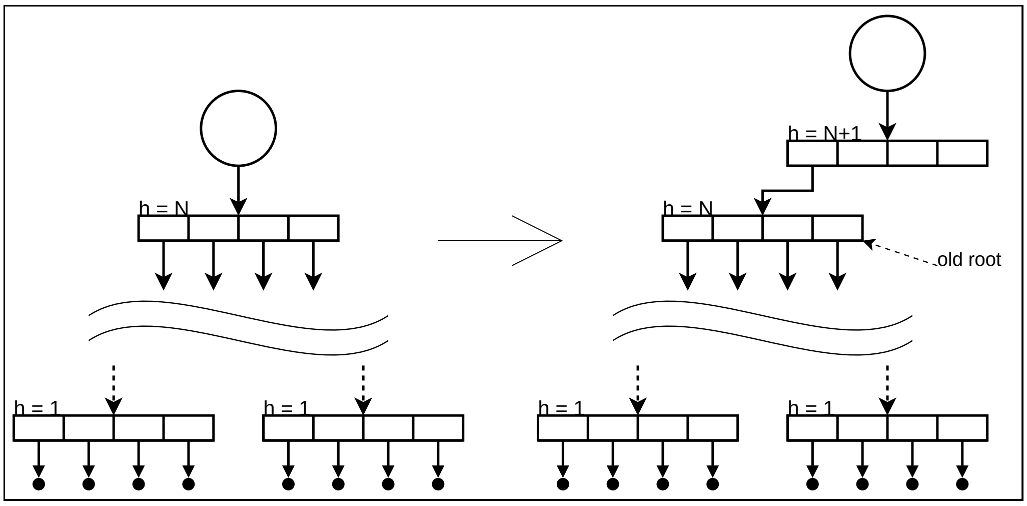

Figure 1.2: Increasing the tree height

The node height field can be set before the node is linked into the tree, and remains unchanged until the node is removed. Having a stable height per-node allows a lockless lookup to determine the number of traversals required until the leaf nodes are reached, even if the tree height is concurrently being modified.

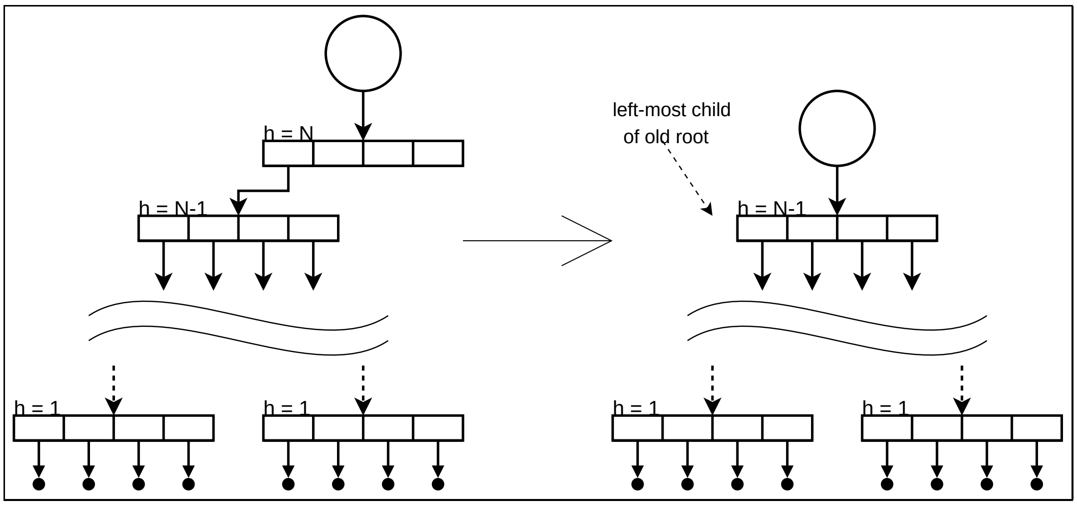

Figure 1.3: Decreasing the tree height

1.2.3 Radix-tree modification

Insertion and removal of data items are the two high level modification operations that can be performed on the radix-tree. Both operations can be broken down into a small number of basic operations that can be made atomic (with respect to a concurrent lookup) by careful ordering of operations. It can then be shown that a concurrent lookup produces the expected results regardless of the interleaving of operations:

1. Populate an empty slot with a pointer to a node or leaf-node;
2. Clear a populated slot (make it empty);
3. Increase the height of the tree by 1;
4. Decrease the height of the tree by 1.

Insertion of a new item involves a combination of operations 1 and 3, while deletion of an item involves a combination of operations 2 and 4. If all these sub-operations are shown to be safe to run in the presense of concurrent lookups, then insertions and deletions themselves may be run in the presence of concurrent lookups.

1.2.3.1 Populate empty slot, clear populated slot

Operations to populate and clear radix-tree slots are trivially atomic with respect to lookup code due to the fact that storing a value to a pointer field is atomic in Linux. A concurrent lookup will only find either an empty slot or the valid pointer.

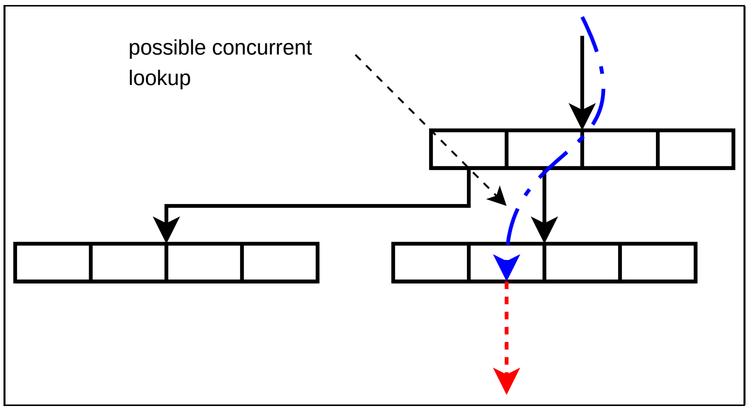

Figure 1.4: Population or clearing of a slot

Figure 1.4 illustrates a slot population or clearing operation in red, and a concurrent lookup operation in blue. Both the population and clearing operations are a special case of the general operation storing a value to a pointer field. In either case, a concurrent lookup may find *either* the old or the new value.

When inserting a node (either leaf or non-leaf), the data in the new node must be first initialised; then a memory ordering instruction must be issued; then its address can be stored in the slot. This ordering ensures that a concurrent lookup will not find uninitialised data in the new node. The correct memory ordering instructions are hidden in RCU primitives, but it is important for the reader to be aware of them.

When removing a node (clearing a populated slot), it must remain valid and allocated for as long as it is possible that a concurrent reader may still have a reference to the old pointer. This existence guarantee ensures that the concurrent reader will not find the node has been subsequently freed and used by something else. This guarantee is provided for non-leaf nodes by RCU delayed freeing. Users of the radix-tree must provide their own guarantees for the existence of leaf nodes.

There are several types of interleavings of operations to be considered for correctness. Note that non-leaf nodes are only ever inserted or removed when they are empty (they have no children).

	non-leaf node being inserted	*non-leaf node being deleted*
lookup finds node	continues at node (may now have children)	continues at empty node, lookup fails
lookup finds NULL	lookup fails	lookup fails

Leaf nodes

	leaf node being inserted	*leaf node being deleted*
lookup finds node	lookup succeeds	lookup succeeds
lookup finds NULL	lookup fails	lookup fails

Under node insertion or removal operations it is possible for a concurrent lookup to have more than one outcome depending on the exact interleaving of operations, unlike a locked lookup that only has one. This is not a fatal property of the lockless lookup, but it does result in more relaxed semantics (given in 1.2.4).

1.2.3.2 Increase tree height

When increasing the height of the tree, a new root node is added and the previous root node becomes its left-most child. The critical part of this operation is switching the root node pointer from the old to the new node. This is done after the new root node has been set up, so a concurrent lookup traversing the pointer to the root node will find either the old or the new nodes. Similarly to inserting a new node, this operation requires the new root to be initialised, then a memory barrier issued, before the root pointer is switched.

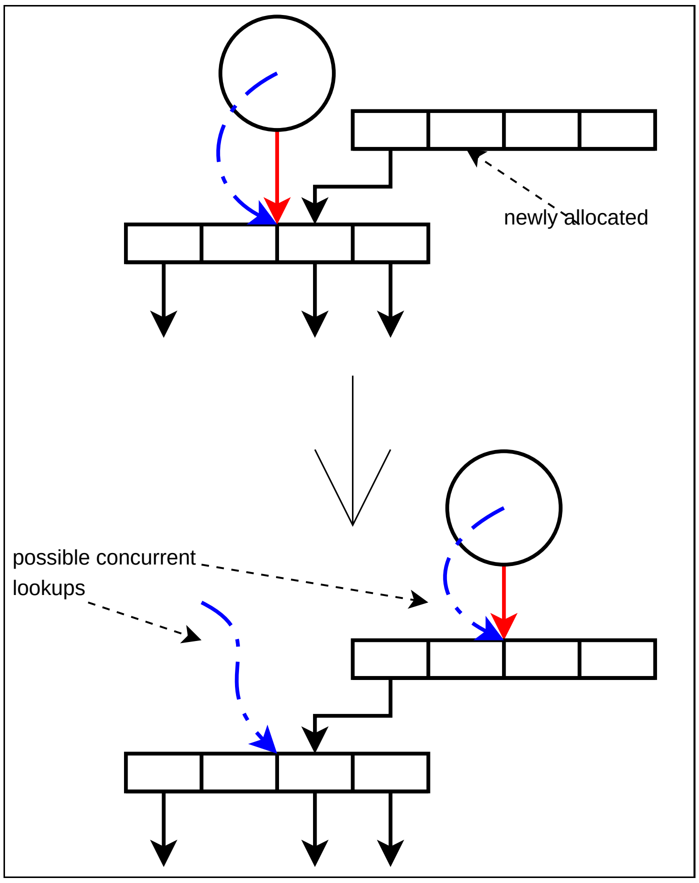

Figure 1.5: Increasing the height of the radix-tree

Figure 1.50.5 shows the process of increasing the radix-tree height. There are several combinations of lookup types and interleavings to consider; the following table illustrates that in each case, behaviour is unchanged regardless of whether the concurrent lookup finds the new or the old root.

	lookup key within old root	*lookup key not within old root*
lookup finds old root	continues at old root	old root out of range, lookup fails
lookup finds new root	leftmost slot of new root taken, continues at old root	if key out of range of new root, lookup fails; else all slots but leftmost empty, lookup fails

1.2.3.3 Decrease tree height

Decreasing the height of the tree is similar to the increasing operation, in reverse. The root node is empty except for its left-most child, which becomes the new root.

The old root is freed with the delayed RCU mechanism. Figure 1.6 illustrates why the existence guarantee provided by RCU is required: a concurrent lookup may still be operating on the old root node *after* the root pointer has switched over; if the old root were immediately freed, the concurrent lookup would be corrupted.

The interleavings of concurrent lookups are symmetric to those in the height increasing operation and can likewise shown to be unchanged regardless of whether a lookup finds the old or the new root.

	lookup key within new root	*lookup key not within new root*
lookup finds new root	continues at new root	new root out of range, lookup fails
lookup finds old root	leftmost slot of old root taken, continues at new root	if key out of range of old root, lookup fails; else all slots but leftmost empty, lookup fails

1.2.4 Lockless radix-tree lookup semantics

Radix-tree lookups which are performed under lock have the following semantics:

1. if the offset was always empty (within the locks), return NULL;
2. if the offset always contained an item, return that item;

A lockless radix-tree lookup has more relaxed semantics due to the the fact that behaviour is not deterministic in the presence of concurrent insertions or removals.

1. if the offset was always empty (according to higher level synchronisation), return NULL;
2. if the offset always contained an item, return that item;

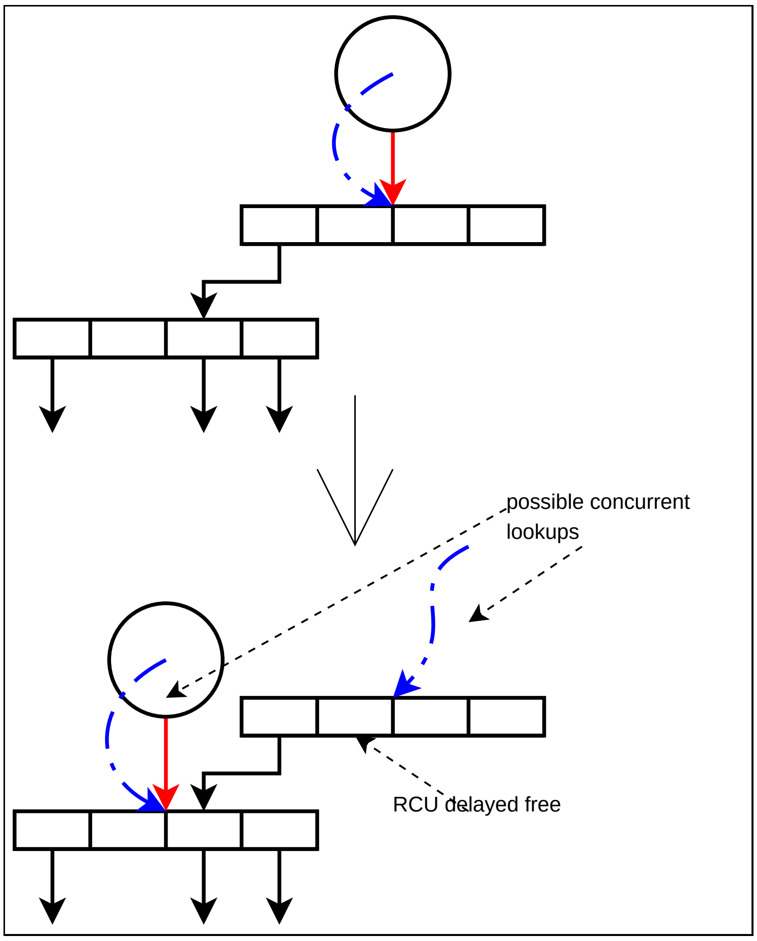

Figure 1.6: Decreasing the height of the radix-tree

3. if NULL is returned, that offset must have been empty at some time;

4. if an item is returned, it must have existed at that offset at some time.

It is shown in Chapter **??** that the Linux pagecache lookup routines can be implemented locklessly with these given radix-tree lockless lookup semantics.

1.3 Implementation details

1.3.1 Radix-tree tags

One detail glossed over in the description of the radix-tree, and design of the lockless lookup are so-called radix-tree tags. In the Linux radix-tree, each slot has a corresponding set of tags which are implemented as a bitmap. These tags are part of the radix-tree node structure, and are set and cleared under lock.

“Tagged lookups” are lookups which return radix-tree entries which have a specific tag set, or may query which tags are set for a given entry. Tagged lookups may be performed in parallel, but they require exclusion from operations which set or clear tags.

The lockless radix-tree requires that tag operations, including read-side operations are still performed under the same synchronisation: most tag operations used on the pagecache radix-tree are associated with relatively infrequent operations such as IO.

The only additional concurrency case this introduces is untagged (lockless) lookups versus tag setting and clearing operations. Untagged lookups do not affect tag operations in any way, nor do any tag operations change the structure of the tree or modify any data which is used used by untagged lookups.

1.3.2 Gang lookups

The radix-tree has facilities to perform gang lookups that return, at most, the next N items starting from a given offset. It is possible to perform gang lookups without taking any locks for the same reasons that a lookup is able to be lockless, described above.

While the locked gang lookup guarantees that all items returned are present in the tree and that they are the only items present over the given range for the duration of the lock, the lockless gang lookup may return items that no longer exist and miss items that are now present. Basically the semantics are the same as those for the lockless lookup, applied to each entry in the range of the gang lookup.

1.3.3 Child count

As well as height, child slots, and tag bits, the radix-tree non-leaf node contains a “count” field. The count field contains the number of children present in its slots [3].

The child count requires no extra consideration when moving to a lockless lookup because the it is only ever read or modified by write-side operations, which continue to maintain the same synchronisation requirements with respect to one another.

[3]Aside, the child count is used to shrink the radix tree when entries are deleted