ECE/CS 5510 Multiprocessor Programming
Project Report - 5. Lock-free red-black tree based CFS scheduler
Mincan Cao micao@vt.edu, Ho-Ren Chuang horenc@vt.edu

1. Problem Definition

This project is trying to design and implement lock-free tree based CFS. As we all know lock-based synchronization mechanism incurs unexpected scheduling anomalies such as priority inversion and convoying and more vulnerable to deadlock. Although lock-free techniques are able to address the problems, the implementation of a lock-free algorithm for complex data structures is for sure completed.

Furthermore, we propose a single thread simulator to show the CFS property in an ideal environment without any interference. We also borrow the idea of fairness in Linux since a "fair" scheduler can be fair to a task but not every task. So we try to understand the definition of fairness in Linux. And then implement them as much as we can in our Linux-like CFS simulator. Moreover, for data structure, we, based on the simulator, try to understand why Linux utilizes RB-tree rather than AVL-tree, which is very similar to RB-tree, simpler, and faster in most of cases, as its queues.

2. Introduction

Lock-free red-black tree (RB-tree):

A red-black tree is an advanced tree based on binary search tree. The binary search tree is not always balanced because of the order of insertion. Then the red-black tree introduces one extra attribute for each tree node: the color, which helps the self-balance of the red-black tree. Thus, for a red-black tree, it guarantees the time complexity of insertion, search, and deletion is O (log n).

The typical red-black tree has to maintenance the following red-black properties [1]:

- Every node is either red or black
- Every leaf (NULL) is black
- If a node is red, then both its children are black
- Every simple path from a node to a descendant leaf contains the same number of black nodes

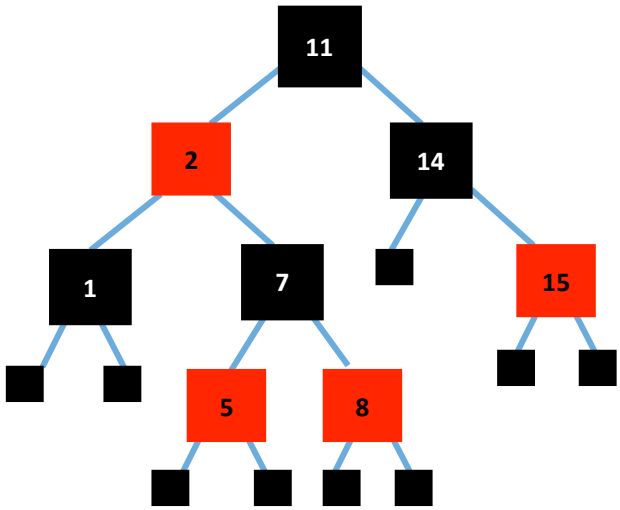

Figure 1. One of typical red-black trees

The core part of a red-black tree is the self-balanced method. Thus, here are two basic operations named left rotation and right rotation. In the fig2, the inorder of the nodes is A -> x -> B -> y -> C.

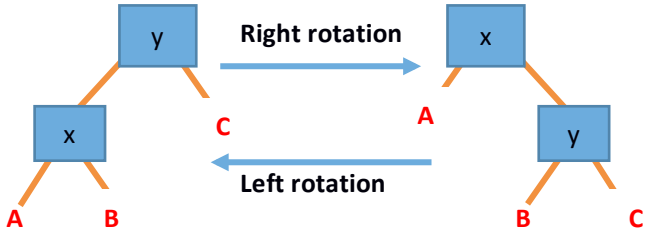

Figure 2. Rotation examples

Therefore, the execution of insertion is divided into two parts: 1) Common binary search tree insertion 2) Bottom-up rotation fix up. The first assumption is every inserted node is red. Hence, the first case for red-black tree insertion is that the node is inserted in the right place and tree is still balanced. Like fig 1, if we insert 13, then the node will become the left children of node 14 and there is no need to do fix up. Another situation is that the parent of inserted node is red. Because the children of red node must be black, then the bottom up fix up must be executed in order to keep the tree balanced. This project will not discuss the details of how the red-black tree executes fix-up.
However, one thing could be concluded by fix-up algorithm is that every time the fix-up routine will obtain the ownership of four nodes: itself, its parent, its grandparent, and its uncle. And one hypothesis we proposed is once the rotation of one node finished, at that point the node's subtree will keep the red-black properties.

RB-tree based Complete fairness scheduler (CFS):
The main idea of CFS is very simple - give equal CPU resource proportion to every task. However, this is probably not the fairest way to users and to kernel since some tasks are more important, some are not. Thereby, the idea of different time slices appears. Time slice means how long can a task use CPU resource. By assigning proportions of the processor and not fixed time slices, Linux CFS is able to enforce fairness: each process gets its fair share of the processor [3]. If a time slice is too long, response time gets longer. User experience might decrease. If a time slice is too short, context switch between processes and kernel overheads might take effects. So a time slice must be not too long and not too short. Minimum granularity is used by Linux to minimize the system context overhead.

In order to give different time slices, we must give different tasks different weights (priorities). This is so called a priority for a task. In Linux we call it "nice value". Processes with the default nice value of zero have a weight of one, so their proportion is unchanged. Time slice equals to actual runtime. Processes with a smaller nice value (higher priority) receive a larger weight, increasing their fraction of the processor. Process with a larger nice value (lower priority) receives a smaller weight, decreasing their fraction of the processor [3]. Legal nice values range from −20 to 19 inclusive. The higher nice value it has, the lower priority it owns. Nice value decreased by 1 means 10% more CPU time the task can utilize. Linux has a RLIMIT_NICE parameter for limiting dynamic nice value range.

All together, Linux records the time each task runs in a "virtual run time" (vruntime), which almost equals to all its time slices accumulation. It means how long does a task occupying CPU resource. The main idea of CFS is to always pick up the minimum virtual run time task to run. The minimum value is always the leftmost node in a RB-tree/AVL-tree. In addition, a new coming task's vruntime is not always 0. Otherwise, the task will always be selected to enjoy CPU resource. Lunux maintain the least vruntime in each runqueue as a reference to solve this problem.

CFS vruntime formula:
/* delta: how long process really runs, the time a process has the CPU resource to it release the resource */
/* weight: based on 10% rule, roughly equivalent to 1024 * (1.25)^(-nice)*/
vruntime += delta * (nice / weight);

To summarize, when the actual execution time is the same, the higher nice value a task has, the

faster its vruntime increases. As a result, this task's actual runtime (the time occupying a CPU) is less than a task having a lower nice value.

After having a CFS scheduler, we based on this try to check how these two different concurrent data structures (RB-tree, AVL-tree) used by CFS scheduler affect the system execution time. In order to make traffic on accessing the runqueue, in multi-threaded simulator, all processor share a same global queue. Whereas Linux maintain local queue on each process and then it comes up a bigger topic "load balance", which is not very relevant the relationship between CFS and RB-tree. We more focus on the relationship between RB-tree and CFS and CFS essential properties. Thus, we don't implement that in our simulator

Other less important Linux mechanisms we haven't support yet:

- Fork()
- Real-time guarantee
- External interrupt
- Per-CPU runqueue
- Group scheduling - instead of scheduling by tasks, schedule by processes. e.g. huge process(100 tasks) VS small process (1task)

* As a side, O(1) uses a heuristic algorithm (complex calculation) to solve interactive or non-interactive task problems, which performance is not well. CFS is introduced to solve older O(1) scheduler's performance problem.

3. Implementation

3.1. Lock-free Red-Black Tree

In order to realize the concurrent red-black tree, one of the simplest solution is to use Coarse-Grained Synchronization method. Every time one thread wants to modify the tree, it will lock the whole tree. However, this project is target at the lock-free red-black tree. Kim in 2006 published a paper discuss the implementation of Lock-Free Red-Black Trees Using CAS [2]. He proposed his method based on Ma's algorithm [3]. Kim both used flag and marked to keep tracking the usage of nodes of each threads. However, Natarajan in 2013 published a paper argued that Kim's method is not completely lock free [4].

In this project, we find out the realization of Lock-free Red-Black tree is extremely hard. The most difficult part of insertion in a red-black tree is the bottom-up fix-up. The basic operations of fix-up is rotation, which will physically change the positions of three nodes. And it potentially has 2 or 3 rotations for one node is inserted into a red-black tree. Concurrently, we want to other threads don't affect the unfinished fix-up when they insert their own nodes into the tree. This part is hard to analysis. According to Kim's method, each thread will obtain the ownerships of fours nodes before it starts insertion. We take the advantages of this idea and treat the insertion operation into two parts: logically add and physical add.

Insertion

The lock-free insertion algorithm will first to find a place to insert. Then the line 45 will try to setup the local area for Insert. It will try to obtain the ownerships of four nodes we mentioned before by CAS operation. Once the thread obtain the ownerships, it won't worry about other threads insert other node at below area. Then it could physically insert the node and call fix-up function. The fix-up function will still obtain ownerships from bottom to up and release the flag at the end.

```
30    public void add(V value) throws NullPointerException{
31        LockFreeRBNode<V> y,z;
32        LockFreeRBNode<V> x = new LockFreeRBNode<V>(value);
33        x.flag.set(true);
34        while(true){
35            z = null;
36            y = this.root;
37            while(y!= null && y.value != null){
38                z = y;
39                if(value.compareTo(y.value)<0){
40                    y = y.left;
41                }else{
42                    y =y.right;
43                }
44            }// end while
45            if(!SetupLocalAreaForInsert(z)){
46                z.flag.set(false); // release help flag
47                continue;
48            }else{
49                break;
50            }
51        }
52        // place new node x as child of z
53        x.parent = z;
54        if(z == null) { this.root = x; }
55        else if(value.compareTo(z.value)<0){
56            z.left = x;
57        }else{
58            z.right = x;
59        }
60        x.left = new LockFreeRBNode<V>();
61        x.right = new LockFreeRBNode<V>();
62        x.isRed = true;
63        insertHelp(x);
64    }
```

Table 1. Lock-free RB-tree add method code

Search

The search method will just traversal the tree because it will not modify the tree. Since this project hasn't realize the lock-free deletion method, search method will treat all nodes inside the tree. It will not meet the deletion condition. Thus, the implementation of search is quite simple.

3.2. RB-tree & AVL-tree based Linux-like simulator

There are two ways to differentiate tasks. First, one can split tasks into non-interactive and interactive. Linux adopts this way since it's a general purpose OS, which cares more about user experience. However, here we want to use the second way, which distinguishes tasks into I/O bound and CPU bound. Although our definition of fairness is slightly different from Linux, we are still able to borrow common CFS optimizations from Linux to our CFS.

- CPU bound：processes are hungry for CPU time
- I/O bound：spend more time blocked waiting for some resource than executing，often issuing and waiting for file or network I/O, blocking on keyboard input [3]

To make an ideal environment for checking CFS property, our simulator doesn't consider the real system overhead. This makes us to get ride of interferences from a system. We concentratedly focus on implement Linux version CFS and use it to compare with different data structures as its run_queue.

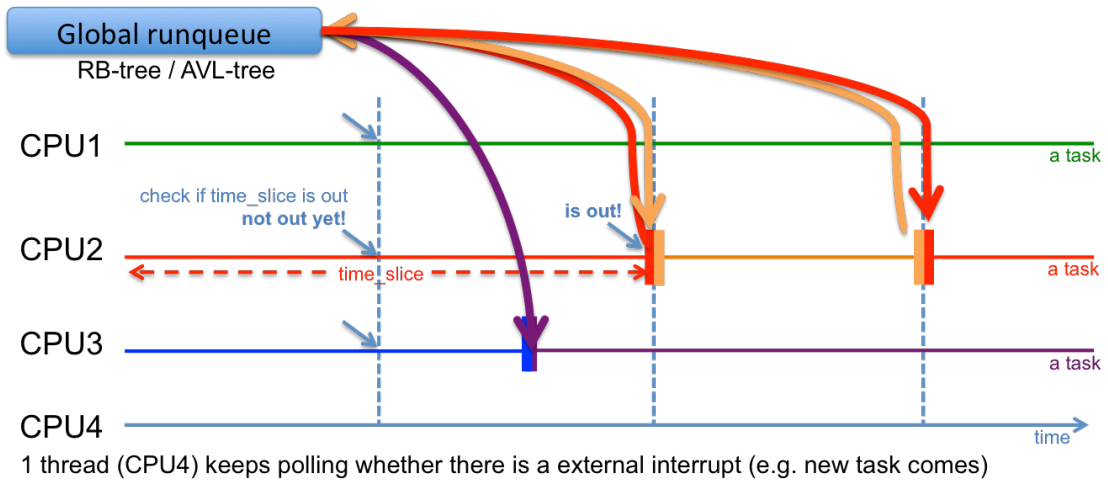

Figure 3. Linux-like CFS scheduler simulator

Linux-like CFS simulator features:

CFS features -

- Minimum granularity for a time slice.
- CPU resource limitation: RLIMIT_NICE
  e.g. if RLIMIT_NICE = 25, nice can be only 20-25＝-5
- New task's vrtime is not always 0. Linux keep a least number for a runqueue.

Scheduler features -

- Timer interrupt: check if delta_exec > time_slice, enq() and then deq()
- thread_exit(): when a thread finishes its jobs
- thread_create() : start_time

4. Experimental results

4.1 Lock-free RB-tree

Test environment: 40 cores

Lock-Free RB-tree add operation experiment:

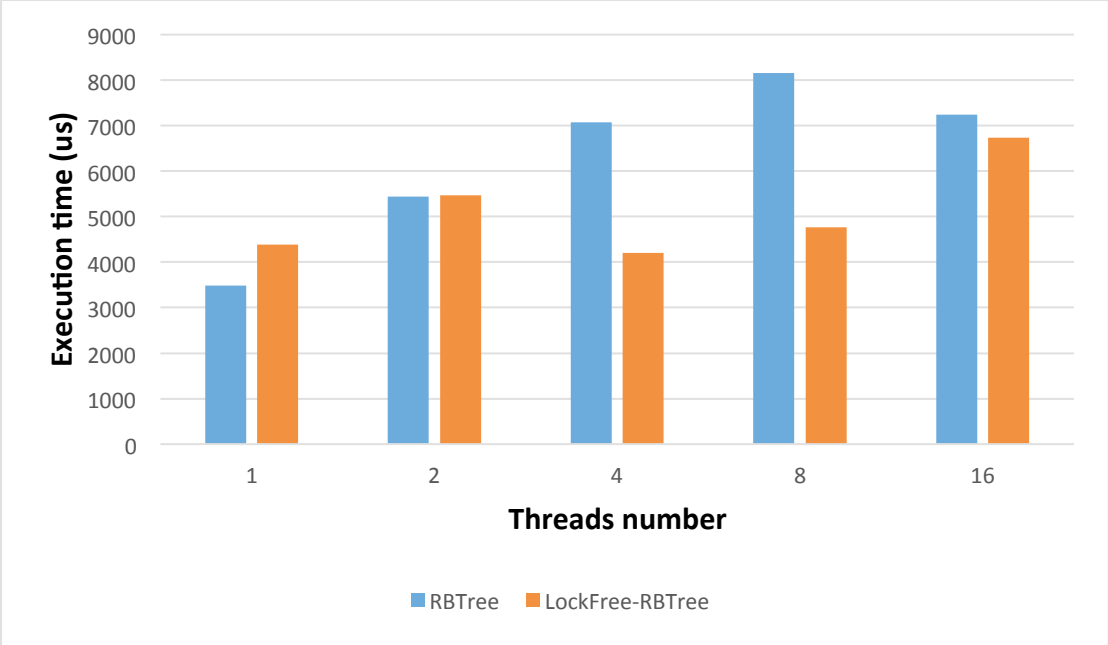

Figure 4. The cost of creating 320 tree nodes in terms of time (unit: us)

We did two experiments. We let different threads equally create the same number of nodes tree, then measure the time spent by lock based Red-Black tree and our Lock-free Red-Black tree. The experiment result depicted as Fig. 4 indicates our Lock-free Red-Black Tree and

lock-based tree have a similar execution time at lower threads condition. More importantly lock-free implementation compared to lock-based compared has less running time at higher threads condition.

Lock-Free RB-tree search operation experiment:

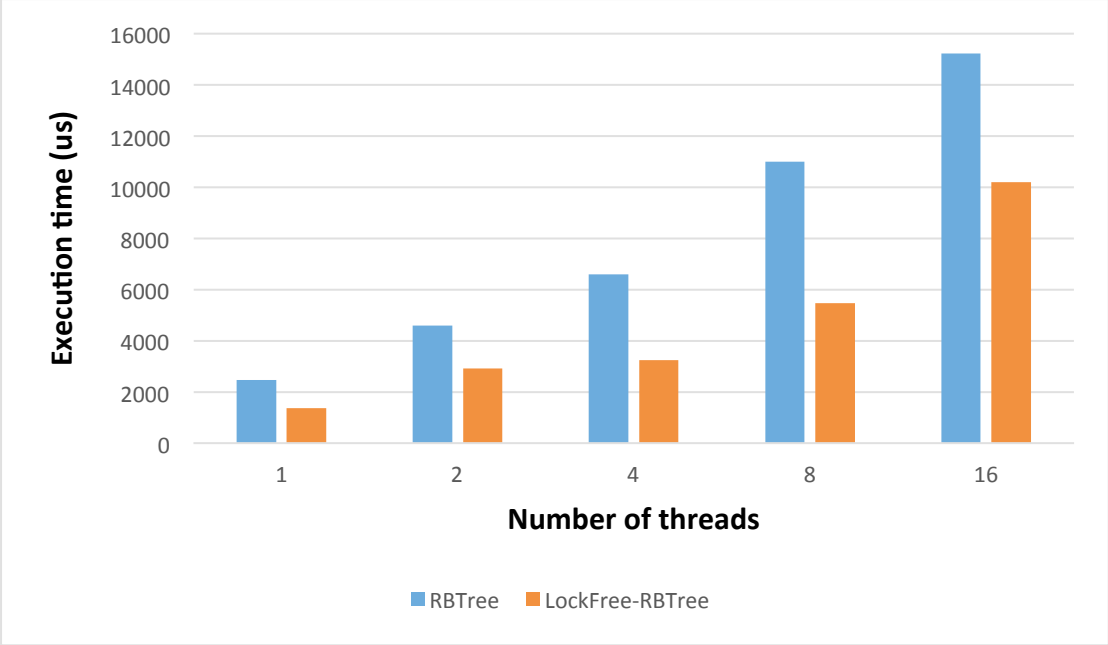

Figure 5. The cost of searching 1000 times on a 320 nodes tree in terms of time (unit: us)

The second experiment is to compare the search performance. The lock-based search operation will linearly increase due to each threads will execute 1000 times. According to experiment result as shown Fig. 5, the lock-free version has shorter execution time, which proves each thread successfully runs search concurrently.

4.2. Linux-like CFS Simulator

Test environment: 4 cores (hyper-threading)

Single-threaded simulator - for checking CFS property

Input task:

in_single.txt

Name	#ofT	CPU(us)	IO(us)	PRI	Nice	Start_time(us)	Note-> (Attention: no blank line)
T1	5	10000	1000	100	1	0	1~5
T2	3	5000	0	100	1	1000	6~8
T3	3	5000	0	110	1	1000	9~11
T4	3	5000	0	120	1	100	12~14
T5	5	1000	10000	100	1	0	15~19

Single thread simulator result:

```
# of TASK=19
g_queue_thread_num=0
g_exec_thread_num=0
g_done_thread_num=19
g_time=15 s (system virtual ticks)
6 7 8 9 10 11 12 13 14 1 2 3 4 5 15 16 17 18 19
```

Since this is a single task program. As long as the in.txt the same, the output order will be also the same. In this experiment, we want to demonstrate out CFS implementation works correctly. As we can see in the figure, even if T2 started later than T4, since its priority is higher than T4, it finished before T4.

Multi-threaded simulator - for checking the difference between using RB-tree and AVL-tree

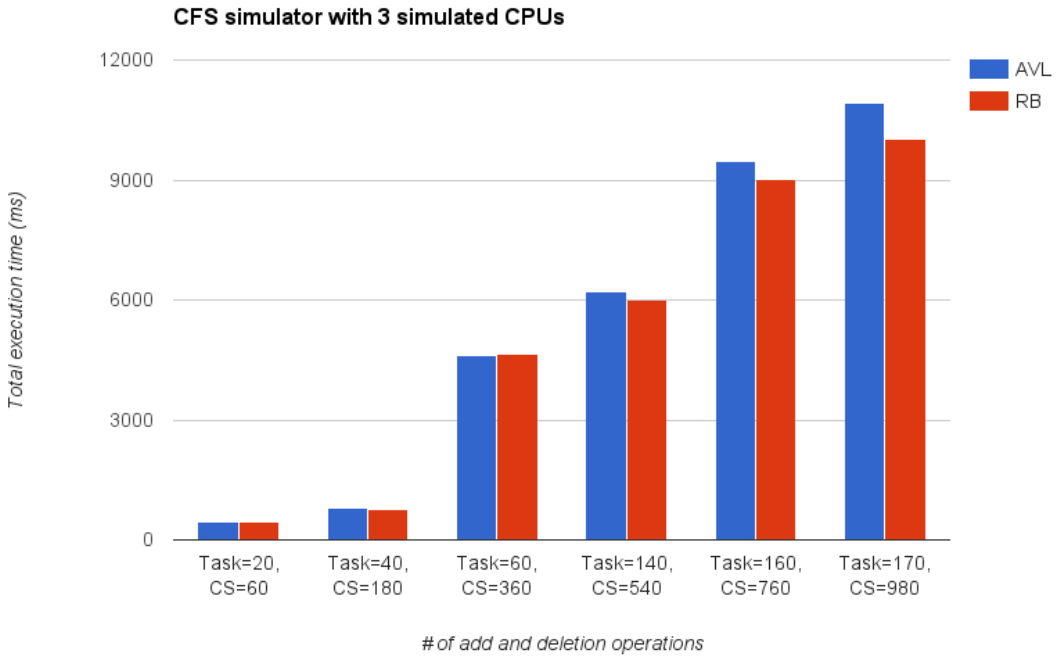

Figure 6. CFS simulator with 3 (fixed) simulated CPUS

The result shows that as increase number of Critical Sections (1 CS = 1 add + 1 del operations), RB-tree starts to take advantage of its RB property. I think this is since AVL tree doesn't use color or marker to make additional information, its rotation operation can be very expensive when the tree is large (complex). Whereas RB-tree can enjoy the benefit of keeping maintaining tree's color when the tree is large (complex). The RB-tree's worst case execution time for an insert operation is 2 times rotation and for a deletion operation, 3 times rotation. This merit can help RB-tree outperform AVL-tree while the tree is large (complex).

Additionally, during the low workloads, it's hard to tell which data structure perform better. More importantly, hardly do users feel the differences. We think the reason why Linux chooses RB-tree as queue data structure is that both data structure don't have much difference during low workload scenario, but RB-tree can significantly outperform AVL-tree while high workload scenario.

5. Conclusion

Through this project we not only try to implement a lock-free data structure by utilizing what we have learn so far but also we have a better understanding about how Linux handle CFS and start to think why Linux choose RB-tree as its implementation. Another big lesson that we have also learned is that before enjoying the performance gain, how hard it is to implement a lock-free data structure. During our debugging process, we suffer from a lot of deadlock and starvation problems, which are really hard to find the root cause and to debug.

Our main contribution:

- Implementing/modifying AVL-tree & RB-tree
- Implementing parts of lock-free RB-tree methods (insertion - partially working, search - fully working)
- Implementing Linux-like CFS simulators (single-threaded and ideal version, multi-threaded and concurrent version)

Reference

[1] Molnar, Ingo. "Modular scheduler core and completely fair scheduler [cfs]."Linux-Kernel mailing list (2007).
[2] Kim, Jong Ho, Helen Cameron, and Peter Graham. "Lock-free red-black trees using cas." Concurrency and Computation: Practice and Experience(2006): 1-40.
[3] Love, Robert. Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.", 2013.
[4] Jianwen Ma. Lock-Free Insertions on Red-Black Trees. MSc thesis, University of Manitoba, October,2003.
[5] Natarajan, Aravind, Lee H. Savoie, and Neeraj Mittal. "Concurrent wait-free red black trees." Symposium on Self-Stabilizing Systems. Springer International Publishing, 2013.

[6] The university of Auckland, Computer Science, Data Structures and Algorithms Course
https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

Appendix

1. CFS simulator

How to run CFS simulator

CFS simulator single-threaded version:

CFS_simulator_single_thread.java

CFS simulator multi-threaded version:

CFS_simulator_multi_thread.java

Choose one run_queue data structure

```
static boolean IS_RBTREE = false; // run with AVLTree
static boolean IS_RBTREE = true;  // run with RBTree
```

Variables in code

```
static int THREADS = 3;           // # of workers (simulated CPUs, not task/jobs)
static int TimerIntThreshold = 1000*1000;  // timer interrupt ticks = 1ms
static int min_granunarity = 1000*1000;    // minimum granularity = 1ms
static int dynaic_nice_rang = 5;      //nice(dynamic)=original_nice+-dynaic_nice_rang
```

Variables in input file

Assign Tasks (Jobs) for single-thread simulator:

$ vi in_single.txt

Assign Tasks (Jobs) for multi-threaded simulator:

$ vi in.txt