Lock Free Red black Tree for CFS

Keerthi S. and Venkata Sai Kiran A.

December 7, 2016

Abstract

Early operating systems did not focus on schedulers with many processes. Linux prior to v2.6.23 used O(1) scheduler which was based on interactivity heuristics and run queue. The scheduler would classify the process as interactive or not based on the average sleep time to prioritize the tasks and usually non interactive processes would starve. The CFS (Completely fair scheduler) scheduler provides fairness based on the process execution time in the critical section. CFS internally uses a red black tree in order to implement a time line of future task execution. The tasks are inserted into the tree by multiple threads and the tree balances itself in order to achieve O(log n) complexity. Based on their key (priority and previous execution time), the left most node in the tree is the one to be scheduled. Our project provides a Lock free red black tree implementation of the CFS.

1 Introduction and Related work

The CFS uses a red black tree implementation which is a self balancing tree. In order to achieve high performance through parallelism, the red black tree needs to be implemented in a concurrent environment allowing operations to perform concurrently in different areas of the tree.

Most of the existing work in lock free synchronization uses CAS and TAS operations. The approach presented in the project uses single valued atomic CAS operations in order to achieve synchronization. We use the idea of *local area* present in the paper [1] and developed our own algorithm for lock free insertion and deletion. We improve upon Kim's algorithm which is very complex and requires locking nodes at seven levels of the tree above the node where either insertion or deletion is operating. [2] is an other paper which implements lock free version of RBT.

1.1 The problem

The basic idea behind lock free red black tree is to allow parallel insertion and deletions to happen concurrently if they are operating in different parts of the tree. However, an insertion or deletion operations at an area must not be affected by another parallel insertion or deletion. In order to achieve this, the inserter and deleter should mark their "local areas" using flags in order to indicate other operations that an insertion or deletion is happening nearby. Both the insertion and deletion algorithms may move up till the root of the tree and balance the tree. Another challenge of Lock free red black tree is to ensure that while moving up and balancing at every stage, the other parallel processes should not be effected.

1.2 Sequential Red black tree

The nodes in the red black tree have an additional property of colour. The red black tree balancing is achieved from the following four properties that need to be maintained.

1. The root node is black.
2. External nodes are black.
3. A red node's children are both black.
4. All paths from a node to its leaf descendants contain the same number of black nodes

The algorithm for sequential insertion and deletion can be found in [1]. Various cases exist for insertion and deletion depending on the colour of the uncle or sibling node for insertion and deletion respectively. Both the insertion and deletion algorithms may move up till the root of the tree and balance the tree.

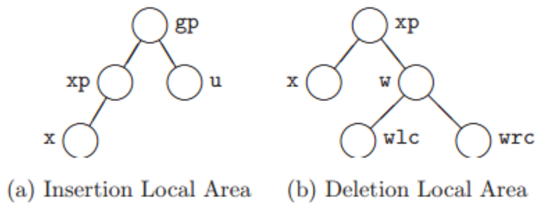

Figure 1: Local Area For Insert and Delete [1]

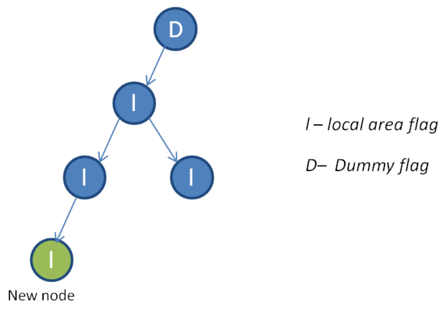

Figure 2: Local Area For Insert

2 Our Approach to Lock free Red black tree

Each node in our red black tree implementation has an additional *flag* field which is an AtomicInteger and can have three legal values *LOCAL*, *NO FLAG* or *MARKER*.

1. **LOCAL** - This flag is set if the node belongs to the "area of operation" either in the insertion or the deletion. The process of how and when to set the flag is described in the algorithm
2. **NO FLAG** - This is the default setting of the nodes. A process expects the node to be in this state before CASing it to the other flags.
3. **MARKER** - This flag is set when in order to facilitate the process to move up towards the root without effecting the other processes.
4. **MOVE UP** - This is a global flag shared across all the processes. After a process fixes the tree properties in its local area, it may have to move up. A process which is able to CAS this flag will be successful in moving up.

Each thread also maintains a Threadlocal list of nodes it has acquired in the process. At the end of its operation irrespective of whether the operation is successful, it releases all the flags present on the nodes. Otherwise, the other threads may deadlock waiting for a node to be released.

2.1 The Algorithm for Insertion

Insertion in red black tree occurs at the leaf node, so the process(a thread) traverses until it reaches the location for insertion. Then the process tries to set up its local area. Setting up the local area is equivalent to setting the flags on the nodes to LOCAL.

2.1.1 Setting up the Local Area

After the insert travels from the root of the tree to the node where it has to insert, it tries to set up the local area by CASing the flags on the following nodes (refer Fig1). x is the new node to be inserted, which is created by the thread and its flag is set to LOCAL. Before inserting the node, the thread tries to acquire the local area in the following order.

1. the Node **xp** *xp.cas(NO FLAG, LOCAL)*
2. xp's parent **gp** *gp.cas(NO FLAG, LOCAL)*

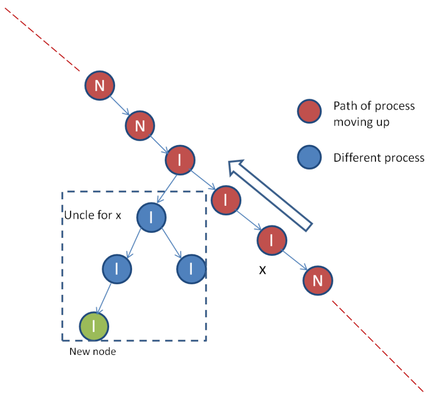

Figure 3: Use of marker Flag

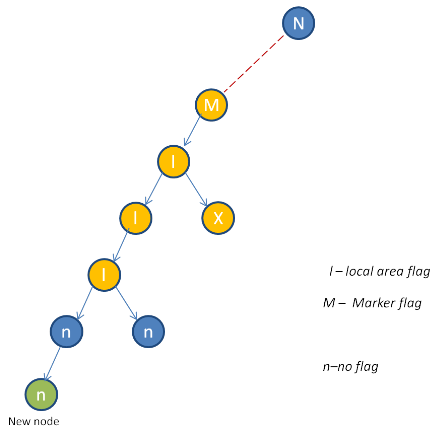

Figure 4: Process moving up

3. gp's parent **gpp** *gpp.cas(NO FLAG, MARKER)*
4. x's uncle **u** *u.cas(NO FLAG, LOCAL)*

The above order is chosen in order to avoid blocking of the other processes (we lock from downwards). Every point where the CAS succeeds, the node is added to the ThreadLocal list. At any point, if the CAS fails it releases all the flags it has acquired till now and empties the list. If it succeeds acquiring the nodes the process continues to the next step. Note that we acquire a marker flag, which ensures safety during rotation at a particular local area, explained in the next section.

2.1.2 Moving Up

Depending on the operations performed in the local area, a process moves up if it violates any of the four red black tree conditions mentioned above. The move up in the case of insertion happens from x to x' s grandparent. So we need to acquire two flags and release all the flags in the local area except gp, which will still be in the new local area. We do not need to acquire the uncles along the path to the root, because even if we rotate, the marker flag makes sure that the other process operating on the other side does not get affected. The marker node is a way of ensuring that no other process comes into the *danger area* and a safe distance is present. If we are unable to acquire the flags (CAS fails), we release the move up flag in order to facilitate the other process to go up. So the other process continues fixing up of the tree until it reaches the root. Also, the marker flags (in the current local area) become LOCAL flags after moving up.

In order to be able to acquire Local Area at the root, we have six dummy nodes above the root, which are not part of the tree.

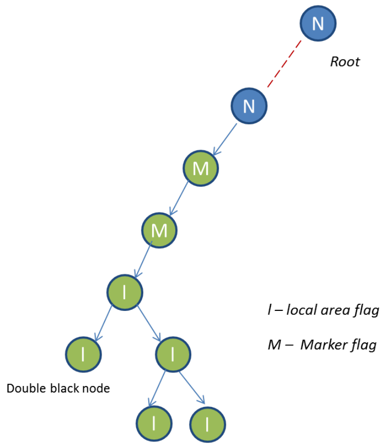

Figure 5: Deletion Local Area

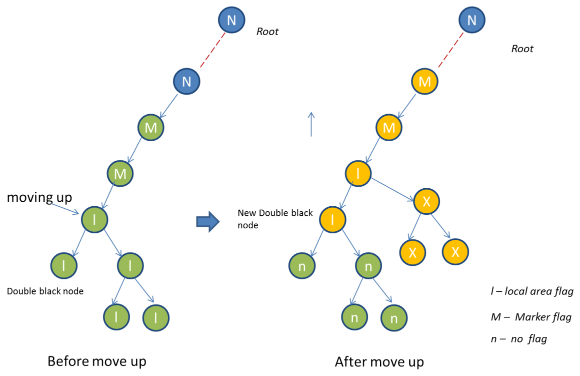

Figure 6: Figure showing the tree state before and after moveup

2.2 Algorithm for Deletion

2.2.1 Local Area for Delete

The process will traverses the tree until it finds the node it has to delete. The process acquires flag on this node so that no other process modifies this. Then, it traverses the tree to find the successor node. Once it finds, it sets up the local area at the successor node (x) as shown in the above figure. The local area includes x, xp, w, wlc and wrc. Two extra marker nodes are required to ensure safe distance between two processes. So, the local area with marker nodes is show in the figure.

The process of acquiring the local area and marker flags is same as it is mentioned in the insert section.

2.2.2 Move up

If a process is deleting a red node, then the tree's properties would not get changed. But if a process is trying to delete a black node, then the deleted node will create an imbalance in its path. This path would contain one less black node, resulting in formation of a double black node. Now the process has to fix this imbalance by series of rotations and move ups. For a process to move up the tree, it has to first acquire the next marker node in the path. If it is successful, then try to acquire the moveup flag. If it is successful, release the local area flag on the lowest node and move up is complete. If it fails in any of the steps, release the flags which it has recently acquired during move and try again. The figure shows the comparison of a process moving up by one level. It also shows the new local area and marker nodes of that process. The additional marker node that a process acquires assures that the two node below this marker node on the other side of the current process are having marker / no flags. It cannot be local area flags of other process. Hence it is safe to operate in this region without affecting the changes made by other processes.

2.3 Algorithm for Contains

Although the CFS operates by doing only insertion and deletion operations, we implemented the contains. The contains algorithm is similar to traversal performed in deletion. However while traversing down the tree, at each node the algorithm checks if the link between the node's parent and grandparent is still present. This check is required in order to prevent the rotation of grandparents node and placing the contains on a different path. If the link is broken, contains restarts traversing again. Also when it reaches the node with the correct key value, the contains checks if the node is unmarked. If it is marked, it waits. If the contains reaches the nil node it returns false and if it finds the correct node and it is unmarked, it returns true.

2.3.1 Modified Algorithm to accommodate two marker nodes for insertion

The algorithm that was presented before for insertion used one marker node. Now to accommodate the lock free delete, the algorithm for insertion has to be modified with two marker nodes. Most part of the algorithm remains same except an addition marker node is required during setting up the local area and move ups.

2.4 Use of CAS

We need to perform CAS on the flag of the nodes while setting up the local area. If we donot use the CAS, it is possible that two inserters come to the same local area and perform operations in the local area without caring for the other process. Even while moving up, CAS is needed because, when we keep on acquiring the flags without performing a CAS, we may modify another process's local area.

3 Properties of the algorithm

1. The algorithm has mutual exclusion and is deadlock free. Setting up the local area ensures that no other process can modify the local area, ensuring mutual exclusion.
2. The algorithm is lock free. At least one process will complete in finite number of steps The process which has the move up flag will complete the process eventually.
3. The algorithm is not fair. It is possible that the process might repeatedly fail the CAS operation on the move up flag because when the move up flag is released, it might be in sleep state and other process might always CAS it to true.

3.1 Linearization Points for the Algorithm

1. Insertion – when we release the local area flags at the leaf nodes. Because after the flags are released on the local area, the chnages are visible to the other threads
2. Successful delete – Replace actual node with the successor and releases flag on actual node.
3. Unsuccessful Delete – when it reaches a nil node. Even if a deleter reached the local area where the inserter is operating, lets say before deleter reached the nil node, it goes to sleep and inserter inserts the node the deleter is searching for. When the deleter wakes up and observes the node it tries to acquire the local area and it will be unsuccessful. So we can linearize it at the point where it reaches nil node while traversing.

4 CFS implementation

When the scheduler is invoked to run a new process, it follows the following procedure [3].

1. Takes the left most node (process) of the scheduling tree and is sent for execution.
2. If the process completes, it is removed from the system and the scheduling tree
3. If the process completes its maximum execution time, it is re-inserted in the scheduling tree and its execution time in the CS is updated
4. The new left most node is selected from the tree, goes to step one.

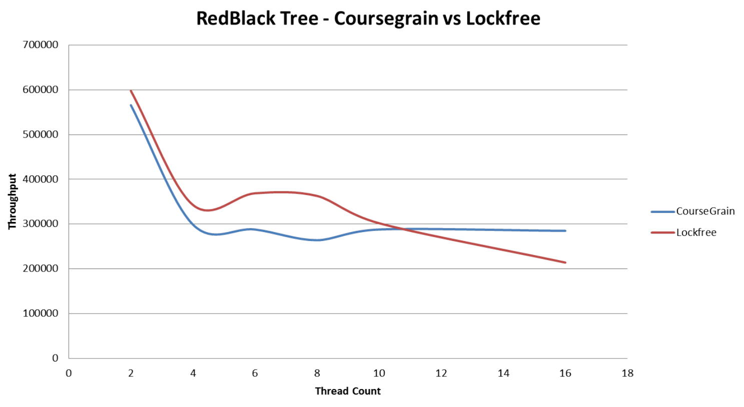

Figure 7: Throughput for concurrent insert and sequential delete

4.1 CFS implementation with priority

In our project we have extended the CFS implementation to include priority of the task. Each node contains the following additional attributes apart from the ones mentioned above.

4.1.1 Node

The node has the following additional attributes.

1. Maximum execution time
2. Priority (5 has the highest priority)
3. Current runtime (equivalent to the key)
4. process ID

4.2 Implementation

In the code, in order to simulate the CFS behaviour, we assumed that the threads represent the cores. The tasks are present in the tree which basically execute a marker method. The Scheduler does the following steps.

1. Each process gets the core only for a limited time
2. Each thread (core) comes and performs delete operation on the tree returning the left most task.
3. The thread calls *process(task)* which checks if the current runtime (equivalent to the key) reached the max execution time and if it has executed its allocated slice of time.
4. Until one of the above cases fails the scheduler allows the process to run its task.
5. After the allotted time slice is executed, and if there is still execution time left for the process, the scheduler re-inserts the task back into the tree by considering priority. Higher priority task gets a virtual lower key, so that it will be inserted near the leftmost part of the tree.

5 Results

Figure 7 shows the comparison of Lock free RedBlack tree with the course grain Red-Black tree with parallel insertions and sequential deletions. The lockfree version performs better than the course grain version for thread count ¡10. As the thread count increases, lockfree version tends to perform worse than course grain. Figure 8 shows similar with just parallel insertions. The possible reason for this could be, as the threads increases, most of the threads might have to move up to fix the violations in the tree. But only one of them would be able to move up. Also, as threads increases, the contention in the tree increases as well causing the threads to back off and try again. The above two reasons could have caused the throughput to go down for lockfree Red-Black tree. One of the future work would be to quantitatively analyze reason for this and try to work around to achieve higher throughput. In Figure 9, given all the tasks pre-populated in the tree, the total time to complete all

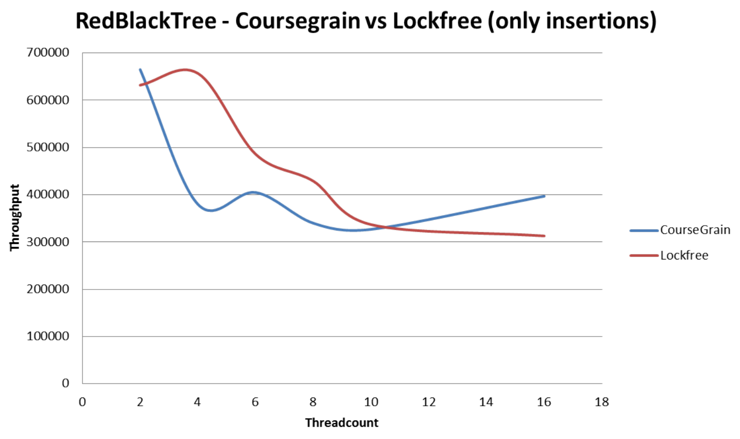

Figure 8: Throughput for concurrent insert

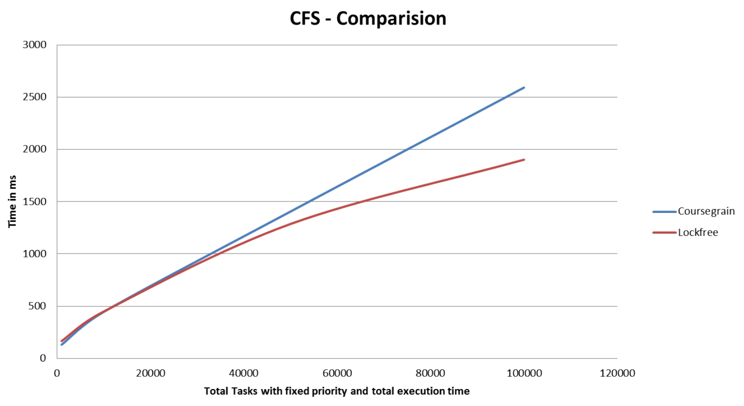

Figure 9: Comparison of CFS for lock free and coarse grained

the tasks was measured with four working threads. As the number of tasks that were pre-populated increases, CFS with lockfree red-black tree performs better than the course grain red-black tree. The CFS thread count is four, number of contentions in the tree would be less and hence as the number of tasks increases, the parallelism that can be achieved goes up. Figure 10 shows the end of all the tasks with same maximum execution time, and same priority. For the scheduler to be fair, all the tasks should complete their execution around the same time. The graph shows that the scheduler is fair with all tasks being completed at almost same time. Figure 11 shows the task execution time vs priority of the task. Execution time of a task is time divided by a factor of its priority. Hence the probability of it being executed again is high. The results show that the tasks with higher priority (5) are completed earlier than the tasks with low priority (1).

6 Future Work

For lock free red-black tree, quantitative analysis has to be done to find reason for lower throughput for higher thread count. Concurrent deletion is partially implemented in this project. This can be taken further to completely implement concurrent deletion.

References

[1] Jong Ho Kim, Helen Cameron, and Peter Graham. https://www.cs.umanitoba.ca/ hacamero/research/rbtreeskim.pdf.

[2] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. *Concurrent Wait-Free Red Black Trees*, pages 45–60. Springer International Publishing, Cham, 2013.

[3] CFS scheduler. https://en.wikipedia.org/wiki/completely$_{f}air_{s}cheduler$.

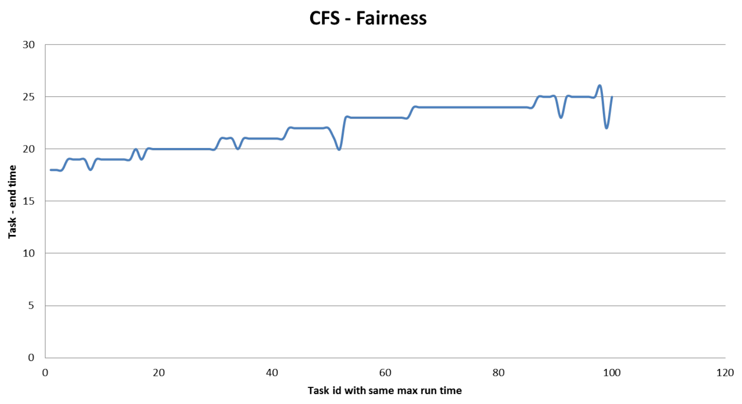

Figure 10: Fairness in CFS

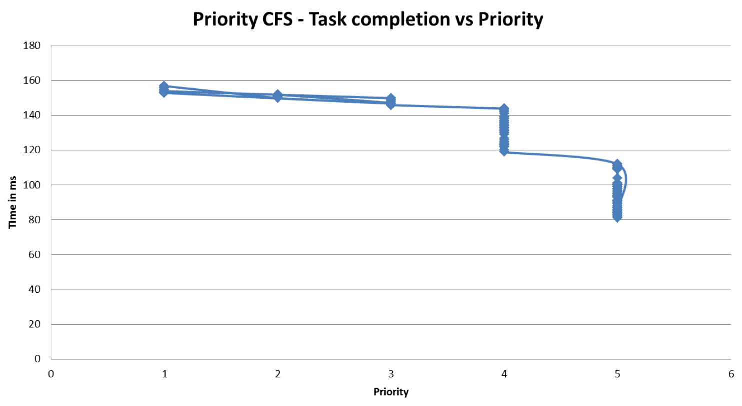

Figure 11: Priority vs completion time