# Tuning the Linux CFS Scheduler

Stuart Bowman

December 15, 2015

## 1 Introduction

Nearly any modern computing device today has a well balanced and finely tuned scheduling system. As a result, operating system kernels must be widely configurable and flexible among a large array of hardware configurations and applications. Additionally, while it may be easy to make a scheduler that performs with very high throughput, it is a much larger task to build a scheduling system that caters to all types of scheduling criteria (turnaround time, responsiveness, wait time) without causing process starvation or instabilities in system operation.

This paper presents an overview of a few of the main components of cpu scheduling under the Linux kernel as well as a set of tests and benchmarks detailing the observed performance of various scheduler modifications. Additionally, it aims to provide an overview of the Linux CFS scheduling operation as well as discuss several other alternative real-world scheduling algorithms.

## 2 Brief History

In 20+ years of Linux development, the kernel scheduler has undergone several major revisions. Among these revisions are three primary scheduling systems.

**O(N) scheduler** This scheduling system ran through every task in the run queue during each scheduling event, picking and choosing which processes to run. Despite the simplicity of rating and picking processes sequentially at each scheduling routine, this method was not suitable for a wide variety of hardware and not ready for multi-core systems. (ibm)

Linux 2.6.x Saw the change of several different scheduling systems as it served the Linux community from 2003 until 2016 through it's long term support variant. Kernel iterations up to 2.6.22 included the **O(1) scheduler** by Ingo Molnar, an effectively constant time scheduling algorithm designed to replace to formerly costly O(N) scheduler. O(1) made use of a scheduling priority run queue, which allowed it to simply pick the top-most process in constant time. As the kernel matured however, the heuristics of the O(1) run queue became difficult to manage and beginning in 2.6.23 the **Completely Fair Scheduler** (CFS) by Ingo Molnar was introduced. (ibm)

With CFS came a large change in process ordering and organization. CFS operates through a modified variant of round robin scheduling and utilizes a self-balancing red-black tree system. All operations in a red-black tree can be performed in O(log n), including adding nodes, deleting nodes and finding nodes. The CFS red-black tree organizes processes that need the cpu the most on the left of the tree and process that need the cpu the least on the right. A process's need for the cpu is inversely proportional to it's "virtual runtime". The virtual runtime for a given process is calculated as the amount of time assigned to a given process.(ibm)

Additionally, CFS provides a method of dynamic process prioritization through the use of "nice" values. This allows the system a secondary way to adjust a process's need for the cpu and allows users to arbitrarily give certain processes higher priority over others. CFS also has support for various other methods of load balancing between cores, sharing runtime quotas between runqueues and various other rules and features.

Other schedulers such as Con Kolivas' **Rotating Staircase Deadline Scheduler** and his **BFS scheduler** make use of alternative methods for organizing and scheduling processes in the kernel. These are frequently offered in the form of a kernel patch or sometimes as a completely packaged kernel, where the kernel variant is represented by the author's initials, such as **linux-ck**. (ibm)

## 3 Scheduler Flow Control

As defined by the Kernel's documentation on CFS,

"80% of CFS's design can be summed up in a single sentence: CFS basically models an 'ideal, precise multi-tasking CPU' on real hardware."(kernel.org)

Looking inside the kernel source, there are several files of interest that hold particular importance to the development and tuning of the scheduler.

**kernel/sched/sched.h** contains numerous preprocessor definitions, function prototypes, as well as defines constants for use in weighting and organizing processes.

**kernel/sched/core.c** handles the primary scheduling overhead including the main schedule() function, and manages multiple scheduling policies and run queues.

**kernel/sched/fair.c** defines nearly all CFS behaviors and rules such as picking the next task to run, defines constants specific to CFS, and includes structs such as fair_sched_class.

**lib/rbtree.c** contains the class definition for the red-black tree system used by CFS, and includes functions for quickly navigating between adjacent nodes in the tree as well as finding the leftmost node which is typically the most critical process to run.

**include/linux/sched/prio.h** holds several priority constants including minimum and maximum niceness values, and the default values for newly created processes

**include/linux/sched/rt.h** contains, among other things, the Round Robin timeslice multiplier for SCHED_RR which directly impacts the system's final default Round Robin timeslice.

Digging deeper into each of these files revealed some particular areas of interest for modifying scheduling behavior. **Core.c** holds the primary overhead control on scheduling in the kernel which makes it a prime candidate to start exploring. Core.c's primary function is `Schedule()` which handles scheduling flow control between runqueues such as the CFS runqueue. `Schedule()` calls can be triggered by blocking from Mutexes, Semephores and several other elements and it is capable of handling both realtime scheduling policies (Round Robin, FIFO) and non-preemptable policies (Batch).

Schedule() makes a call to `pick_next_task(rq, prev)`, where rq represents the active runqueue and prev represents the current task_struct being executed. `Pick_next_task()` then passes control to the appropriate scheduler class (such as CFS) where the scheduler class will run it's own implementation of `pick_next_task()`. When CFS is the current scheduling class, `fair_sched_class.pick_next_task(rq, prev)` is called, sending control to fair.c's function `pick_next_task_fair(rq,prev)`.

`pick_next_task_fair()` contains a variety of additional rules and regulations and exceptions for choosing tasks. `pick_next_task_fair()` will return the task_struct of the process it selects to run next to core.c and in the event that the CFS chooses to run the idle process, it will return `NULL`.

Under the "Simple" clause of this function, `pick_next_entity(cfs_rq, sched_entity)` is called, where the leftmost node of the red-black tree of tasks is chosen unless overridden to re-run the last process or run the skip buddy. This is effectively the method where processes are actually chosen in the CFS scheduler.

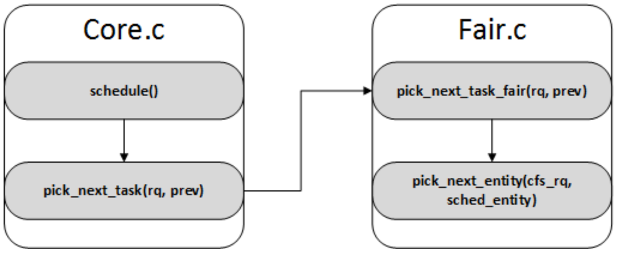

Core.c
schedule()
pick_next_task(rq, prev)
Fair.c
pick_next_task_fair(rq, prev)
pick_next_entity(cfs_rq, sched_entity)

Figure 1: Simplified version of the scheduler's flow control

# 4 Scheduler Modifications

With the above scheduling flow control in mind, we sought out to change and tune the CFS system in some meaningful way. Among the many traits a scheduler is desired to exhibit, we decided to modify the scheduler to measure and tweak it's throughput, scheduling period, and individual process runtime. These tests were not necessarily designed to improve the kernel, but rather to explicitly test how various properties affect it's performance. There were three main source modifications we attempted:

1. Modify the logic behind which process is chosen in `pick_next_entity()`

2. Change the kernel parameters at runtime to cause frequent context switching

3. Modify CFS bandwidth and quota to cause artificial throttling

### 1 Picking Next Entity Changes

The first and most simplistic change involved modifying the operation of `pick_next_entity()` in fair.c. As mentioned above, this function is the last authority queried in the scheduling flow control of CFS, which means process traffic can be changed here. A simplified psudocode version of `pick_next_entity()` is as follows:

```
1  pick_next_entity(cfs_rq, currentProcess)
2
3  SchedulerEntity left = PickFirstEntity()
4  SchedulerEntity se
5
6  if (currentProcess is left of leftmost entity)
7          left = currentProcess
8
9  se = left
10
11 if (cfs_rq->skip should skip se){
12         //skip buddy
13 }
14
15 if (cfs_rq->last and WakeupPreemptEntity())
16         se = cfs_rq->last
17
18 if (cfs_rq->next and WakeupPreemptEntity())
19         se = cfs_rq->next
20
21 return se
```

By removing the additional logic from lines 11-19, we can force the cfs scheduler to strictly run the leftmost (most needed) entity from the red-black tree, and bypass the additional logic used to make exceptions. The resulting psudocode is as follows:

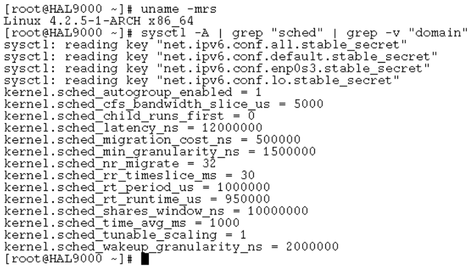
```
pick_next_entity(cfs_rq, currentProcess)

SchedulerEntity left = PickFirstEntity()
SchedulerEntity se

if (currentProcess is left of leftmost entity)
        left = currentProcess

se = left

return se
```

### 2 Changing Kernel Parameters

Changing kernel parameters is not terribly difficult and can be done fairly easily from several different source files before compilation. Areas of focus include the targeted preemption latency and minimum preemption granularity in fair.c, min/max priority niceness values in prio.h, `SCHED_LOAD_SCALE` (which adjusts the number of shares available for the root group) and `RUNTIME_INF` for cfs_rq quota in sched.h. (landley.com)

Some kernel parameters have the benefit of being publically readable and writable during runtime. The utility `sysctl` can be used to read and write to kernel parameters at runtime, changing the behavior of everything from average Round Robin time slicing to CFS bandwidth.

Additionally, many kernel parameters can be read from the /proc folder, providing a secondary resource for details on cgroups, hardware info and process specific information.



```
[root@HAL9000 ~]# uname -mrs
Linux 4.2.5-1-ARCH x86_64
[root@HAL9000 ~]# sysctl -A | grep "sched" | grep -v "domain"
sysctl: reading key "net.ipv6.conf.all.stable_secret"
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.enp0s3.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
kernel.sched_autogroup_enabled = 1
kernel.sched_cfs_bandwidth_slice_us = 5000
kernel.sched_child_runs_first = 0
kernel.sched_latency_ns = 12000000
kernel.sched_migration_cost_ns = 500000
kernel.sched_min_granularity_ns = 1500000
kernel.sched_nr_migrate = 32
kernel.sched_rr_timeslice_ms = 30
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
kernel.sched_shares_window_ns = 10000000
kernel.sched_time_avg_ms = 1000
kernel.sched_tunable_scaling = 1
kernel.sched_wakeup_granularity_ns = 2000000
[root@HAL9000 ~]# █
```

Figure 2: Displaying the default values for the stock Arch Linux Kernel

### 3 CFS Bandwidth and Quota

As it turns out, the Linux scheduling system makes use of time in any way it possibly can. Additionally, it can also be made to limit or even restrict certain groups of processes from using 100% of cpu resources at all times. These features are made possible through several parameters:

**rt_period_us** represents the bandwidth enforcement interval in microseconds (default is 100000000us)

**rt_runtime_us** represents the permitted amount of rt_period_us that may be used during a given period (default is 95000000us)

**runtime_inf** changes the default quota assigned to CFS (default value is disabled, meaning that CFS has unconstrained use of the cpu)

(blaess)(landley)

## 5 Benchmarks

The following benchmarks were performed individually for each of the previously mentioned kernel modifications on an Arch Linux Virtual Machine. To empirically measure the throughput and turnaround time of various workloads, Prime95 was used to stress test the various kernel builds. In addition, a custom built C++ program was used to run a synthetic workload on 16 concurrent child processes, and calculated the average time required to finish each process.

The benchmarks used were as follows:

- Prime 95 Benchmark, sampled when running 25 iterations of 2048K FFT on 1, 2 and 4 cores.

- SyntheticProcess, our custom C++ synthetic load program, running 16 child processes each running a synthetic load and then averaging the completion time of all children (with a 1 sec delay between iterations to ensure the children were dequeued completely).

- A composite test where Prime 95 and SyntheticProcess are run in parallel to evaluate CFS's ability to share between entirely separate pid parents.

The Kernels used in our benchmarks were as follows:

**STOCK** A stock build of Linux 4.2.6

**Test1** A modified version of the stock Kernel, implementing the pick_next_task psudocode changes detailed above.

**Test2** A stock build of Linux 4.2.6 with the following kernel parameters configured using `sysctl`:

sched_rt_period_us = 10,

sched_rt_runtime_us = 10,

sched_rr_timeslice_ms = 1,

sched_cfs_bandwidth_slice_us = 1

**Test3** A modified version of the stock Kernel, implementing the CFS Bandwidth and Quota changes detailed above. Specifically, RUNTIME_INF was set to 10% of the CFS bandwidth period (previously it was unrestricted), refill_cfs_bandwidth_runtime() was short circuited to disable refilling alloted bandwidth, and the sched.h preprocessor constant MAX_SHARES was changed from 2î8 to 29̂.