CFS Bandwidth Control

> **Note**
>
> This document only discusses CPU bandwidth control for SCHED_NORMAL. The SCHED_RT case is covered in Documentation/scheduler/sched-rt-group.rst

CFS bandwidth control is a CONFIG_FAIR_GROUP_SCHED extension which allows the specification of the maximum CPU bandwidth available to a group or hierarchy.

The bandwidth allowed for a group is specified using a quota and period. Within each given "period" (microseconds), a task group is allocated up to "quota" microseconds of CPU time. That quota is assigned to per-cpu run queues in slices as threads in the cgroup become runnable. Once all quota has been assigned any additional requests for quota will result in those threads being throttled. Throttled threads will not be able to run again until the next period when the quota is replenished.

A group's unassigned quota is globally tracked, being refreshed back to cfs_quota units at each period boundary. As threads consume this bandwidth it is transferred to cpu-local "silos" on a demand basis. The amount transferred within each of these updates is tunable and described as the "slice".

Burst feature

This feature borrows time now against our future underrun, at the cost of increased interference against the other system users. All nicely bounded.

Traditional (UP-EDF) bandwidth control is something like:

(U = Sum u_i) <= 1

This guaranteeds both that every deadline is met and that the system is stable. After all, if U were > 1, then for every second of walltime, we'd have to run more than a second of program time, and obviously miss our deadline, but the next deadline will be further out still, there is never time to catch up, unbounded fail.

The burst feature observes that a workload doesn't always executes the full quota; this enables one to describe u_i as a statistical distribution.

For example, have u_i = {x,e}_i, where x is the p(95) and x+e p(100) (the traditional WCET). This effectively allows u to be smaller, increasing the efficiency (we can pack more tasks in the system), but at the cost of missing deadlines when all the odds line up. However, it does maintain stability, since every overrun must be paired with an underrun as long as our x is above the average.

That is, suppose we have 2 tasks, both specify a p(95) value, then we have a p(95)*p(95) = 90.25% chance both tasks are within their quota and everything is good. At the same time we have a p(5)p(5) = 0.25% chance both tasks will exceed their quota at the same time (guaranteed deadline fail). Somewhere in between there's a threshold where one exceeds and the other doesn't underrun enough to compensate; this depends on the specific CDFs.

At the same time, we can say that the worst case deadline miss, will be Sum e_i; that is, there is a bounded tardiness (under the assumption that x+e is indeed WCET).

The interferenece when using burst is valued by the possibilities for missing the deadline and the average WCET. Test results showed that when there many cgroups or CPU is under utilized, the interference is limited. More details are shown in: https://lore.kernel.org/lkml/5371BD36-55AE-4F71-B9D7-B86DC32E3D2B@linux.alibaba.com/

Management

Quota, period and burst are managed within the cpu subsystem via cgroupfs.

> **Note**
>
> The cgroupfs files described in this section are only applicable to cgroup v1. For cgroup v2, see :ref:`Documentation/admin-guide/cgroup-v2.rst <cgroup-v2-cpu>`.
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\scheduler\[linux-master][Documentation][scheduler]sched-bwc.rst`, **line 73**); ***backlink***
> >
> > Unknown interpreted text role "ref".

- cpu.cfs_quota_us: run-time replenished within a period (in microseconds)

- cpu.cfs_period_us: the length of a period (in microseconds)
- cpu.stat: exports throttling statistics [explained further below]
- cpu.cfs_burst_us: the maximum accumulated run-time (in microseconds)

The default values are:

```
cpu.cfs_period_us=100ms
cpu.cfs_quota_us=-1
cpu.cfs_burst_us=0
```

A value of -1 for cpu.cfs_quota_us indicates that the group does not have any bandwidth restriction in place, such a group is described as an unconstrained bandwidth group. This represents the traditional work-conserving behavior for CFS.

Writing any (valid) positive value(s) no smaller than cpu.cfs_burst_us will enact the specified bandwidth limit. The minimum quota allowed for the quota or period is 1ms. There is also an upper bound on the period length of 1s. Additional restrictions exist when bandwidth limits are used in a hierarchical fashion, these are explained in more detail below.

Writing any negative value to cpu.cfs_quota_us will remove the bandwidth limit and return the group to an unconstrained state once more.

A value of 0 for cpu.cfs_burst_us indicates that the group can not accumulate any unused bandwidth. It makes the traditional bandwidth control behavior for CFS unchanged. Writing any (valid) positive value(s) no larger than cpu.cfs_quota_us into cpu.cfs_burst_us will enact the cap on unused bandwidth accumulation.

Any updates to a group's bandwidth specification will result in it becoming unthrottled if it is in a constrained state.

System wide settings

For efficiency run-time is transferred between the global pool and CPU local "silos" in a batch fashion. This greatly reduces global accounting pressure on large systems. The amount transferred each time such an update is required is described as the "slice".

This is tunable via procfs:

```
/proc/sys/kernel/sched_cfs_bandwidth_slice_us (default=5ms)
```

Larger slice values will reduce transfer overheads, while smaller values allow for more fine-grained consumption.

Statistics

A group's bandwidth statistics are exported via 5 fields in cpu.stat.

cpu.stat:

- nr_periods: Number of enforcement intervals that have elapsed.
- nr_throttled: Number of times the group has been throttled/limited.
- throttled_time: The total time duration (in nanoseconds) for which entities of the group have been throttled.
- nr_bursts: Number of periods burst occurs.
- burst_time: Cumulative wall-time (in nanoseconds) that any CPUs has used above quota in respective periods.

This interface is read-only.

Hierarchical considerations

The interface enforces that an individual entity's bandwidth is always attainable, that is: max(c_i) <= C. However, over-subscription in the aggregate case is explicitly allowed to enable work-conserving semantics within a hierarchy:

e.g. Sum (c_i) may exceed C

[Where C is the parent's bandwidth, and c_i its children]

There are two ways in which a group may become throttled:

a. it fully consumes its own quota within a period
b. a parent's quota is fully consumed within its period

In case b) above, even though the child may have runtime remaining it will not be allowed to until the parent's runtime is refreshed.

CFS Bandwidth Quota Caveats

Once a slice is assigned to a cpu it does not expire. However all but 1ms of the slice may be returned to the global pool if all threads on that cpu become unrunnable. This is configured at compile time by the min_cfs_rq_runtime variable. This is a performance tweak that helps prevent added contention on the global lock.

The fact that cpu-local slices do not expire results in some interesting corner cases that should be understood.

For cgroup cpu constrained applications that are cpu limited this is a relatively moot point because they will naturally consume the entirety of their quota as well as the entirety of each cpu-local slice in each period. As a result it is expected that nr_periods roughly equal nr_throttled, and that cpuacct.usage will increase roughly equal to cfs_quota_us in each period.

For highly-threaded, non-cpu bound applications this non-expiration nuance allows applications to briefly burst past their quota limits by the amount of unused slice on each cpu that the task group is running on (typically at most 1ms per cpu or as defined by min_cfs_rq_runtime). This slight burst only applies if quota had been assigned to a cpu and then not fully used or returned in previous periods. This burst amount will not be transferred between cores. As a result, this mechanism still strictly limits the task group to quota average usage, albeit over a longer time window than a single period. This also limits the burst ability to no more than 1ms per cpu. This provides better more predictable user experience for highly threaded applications with small quota limits on high core count machines. It also eliminates the propensity to throttle these applications while simultanously using less than quota amounts of cpu. Another way to say this, is that by allowing the unused portion of a slice to remain valid across periods we have decreased the possibility of wastefully expiring quota on cpu-local silos that don't need a full slice's amount of cpu time.

The interaction between cpu-bound and non-cpu-bound-interactive applications should also be considered, especially when single core usage hits 100%. If you gave each of these applications half of a cpu-core and they both got scheduled on the same CPU it is theoretically possible that the non-cpu bound application will use up to 1ms additional quota in some periods, thereby preventing the cpu-bound application from fully using its quota by that same amount. In these instances it will be up to the CFS algorithm (see sched-design-CFS.rst) to decide which application is chosen to run, as they will both be runnable and have remaining quota. This runtime discrepancy will be made up in the following periods when the interactive application idles.

Examples

1. Limit a group to 1 CPU worth of runtime:

   ```
   If period is 250ms and quota is also 250ms, the group will get
   1 CPU worth of runtime every 250ms.

   # echo 250000 > cpu.cfs_quota_us /* quota = 250ms */
   # echo 250000 > cpu.cfs_period_us /* period = 250ms */
   ```

2. Limit a group to 2 CPUs worth of runtime on a multi-CPU machine

   With 500ms period and 1000ms quota, the group can get 2 CPUs worth of runtime every 500ms:

   ```
   # echo 1000000 > cpu.cfs_quota_us /* quota = 1000ms */
   # echo 500000 > cpu.cfs_period_us /* period = 500ms */

   The larger period here allows for increased burst capacity.
   ```

3. Limit a group to 20% of 1 CPU.

   With 50ms period, 10ms quota will be equivalent to 20% of 1 CPU:

   ```
   # echo 10000 > cpu.cfs_quota_us /* quota = 10ms */
   # echo 50000 > cpu.cfs_period_us /* period = 50ms */
   ```

   By using a small period here we are ensuring a consistent latency response at the expense of burst capacity.

4. Limit a group to 40% of 1 CPU, and allow accumulate up to 20% of 1 CPU additionally, in case accumulation has been done.

   With 50ms period, 20ms quota will be equivalent to 40% of 1 CPU. And 10ms burst will be equivalent to 20% of 1 CPU:

   ```
   # echo 20000 > cpu.cfs_quota_us /* quota = 20ms */
   # echo 50000 > cpu.cfs_period_us /* period = 50ms */
   # echo 10000 > cpu.cfs_burst_us /* burst = 10ms */
   ```

   Larger buffer setting (no larger than quota) allows greater burst capacity.