ECE/CS 5510 Multiprocessor Programming

Project Proposal - 5. Lock-free red-black tree based CFS scheduler

Mincan Cao [micao@vt.edu](mailto:micao@vt.edu), Ho-Ren Chuang [horenc@vt.edu](mailto:horenc@vt.edu)

**1. Problem**

Completely Fair Scheduler (CFS)

For scheduler,

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 CFS simulator.

By fair

For data structure, we also 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.

There are two ways to differentiate tasks. We can split tasks into non-interactive and interactive, batch. 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密集型：processes are hungry for CPU time，比如科学计算、数学计算、图像处理

I/O密集型：spend more time blocked waiting for some resource than executing，often issuing and waiting for file or network I/O, blocking on keyboard input

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

**2. Introduction**

The main idea of CFS is very simple - give equal CPU resource to every task. However, this is probably not the best way to users and to kernel since some tasks are more important, some are not.

differet time slice

2. time slice - how long can a task use CPU resource

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.

如果时间片过大，那么 挂起进程 开始执行前的等待时间过长，这将减小 并发执行的粒度，甚至用户觉察到延迟

如果时间片过小，那么 系统在进程之间切换的时间花销将很大，时间局部性的优势将丧失

**Minimum granularity** is used by Linux to minimize the system context overhead. 是任一进程运行时间长度的下限，这将保证 上下文切换开销占 系统总时间开销的比例 不会过大

**By assigning proportions of the processor and not fixed timeslices, Linux CFS is able to enforce fairness: each process gets its fair share of the processor**

4. priority

Processes with the default nice value of zero have a weight of one, so their proportion is unchanged

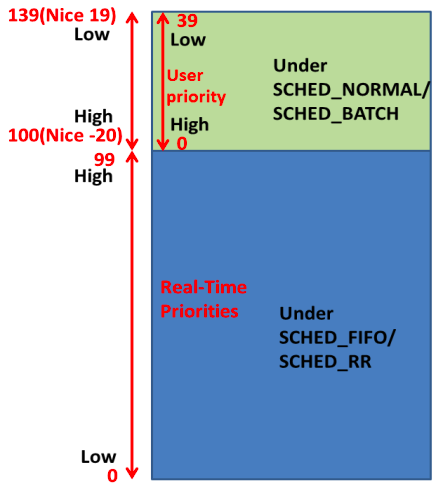
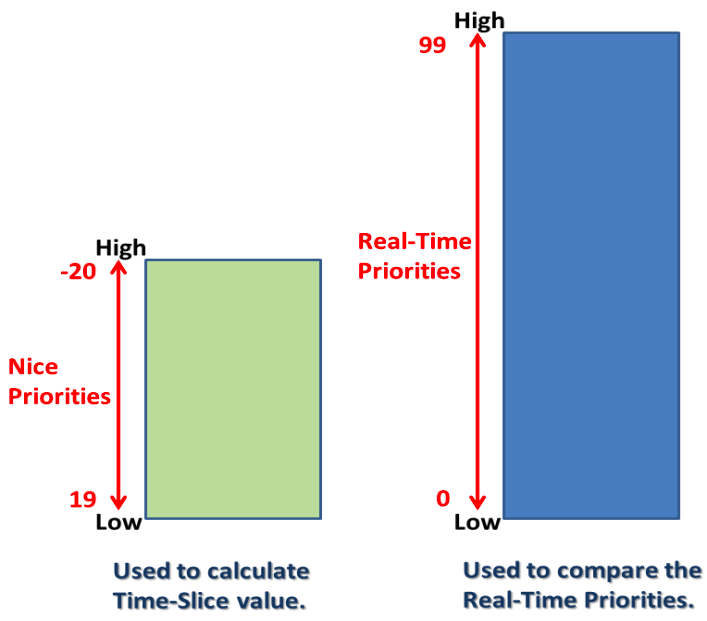
Processes with a smaller nice value (higher priority) receive a larger weight, increasing their fraction of the processor

process with a larger nice value (lower priority) receive a smaller weight, decreasing their fraction of the processor

4. nice val

**processes are assigned priorities that affect how long they run，Unix has historically called these priorities nice values.**

Legal nice values range from −20 to 19 inclusive, with a default value of 0， nice值越大，优先级越低，nice值越小，优先级越高



进程每降低一个nice值，将多获得10%的CPU时间。

in linux nice value = 10%

weight = 1024 / (1.25 ^ nice\_value)

1.25 <= **将多获得**10%**的**CPU**时间**

**Features: Linux提供了RLIMIT\_NICE资源限制。如果一个进程的RLIMIT\_NICE限制为25，那么其nice值可以提升到20-25＝-5**

5. all together

virtual runtime

virtual time

virtual time += actual run time \* weight (become faster or slower)

nice = > weight =>

Virtual RunTime會根據Task的Priority而有增長的差異,例如:Nice Priority低的Task Virutal RunTime會增長的比較快,相對佔據處理器執行的時間就會比較短,反之,Nice Priority 高的Task會因為Virutal RunTime增長的慢,而得到相對比較多實際的處理器執行時間.

1. CFS virtual\_run\_time formula:

**vruntime += delta \* (1024/se.load.weight);**

**curr\_task.time\_slice = (int) ((1\*1000) \* (float)(curr\_task.nice / (1024 / Math.pow(1.25, curr\_task.nice))));**

/\*delta：how long process really runs, the time a process has the CPU resource to it release the resource \*/

Summary：When the execution time is the same , the higher weight, the slower vruntime increased.

在实际运行时间相同的情况下，调度实体权重越大，vruntime增加的越慢。

2. ideal time\_slice formula:

ideal\_time = slice \*(se.load.weight/cfs\_rq.load.weight);

/\*slice为CFS运行队列中所有进程运行一遍所需要的时间\*/

/\*slice的经验计算公式如下：\*/

if(cfs\_rq->nr\_running > 5) 大於五個人就算

　　slice = 4 \* cfs\_rq->nr\_running;

else 小於五個人直接給20ms

slice = 20; /\*单位ms\*/

基於CFS RBTree的概念,由於會一直確保Virtual RunTime值的平衡,透過抓取佔據Virtual RunTime執行時間最短的Task來執行,因此像是原本排程中會透過Nice值計算對應Task可運作Time Slice的設計方式,就變的需要有所調整,也因此在CFS中會透過函式**calc\_delta\_mine** (實作在 kernel/sched.c),**根據目前Task的優先級計算在排程時每次一個Scheduling Tick要增加的Delta值**,也就是說如果該Task Nice值越高(優先級越低)該Delta值就越大,對該執行的Task來說就是Virtual RunTime在運作時,增加的速度越快(等於縮短實際執行的時間),反之,如果該Task Nice值越低(優先級越高)則Delta值就越小,在每次Scheduling Tick觸發時,Virtual RunTime增加的速度越慢,能獲得處理器執行的實際時間也就越長.

問題剛加進來的vtime會很低

Linux解決方法: 挑選目前runqueue最小的給他

virtual\_runtime is not always 0

提這個

In order to make traffic on accesing the queue, 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 CFSand RB-tree. Therefore, we don’t implement in our simulator.

Since there we focus on the concurrent data structure. We don’t implement it . a simulator

Migrating a process take different time on differet machine. we more focus on the relationship between RB-tree and CFS and CFS essential properties.

6 others we don’t have

Group scheduling- instead of scheduling by tasks, schedule by processes

e.g. huge process(100 tasks) vs small process (1tasks)

so別提load balance

7. 处理器关联

the process scheduler must decide which processes run on each CPU

如果一个进程在一个CPU核上被调度，the process scheduler should aim to schedule it on the same CPU in the future

因为 进程从一个CPU核迁移到另一个CPU核的代价是巨大的（主要是缓存影响）

if a process moves to a new CPU and writes new data into memory, the data in the old CPU's cache can become stale

这样，进程调度器必须在 进程迁移CPU花销 和 多个CPU负载均衡 之间取得平衡

The Linux scheduler attempts to schedule the same processes on the same processors for as long as possible, migrating a process from one CPU

to another only in situations of extreme load imbalance. This allows the scheduler to minimize the cache effects of migration but still ensure that

all processors in a system are evenly loaded

**3. Experiments**

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

time for invoking a scheduler

圖

**(2) 当前调度实体的时机运行时间大于理想运行时间（delta\_exec > ideal\_runtime）,这一步在时钟中断 处理函数中完成；**

3. CFS调度时机

在有了上面几个计算公式之后，就可以总结出CFS调度算法的几个调度时机：

(1) 调度实体的状态转换的时刻：进程终止、进程睡眠等，广义上还包括进程的创建(fork)；

**(2) 当前调度实体的时机运行时间大于理想运行时间（delta\_exec > ideal\_runtime）,这一步在时钟中断 处理函数中完成；**

(3) 调度实体主动放弃CPU，直接调度schedule函数，放弃CPU

(4) 调度实体从中断、异常及系统调用返回到用户态时，回去检查是否需要调度；

**内核会周期性地检查进程是不是已经耗完了自己的时间片(Jack 表示在timmer interrupt才檢查 我是對的)**

**io\_runtime + cpu\_runtime >** time\_slice => preemption (enq+ deq)

**Simulator Linux-like CFS features:**

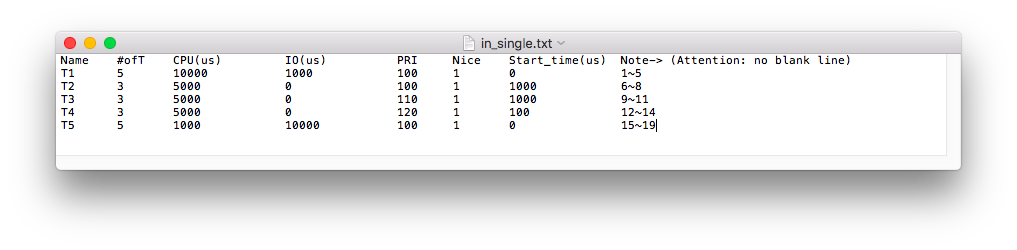
(all from Linux)

**4. Experimental results**

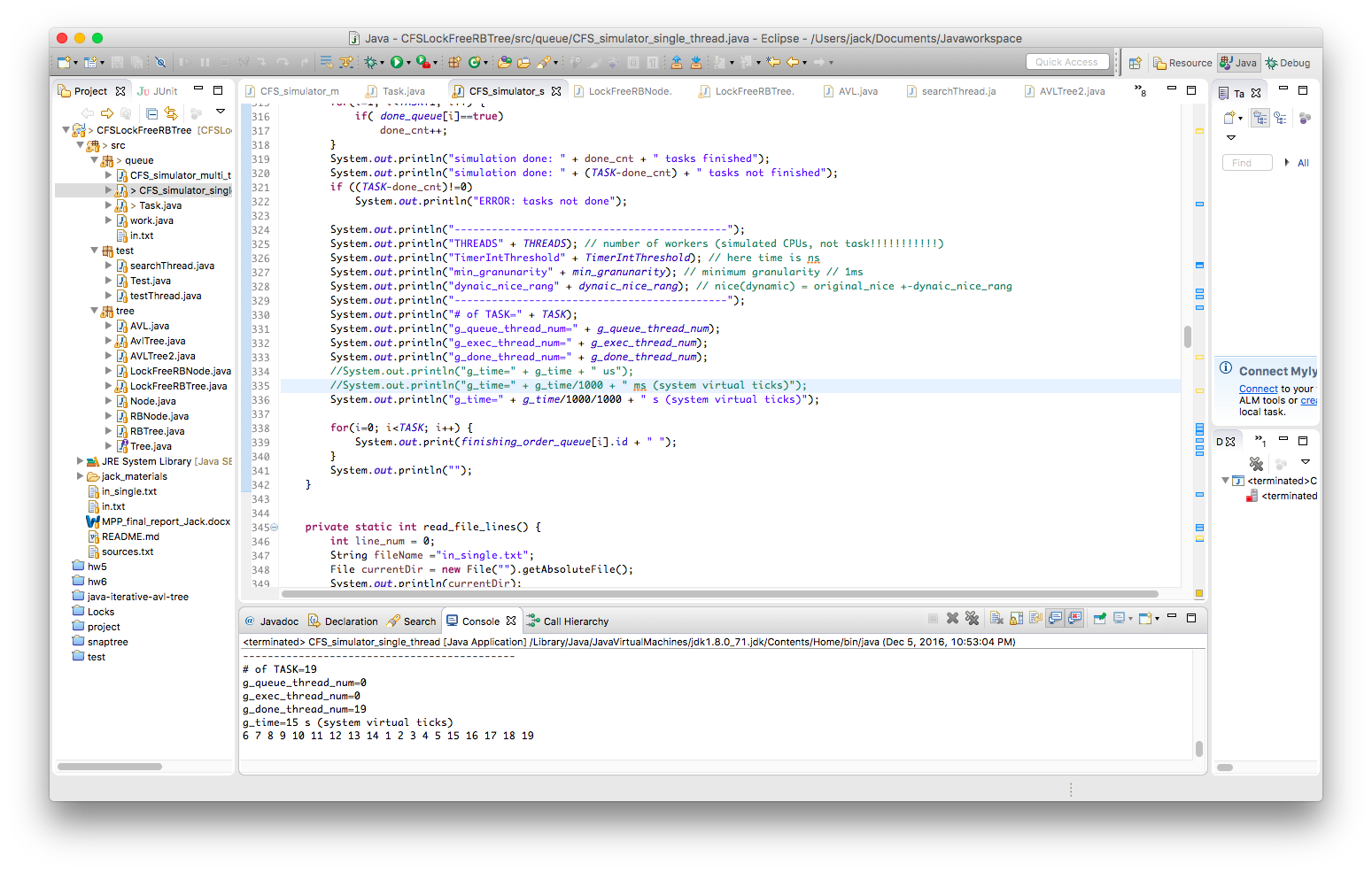
**4.1. CFS part**

**Single-threaded simulator - for checking CFS property**

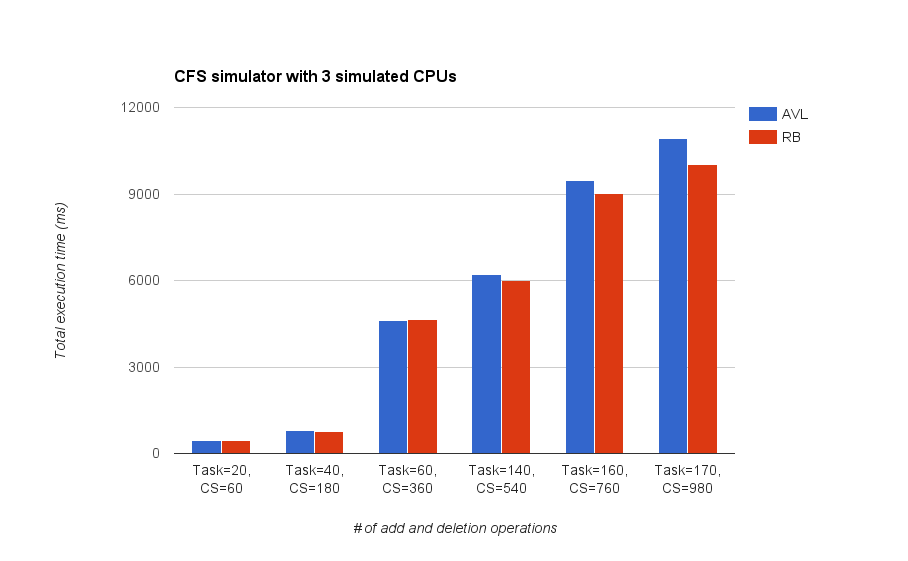
Input task:



Single thread simulator result:

  
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

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



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).

And 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.

**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.

兩個thread同時做完

還是會有先後 不過盡量控制

Our contribution:

* Implementing/modifying AVL-tree & RB-tree
* Implementing parts of lock-free RB-tree methods
* 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.

**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

Note for ppt

concurrent object here is a queue.

timer interrupt: is implemented in a function

一般來說是每1/100秒或每1/1000秒觸發一次Scheduling Tick,讓系統的排程機制可以去計算每個Task執行的時間,並藉此決定是不是要進行Task Context-Switch. (已經做)

**在Linux CFS (Completely Fair Scheduler)的設定下time slice 並非一個固定值. 而是依據正在執行的processes它們的nice value來分配各個process 使用CPU的比例(proportion)**