Real-Time group scheduling

0. WARNING

> Fiddling with these settings can result in an unstable system, the knobs are root only and assumes root knows what he is doing.

Most notable:

- very small values in sched_rt_period_us can result in an unstable system when the period is smaller than either the available hrtimer resolution, or the time it takes to handle the budget refresh itself.
- very small values in sched_rt_runtime_us can result in an unstable system when the runtime is so small the system has difficulty making forward progress (NOTE: the migration thread and kstopmachine both are real-time processes).

1. Overview

1.1 The problem

Realtime scheduling is all about determinism, a group has to be able to rely on the amount of bandwidth (eg. CPU time) being constant. In order to schedule multiple groups of realtime tasks, each group must be assigned a fixed portion of the CPU time available. Without a minimum guarantee a realtime group can obviously fall short. A fuzzy upper limit is of no use since it cannot be relied upon. Which leaves us with just the single fixed portion.

1.2 The solution

CPU time is divided by means of specifying how much time can be spent running in a given period. We allocate this "run time" for each realtime group which the other realtime groups will not be permitted to use.

Any time not allocated to a realtime group will be used to run normal priority tasks (SCHED_OTHER). Any allocated run time not used will also be picked up by SCHED_OTHER.

Let's consider an example: a frame fixed realtime renderer must deliver 25 frames a second, which yields a period of 0.04s per frame. Now say it will also have to play some music and respond to input, leaving it with around 80% CPU time dedicated for the graphics. We can then give this group a run time of 0.8 * 0.04s = 0.032s.

This way the graphics group will have a 0.04s period with a 0.032s run time limit. Now if the audio thread needs to refill the DMA buffer every 0.005s, but needs only about 3% CPU time to do so, it can do with a 0.03 * 0.005s = 0.00015s. So this group can be scheduled with a period of 0.005s and a run time of 0.00015s.

The remaining CPU time will be used for user input and other tasks. Because realtime tasks have explicitly allocated the CPU time they need to perform their tasks, buffer underruns in the graphics or audio can be eliminated.

NOTE: the above example is not fully implemented yet. We still lack an EDF scheduler to make non-uniform periods usable.

2. The Interface

2.1 System wide settings

The system wide settings are configured under the /proc virtual file system:

/proc/sys/kernel/sched_rt_period_us:
: The scheduling period that is equivalent to 100% CPU bandwidth

/proc/sys/kernel/sched_rt_runtime_us:
: A global limit on how much time realtime scheduling may use. Even without CONFIG_RT_GROUP_SCHED enabled, this will limit time reserved to realtime processes. With CONFIG_RT_GROUP_SCHED it signifies the total bandwidth available to all realtime groups.

  - Time is specified in us because the interface is s32. This gives an operating range from 1us to about 35 minutes.
  - sched_rt_period_us takes values from 1 to INT_MAX.
  - sched_rt_runtime_us takes values from -1 to (INT_MAX - 1).
  - A run time of -1 specifies runtime == period, ie. no limit.

2.2 Default behaviour

The default values for sched_rt_period_us (1000000 or 1s) and sched_rt_runtime_us (950000 or 0.95s). This gives 0.05s to be used by SCHED_OTHER (non-RT tasks). These defaults were chosen so that a run-away realtime tasks will not lock up the machine but leave a little time to recover it. By setting runtime to -1 you'd get the old behaviour back.

By default all bandwidth is assigned to the root group and new groups get the period from /proc/sys/kernel/sched_rt_period_us and a run time of 0. If you want to assign bandwidth to another group, reduce the root group's bandwidth and assign some or all of the difference to another group.

Realtime group scheduling means you have to assign a portion of total CPU bandwidth to the group before it will accept realtime tasks. Therefore you will not be able to run realtime tasks as any user other than root until you have done that, even if the user has the rights to run processes with realtime priority!

2.3 Basis for grouping tasks

Enabling CONFIG_RT_GROUP_SCHED lets you explicitly allocate real CPU bandwidth to task groups.

This uses the cgroup virtual file system and "<cgroup>/cpu.rt_runtime_us" to control the CPU time reserved for each control group.

For more information on working with control groups, you should read Documentation/admin-guide/cgroup-v1/cgroups.rst as well.

Group settings are checked against the following limits in order to keep the configuration schedulable:

    Sum_{i} runtime_{i} / global_period <= global_runtime / global_period

For now, this can be simplified to just the following (but see Future plans):

    Sum_{i} runtime_{i} <= global_runtime

3. Future plans

There is work in progress to make the scheduling period for each group ("<cgroup>/cpu.rt_period_us") configurable as well.

The constraint on the period is that a subgroup must have a smaller or equal period to its parent. But realistically its not very useful _yet_ as its prone to starvation without deadline scheduling.

Consider two sibling groups A and B; both have 50% bandwidth, but A's period is twice the length of B's.

- group A: period=100000us, runtime=50000us
    - this runs for 0.05s once every 0.1s
- group B: period= 50000us, runtime=25000us
    - this runs for 0.025s twice every 0.1s (or once every 0.05 sec).

This means that currently a while (1) loop in A will run for the full period of B and can starve B's tasks (assuming they are of lower priority) for a whole period.

The next project will be SCHED_EDF (Earliest Deadline First scheduling) to bring full deadline scheduling to the linux kernel. Deadline scheduling the above groups and treating end of the period as a deadline will ensure that they both get their allocated time.

Implementing SCHED_EDF might take a while to complete. Priority Inheritance is the biggest challenge as the current linux PI infrastructure is geared towards the limited static priority levels 0-99. With deadline scheduling you need to do deadline inheritance (since priority is inversely proportional to the deadline delta (deadline - now)).

This means the whole PI machinery will have to be reworked - and that is one of the most complex pieces of code we have.