Deadline Task Scheduling

0. WARNING

Fiddling with these settings can result in an unpredictable or even unstable system behavior. As for -rt (group) scheduling, it is assumed that root users know what they're doing.

1. Overview

The SCHED_DEADLINE policy contained inside the sched_dl scheduling class is basically an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a mechanism (called Constant Bandwidth Server, CBS) that makes it possible to isolate the behavior of tasks between each other.

2. Scheduling algorithm

2.1 Main algorithm

SCHED_DEADLINE [18] uses three parameters, named "runtime", "period", and "deadline", to schedule tasks. A SCHED_DEADLINE task should receive "runtime" microseconds of execution time every "period" microseconds, and these "runtime" microseconds are available within "deadline" microseconds from the beginning of the period. In order to implement this behavior, every time the task wakes up, the scheduler computes a "scheduling deadline" consistent with the guarantee (using the CBS[2,3] algorithm). Tasks are then scheduled using EDF[1] on these scheduling deadlines (the task with the earliest scheduling deadline is selected for execution). Notice that the task actually receives "runtime" time units within "deadline" if a proper "admission control" strategy (see Section "4. Bandwidth management") is used (clearly, if the system is overloaded this guarantee cannot be respected).

Summing up, the CBS[2,3] algorithm assigns scheduling deadlines to tasks so that each task runs for at most its runtime every period, avoiding any interference between different tasks (bandwidth isolation), while the EDF[1] algorithm selects the task with the earliest scheduling deadline as the one to be executed next. Thanks to this feature, tasks that do not strictly comply with the "traditional" real-time task model (see Section 3) can effectively use the new policy.

In more details, the CBS algorithm assigns scheduling deadlines to tasks in the following way:

- Each SCHED_DEADLINE task is characterized by the "runtime", "deadline", and "period" parameters;
- The state of the task is described by a "scheduling deadline", and a "remaining runtime". These two parameters are initially set to 0;
- When a SCHED_DEADLINE task wakes up (becomes ready for execution), the scheduler checks if:

  ```
            remaining runtime                  runtime
  ----------------------------------    >    ---------
  scheduling deadline - current time           period
  ```

  then, if the scheduling deadline is smaller than the current time, or this condition is verified, the scheduling deadline and the remaining runtime are re-initialized as

  scheduling deadline = current time + deadline remaining runtime = runtime

  otherwise, the scheduling deadline and the remaining runtime are left unchanged;
- When a SCHED_DEADLINE task executes for an amount of time t, its remaining runtime is decreased as:

  ```
  remaining runtime = remaining runtime - t
  ```

  (technically, the runtime is decreased at every tick, or when the task is descheduled / preempted);
- When the remaining runtime becomes less or equal than 0, the task is said to be "throttled" (also known as "depleted" in real-time literature) and cannot be scheduled until its scheduling deadline. The "replenishment time" for this task (see next item) is set to be equal to the current value of the scheduling deadline;
- When the current time is equal to the replenishment time of a throttled task, the scheduling deadline and the remaining runtime are updated as:

  ```
  scheduling deadline = scheduling deadline + period
  remaining runtime = remaining runtime + runtime
  ```

The SCHED_FLAG_DL_OVERRUN flag in sched_attr's sched_flags field allows a task to get informed about runtime overruns through the delivery of SIGXCPU signals.

2.2 Bandwidth reclaiming

Bandwidth reclaiming for deadline tasks is based on the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm [15, 16, 17] and it is enabled when flag SCHED_FLAG_RECLAIM is set.

The following diagram illustrates the state names for tasks handled by GRUB:

```
                             ------------
                 (d)        |   Active   |
              ------------->|            |
              |             | Contending |
              |              ------------
              |                A      |
          ----------           |      |
         |          |          |      |
         | Inactive |          |(b)   | (a)
         |          |          |      |
          ----------           |      |
              A                |      V
              |              ------------
              |             |   Active   |
              --------------|     Non    |
                 (c)        | Contending |
                             ------------
```

A task can be in one of the following states:

- ActiveContending: if it is ready for execution (or executing);
- ActiveNonContending: if it just blocked and has not yet surpassed the 0-lag time;
- Inactive: if it is blocked and has surpassed the 0-lag time.

State transitions:

a. When a task blocks, it does not become immediately inactive since its bandwidth cannot be immediately reclaimed without breaking the real-time guarantees. It therefore enters a transitional state called ActiveNonContending. The scheduler arms the "inactive timer" to fire at the 0-lag time, when the task's bandwidth can be reclaimed without breaking the real-time guarantees.

   The 0-lag time for a task entering the ActiveNonContending state is computed as:

   ```
                   (runtime * dl_period)
      deadline - ---------------------
                        dl_runtime
   ```

   where runtime is the remaining runtime, while dl_runtime and dl_period are the reservation parameters.

b. If the task wakes up before the inactive timer fires, the task re-enters the ActiveContending state and the "inactive timer" is canceled. In addition, if the task wakes up on a different runqueue, then the task's utilization must be removed from the previous runqueue's active utilization and must be added to the new runqueue's active utilization. In order to avoid races between a task waking up on a runqueue while the "inactive timer" is running on a different CPU, the "dl_non_contending" flag is used to indicate that a task is not on a runqueue but is active (so, the flag is set when the task blocks and is cleared when the "inactive timer" fires or when the task wakes up).

c. When the "inactive timer" fires, the task enters the Inactive state and its utilization is removed from the runqueue's active utilization.

d. When an inactive task wakes up, it enters the ActiveContending state and its utilization is added to the active utilization of the runqueue where it has been enqueued.

For each runqueue, the algorithm GRUB keeps track of two different bandwidths:

- Active bandwidth (running_bw): this is the sum of the bandwidths of all tasks in active state (i.e., ActiveContending or ActiveNonContending);
- Total bandwidth (this_bw): this is the sum of all tasks "belonging" to the runqueue, including the tasks in Inactive state.

The algorithm reclaims the bandwidth of the tasks in Inactive state. It does so by decrementing the runtime of the executing task Ti at a pace equal to

    dq = -max{ Ui / Umax, (1 - Uinact - Uextra) } dt

where:

- Ui is the bandwidth of task Ti;
- Umax is the maximum reclaimable utilization (subjected to RT throttling limits);
- Uinact is the (per runqueue) inactive utilization, computed as (this_bq - running_bw);
- Uextra is the (per runqueue) extra reclaimable utilization (subjected to RT throttling limits).

Let's now see a trivial example of two deadline tasks with runtime equal to 4 and period equal to 8 (i.e., bandwidth equal to 0.5):

```
         A            Task T1
         |
         |                               |
         |                               |
         |--------                       |----
         |       |                       V
         |---|---|---|---|---|---|---|---|--------->t
         0   1   2   3   4   5   6   7   8


         A            Task T2
         |
         |                               |
         |                               |
         |       ------------------------|
         |       |                       V
         |---|---|---|---|---|---|---|---|--------->t
         0   1   2   3   4   5   6   7   8


         A            running_bw
         |
       1 -----------------               ------
         |               |               |
      0.5-               -----------------
         |                               |
         |---|---|---|---|---|---|---|---|--------->t
         0   1   2   3   4   5   6   7   8
```

```
- Time t = 0:

  Both tasks are ready for execution and therefore in ActiveContending state.
  Suppose Task T1 is the first task to start execution.
  Since there are no inactive tasks, its runtime is decreased as dq = -1 dt.

- Time t = 2:

  Suppose that task T1 blocks
  Task T1 therefore enters the ActiveNonContending state. Since its remaining
  runtime is equal to 2, its 0-lag time is equal to t = 4.
  Task T2 start execution, with runtime still decreased as dq = -1 dt since
  there are no inactive tasks.

- Time t = 4:

  This is the 0-lag time for Task T1. Since it didn't woken up in the
  meantime, it enters the Inactive state. Its bandwidth is removed from
  running_bw.
  Task T2 continues its execution. However, its runtime is now decreased as
  dq = - 0.5 dt because Uinact = 0.5.
  Task T2 therefore reclaims the bandwidth unused by Task T1.

- Time t = 8:

  Task T1 wakes up. It enters the ActiveContending state again, and the
  running_bw is incremented.
```

2.3 Energy-aware scheduling

When cpufreq's schedutil governor is selected, SCHED_DEADLINE implements the GRUB-PA [19] algorithm, reducing the CPU operating frequency to the minimum value that still allows to meet the deadlines. This behavior is currently implemented only for ARM architectures.

A particular care must be taken in case the time needed for changing frequency is of the same order of magnitude of the reservation period. In such cases, setting a fixed CPU frequency results in a lower amount of deadline misses.

3. Scheduling Real-Time Tasks

Warning

This section contains a (not-thorough) summary on classical deadline scheduling theory, and how it applies to SCHED_DEADLINE. The reader can "safely" skip to Section 4 if only interested in seeing how the scheduling policy can be used. Anyway, we strongly recommend to come back here and continue reading (once the urge for testing is satisfied :P) to be sure of fully understanding all technical details.

There are no limitations on what kind of task can exploit this new scheduling discipline, even if it must be said that it is particularly suited for periodic or sporadic real-time tasks that need guarantees on their timing behavior, e.g., multimedia, streaming, control applications, etc.

3.1 Definitions

A typical real-time task is composed of a repetition of computation phases (task instances, or jobs) which are activated on a periodic or sporadic fashion. Each job J_j (where J_j is the j^th job of the task) is characterized by an arrival time r_j (the time when the job starts), an amount of computation time c_j needed to finish the job, and a job absolute deadline d_j, which is the time within which the job should be finished. The maximum execution time max{c_j} is called "Worst Case Execution Time" (WCET) for the task. A real-time task can be periodic with period P if r_{j+1} = r_j + P, or sporadic with minimum inter-arrival time P is r_{j+1} >= r_j + P. Finally, d_j = r_j + D, where D is the task's relative deadline. Summing up, a real-time task can be described as

Task = (WCET, D, P)

The utilization of a real-time task is defined as the ratio between its WCET and its period (or minimum inter-arrival time), and represents the fraction of CPU time needed to execute the task.

If the total utilization U=sum(WCET_i/P_i) is larger than M (with M equal to the number of CPUs), then the scheduler is unable to respect all the deadlines. Note that total utilization is defined as the sum of the utilizations WCET_i/P_i over all the real-time tasks in the system. When considering multiple real-time tasks, the parameters of the i-th task are indicated with the "_i" suffix. Moreover, if the total utilization is larger than M, then we risk starving non- real-time tasks by real-time tasks. If, instead, the total utilization is smaller than M, then non real-time tasks will not be starved and the system might be able to respect all the deadlines. As a matter of fact, in this case it is possible to provide an upper bound for tardiness (defined as the maximum between 0 and the difference between the finishing time of a job and its absolute deadline). More precisely, it can be proven that using a global EDF scheduler the maximum tardiness of each task is smaller or equal than

((M âˆ’ 1) Â· WCET_max âˆ’ WCET_min)/(M âˆ’ (M âˆ’ 2) Â· U_max) + WCET_max

where WCET_max = max{WCET_i} is the maximum WCET, WCET_min=min{WCET_i} is the minimum WCET, and U_max = max{WCET_i/P_i} is the maximum utilization[12].

3.2 Schedulability Analysis for Uniprocessor Systems

If M=1 (uniprocessor system), or in case of partitioned scheduling (each real-time task is statically assigned to one and only one CPU), it is possible to formally check if all the deadlines are respected. If D_i = P_i for all tasks, then EDF is able to respect all the deadlines of all the tasks executing on a CPU if and only if the total utilization of the tasks running on such a CPU is smaller or equal than 1. If D_i != P_i for some task, then it is possible to define the density of a task as WCET_i/min{D_i,P_i}, and EDF is able to respect all the deadlines of all the tasks running on a CPU if the sum of the densities of the tasks running on such a CPU is smaller or equal than 1:

sum(WCET_i / min{D_i, P_i}) <= 1

It is important to notice that this condition is only sufficient, and not necessary: there are task sets that are schedulable, but do not respect the condition. For example, consider the task set {Task_1,Task_2} composed by Task_1=(50ms,50ms,100ms) and Task_2=(10ms,100ms,100ms). EDF is clearly able to schedule the two tasks without missing any deadline (Task_1 is scheduled as soon as it is released, and finishes just in time to respect its deadline; Task_2 is scheduled immediately after Task_1, hence its response time cannot be larger than 50ms + 10ms = 60ms) even if

50 / min{50,100} + 10 / min{100, 100} = 50 / 50 + 10 / 100 = 1.1

Of course it is possible to test the exact schedulability of tasks with D_i != P_i (checking a condition that is both sufficient and necessary), but this cannot be done by comparing the total utilization or density with a constant. Instead, the so called "processor demand" approach can be used, computing the total amount of CPU time h(t) needed by all the tasks to respect all of their deadlines in a time interval of size t, and comparing such a time with the interval size t. If h(t) is smaller than t (that is, the amount of time needed by the tasks in a time interval of size t is smaller than the size of the interval) for all the possible values of t, then EDF is able to schedule the tasks respecting all of their deadlines. Since performing this check for all possible values of t is impossible, it has been proven[4,5,6] that it is sufficient to perform the test for values of t between 0 and a maximum value L. The cited papers contain all of the mathematical details and explain how to compute h(t) and L. In any case, this kind of analysis is too complex as well as too time-consuming to be performed on-line. Hence, as explained in Section 4 Linux uses an admission test based on the tasks' utilizations.

3.3 Schedulability Analysis for Multiprocessor Systems

On multiprocessor systems with global EDF scheduling (non partitioned systems), a sufficient test for schedulability can not be based on the utilizations or densities: it can be shown that even if D_i = P_i task sets with utilizations slightly larger than 1 can miss deadlines regardless of the number of CPUs.

Consider a set {Task_1,...Task_{M+1}} of M+1 tasks on a system with M CPUs, with the first task Task_1=(P,P,P) having period, relative deadline and WCET equal to P. The remaining M tasks Task_i=(e,P-1,P-1) have an arbitrarily small worst case execution time (indicated as "e" here) and a period smaller than the one of the first task. Hence, if all the tasks activate at the same time t, global EDF schedules these M tasks first (because their absolute deadlines are equal to t + P - 1, hence they are smaller than the absolute deadline of Task_1, which is t + P). As a result, Task_1 can be scheduled only at time t + e, and will finish at time t + e + P, after its absolute deadline. The total utilization of the task set is U = M Â· e / (P - 1) + P / P = M Â· e / (P - 1) + 1, and for small values of e this can become very close to 1. This is known as "Dhall's effect"[7]. Note: the example in the original paper by Dhall has been slightly simplified here (for example, Dhall more correctly computed lim_{e->0}U).

More complex schedulability tests for global EDF have been developed in real-time literature[8,9], but they are not based on a simple comparison between total utilization (or density) and a fixed constant. If all tasks have D_i = P_i, a sufficient schedulability condition can be expressed in a simple way:

sum(WCET_i / P_i) <= M - (M - 1) Â· U_max

where U_max = max{WCET_i / P_i}[10]. Notice that for U_max = 1, M - (M - 1) Â· U_max becomes M - M + 1 = 1 and this schedulability condition just confirms the Dhall's effect. A more complete survey of the literature about schedulability tests for multi-processor real-time scheduling can be found in [11].

As seen, enforcing that the total utilization is smaller than M does not guarantee that global EDF schedules the tasks without missing any deadline (in other words, global EDF is not an optimal scheduling algorithm). However, a total utilization smaller than M is enough to guarantee that non real-time tasks are not starved and that the tardiness of real-time tasks has an upper bound[12] (as previously noted). Different bounds on the maximum tardiness experienced by real-time tasks have been developed in various papers[13,14], but the theoretical result that is important for SCHED_DEADLINE is that if the total utilization is smaller or equal than M then the response times of the tasks are limited.

3.4 Relationship with SCHED_DEADLINE Parameters

Finally, it is important to understand the relationship between the SCHED_DEADLINE scheduling parameters described in Section 2 (runtime, deadline and period) and the real-time task parameters (WCET, D, P) described in this section. Note that the tasks' temporal constraints are represented by its absolute deadlines d_j = r_j + D described above, while SCHED_DEADLINE schedules the tasks according to scheduling deadlines (see Section 2). If an admission test is used to guarantee that the scheduling deadlines are respected, then SCHED_DEADLINE can be used to schedule real-time tasks guaranteeing that all the jobs' deadlines of a task are respected. In order to do this, a task must be scheduled by setting:

- runtime >= WCET
- deadline = D
- period <= P

IOW, if runtime >= WCET and if period is <= P, then the scheduling deadlines and the absolute deadlines (d_j) coincide, so a proper admission control allows to respect the jobs' absolute deadlines for this task (this is what is called "hard schedulability property" and is an extension of Lemma 1 of [2]). Notice that if runtime > deadline the admission control will surely reject this task, as it is not possible to respect its temporal constraints.

References:

1 - C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the Association for Computing Machinery, 20(1), 1973.

2 - L. Abeni , G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. Proceedings of the 19th IEEE Real-time Systems Symposium, 1998. http://retis.sssup.it/~giorgio/paps/1998/rtss98-cbs.pdf

3 - L. Abeni. Server Mechanisms for Multimedia Applications. ReTiS Lab Technical Report. http://disi.unitn.it/~abeni/tr-98-01.pdf

4 - J. Y. Leung and M.L. Merril. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. Information Processing Letters, vol. 11, no. 3, pp. 115-118, 1980.

5 - S. K. Baruah, A. K. Mok and L. E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. Proceedings of the 11th IEEE Real-time Systems

Symposium, 1990.
6 - S. K. Baruah, L. E. Rosier and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time tasks on One Processor. Real-Time Systems Journal, vol. 4, no. 2, pp 301-324, 1990.
7 - S. J. Dhall and C. L. Liu. On a real-time scheduling problem. Operations research, vol. 26, no. 1, pp 127-140, 1978.
8 - T. Baker. Multiprocessor EDF and Deadline Monotonic Schedulability Analysis. Proceedings of the 24th IEEE Real-Time Systems Symposium, 2003.
9 - T. Baker. An Analysis of EDF Schedulability on a Multiprocessor. IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 8, pp 760-768, 2005.
10 - J. Goossens, S. Funk and S. Baruah, Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. Real-Time Systems Journal, vol. 25, no. 2â€“3, pp. 187â€“205, 2003.
11 - R. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. ACM Computing Surveys, vol. 43, no. 4, 2011. http://www-users.cs.york.ac.uk/~robdavis/papers/MPSurveyv5.0.pdf
12 - U. C. Devi and J. H. Anderson. Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. Real-Time Systems Journal, vol. 32, no. 2, pp 133-189, 2008.
13 - P. Valente and G. Lipari. An Upper Bound to the Lateness of Soft Real-Time Tasks Scheduled by EDF on Multiprocessors. Proceedings of the 26th IEEE Real-Time Systems Symposium, 2005.
14 - J. Erickson, U. Devi and S. Baruah. Improved tardiness bounds for Global EDF. Proceedings of the 22nd Euromicro Conference on Real-Time Systems, 2010.
15 - G. Lipari, S. Baruah, Greedy reclamation of unused bandwidth in constant-bandwidth servers, 12th IEEE Euromicro Conference on Real-Time Systems, 2000.
16 - L. Abeni, J. Lelli, C. Scordino, L. Palopoli, Greedy CPU reclaiming for SCHED DEADLINE. In Proceedings of the Real-Time Linux Workshop (RTLWS), Dusseldorf, Germany, 2014.
17 - L. Abeni, G. Lipari, A. Parri, Y. Sun, Multicore CPU reclaiming: parallel or sequential?. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016.
18 - J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the Linux kernel, Software: Practice and Experience, 46(6): 821-839, June 2016.
19 - C. Scordino, L. Abeni, J. Lelli, Energy-Aware Real-Time Scheduling in the Linux Kernel, 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), Pau, France, April 2018.

4. Bandwidth management

As previously mentioned, in order for -deadline scheduling to be effective and useful (that is, to be able to provide "runtime" time units within "deadline"), it is important to have some method to keep the allocation of the available fractions of CPU time to the various tasks under control. This is usually called "admission control" and if it is not performed, then no guarantee can be given on the actual scheduling of the -deadline tasks.

As already stated in Section 3, a necessary condition to be respected to correctly schedule a set of real-time tasks is that the total utilization is smaller than M. When talking about -deadline tasks, this requires that the sum of the ratio between runtime and period for all tasks is smaller than M. Notice that the ratio runtime/period is equivalent to the utilization of a "traditional" real-time task, and is also often referred to as "bandwidth". The interface used to control the CPU bandwidth that can be allocated to -deadline tasks is similar to the one already used for -rt tasks with real-time group scheduling (a.k.a. RT-throttling - see Documentation/scheduler/sched-rt-group.rst), and is based on readable/ writable control files located in procfs (for system wide settings). Notice that per-group settings (controlled through cgroupfs) are still not defined for -deadline tasks, because more discussion is needed in order to figure out how we want to manage SCHED_DEADLINE bandwidth at the task group level.

A main difference between deadline bandwidth management and RT-throttling is that -deadline tasks have bandwidth on their own (while -rt ones don't!), and thus we don't need a higher level throttling mechanism to enforce the desired bandwidth. In other words, this means that interface parameters are only used at admission control time (i.e., when the user calls sched_setattr()). Scheduling is then performed considering actual tasks' parameters, so that CPU bandwidth is allocated to SCHED_DEADLINE tasks respecting their needs in terms of granularity. Therefore, using this simple interface we can put a cap on total utilization of -deadline tasks (i.e., Sum (runtime_i / period_i) < global_dl_utilization_cap).

4.1 System wide settings

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

For now the -rt knobs are used for -deadline admission control and the -deadline runtime is accounted against the -rt runtime. We realize that this isn't entirely desirable; however, it is better to have a small interface for now, and be able to change it easily later. The ideal situation (see 5.) is to run -rt tasks from a -deadline server; in which case the -rt bandwidth is a direct subset of dl_bw.

This means that, for a root_domain comprising M CPUs, -deadline tasks can be created while the sum of their bandwidths stays below:

> M * (sched_rt_runtime_us / sched_rt_period_us)

It is also possible to disable this bandwidth management logic, and be thus free of oversubscribing the system up to any arbitrary level. This is done by writing -1 in /proc/sys/kernel/sched_rt_runtime_us.

4.2 Task interface

Specifying a periodic/sporadic task that executes for a given amount of runtime at each instance, and that is scheduled according to the urgency of its own timing constraints needs, in general, a way of declaring:

- a (maximum/typical) instance execution time,
- a minimum interval between consecutive instances,
- a time constraint by which each instance must be completed.

Therefore:

- a new struct sched_attr, containing all the necessary fields is provided;
- the new scheduling related syscalls that manipulate it, i.e., sched_setattr() and sched_getattr() are implemented.

For debugging purposes, the leftover runtime and absolute deadline of a SCHED_DEADLINE task can be retrieved through /proc/<pid>/sched (entries dl.runtime and dl.deadline, both values in ns). A programmatic way to retrieve these values from production code is under discussion.

4.3 Default behavior

The default value for SCHED_DEADLINE bandwidth is to have rt_runtime equal to 950000. With rt_period equal to 1000000, by default, it means that -deadline tasks can use at most 95%, multiplied by the number of CPUs that compose the root_domain, for each root_domain. This means that non -deadline tasks will receive at least 5% of the CPU time, and that -deadline tasks will receive their runtime with a guaranteed worst-case delay respect to the "deadline" parameter. If "deadline" = "period" and the cpuset mechanism is used to implement partitioned scheduling (see Section 5), then this simple setting of the bandwidth management is able to deterministically guarantee that -deadline tasks will receive their runtime in a period.

Finally, notice that in order not to jeopardize the admission control a -deadline task cannot fork.

4.4 Behavior of sched_yield()

When a SCHED_DEADLINE task calls sched_yield(), it gives up its remaining runtime and is immediately throttled, until the next period, when its runtime will be replenished (a special flag dl_yielded is set and used to handle correctly throttling and runtime replenishment after a call to sched_yield()).

This behavior of sched_yield() allows the task to wake-up exactly at the beginning of the next period. Also, this may be useful in the future with bandwidth reclaiming mechanisms, where sched_yield() will make the leftoever runtime available for reclamation by other SCHED_DEADLINE tasks.

5. Tasks CPU affinity

-deadline tasks cannot have an affinity mask smaller that the entire root_domain they are created on. However, affinities can be specified through the cpuset facility (Documentation/admin-guide/cgroup-v1/cpusets.rst).

5.1 SCHED_DEADLINE and cpusets HOWTO

An example of a simple configuration (pin a -deadline task to CPU0) follows (rt-app is used to create a -deadline task):

```
mkdir /dev/cpuset
mount -t cgroup -o cpuset cpuset /dev/cpuset
cd /dev/cpuset
mkdir cpu0
echo 0 > cpu0/cpuset.cpus
echo 0 > cpu0/cpuset.mems
echo 1 > cpuset.cpu_exclusive
echo 0 > cpuset.sched_load_balance
echo 1 > cpu0/cpuset.cpu_exclusive
echo 1 > cpu0/cpuset.mem_exclusive
echo $$ > cpu0/tasks
rt-app -t 100000:10000:d:0 -D5 # it is now actually superfluous to specify
                               # task affinity
```

6. Future plans

Still missing:

- programmatic way to retrieve current runtime and absolute deadline
- refinements to deadline inheritance, especially regarding the possibility of retaining bandwidth isolation among non-interacting tasks. This is being studied from both theoretical and practical points of view, and hopefully we should be able to produce some demonstrative code soon;
- (c)group based bandwidth management, and maybe scheduling;
- access control for non-root users (and related security concerns to address), which is the best way to allow unprivileged use of the mechanisms and how to prevent non-root users "cheat" the system?

As already discussed, we are planning also to merge this work with the EDF throttling patches [https://lore.kernel.org/r/cover.1266931410.git.fabio@helm.retis] but we still are in the preliminary phases of the merge and we really seek feedback that would help us decide on the direction it should take.

Appendix A. Test suite

The SCHED_DEADLINE policy can be easily tested using two applications that are part of a wider Linux Scheduler validation suite. The suite is available as a GitHub repository: https://github.com/scheduler-tools.

The first testing application is called rt-app and can be used to start multiple threads with specific parameters. rt-app supports SCHED_{OTHER,FIFO,RR,DEADLINE} scheduling policies and their related parameters (e.g., niceness, priority, runtime/deadline/period). rt-app is a valuable tool, as it can be used to synthetically recreate certain workloads (maybe mimicking real use-cases) and evaluate how the scheduler behaves under such workloads. In this way, results are easily reproducible. rt-app is available at: https://github.com/scheduler-tools/rt-app.

Thread parameters can be specified from the command line, with something like this:

```
# rt-app -t 100000:10000:d -t 150000:20000:f:10 -D5
```

The above creates 2 threads. The first one, scheduled by SCHED_DEADLINE, executes for 10ms every 100ms. The second one, scheduled at SCHED_FIFO priority 10, executes for 20ms every 150ms. The test will run for a total of 5 seconds.

More interestingly, configurations can be described with a json file that can be passed as input to rt-app with something like this:

```
# rt-app my_config.json
```

The parameters that can be specified with the second method are a superset of the command line options. Please refer to rt-app documentation for more details (*<rt-app-sources>/doc/*.json*).

The second testing application is a modification of schedtool, called schedtool-dl, which can be used to setup SCHED_DEADLINE parameters for a certain pid/application. schedtool-dl is available at: https://github.com/scheduler-tools/schedtool-dl.git.

The usage is straightforward:

```
# schedtool -E -t 10000000:100000000 -e ./my_cpuhog_app
```

With this, my_cpuhog_app is put to run inside a SCHED_DEADLINE reservation of 10ms every 100ms (note that parameters are expressed in microseconds). You can also use schedtool to create a reservation for an already running application, given that you know its pid:

```
# schedtool -E -t 10000000:100000000 my_app_pid
```

Appendix B. Minimal main()

We provide in what follows a simple (ugly) self-contained code snippet showing how SCHED_DEADLINE reservations can be created by a real-time application developer:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <sys/syscall.h>
#include <pthread.h>

#define gettid() syscall(__NR_gettid)

#define SCHED_DEADLINE       6

/* XXX use the proper syscall numbers */
#ifdef __x86_64__
#define __NR_sched_setattr           314
#define __NR_sched_getattr           315
#endif

#ifdef __i386__
#define __NR_sched_setattr           351
#define __NR_sched_getattr           352
#endif

#ifdef __arm__
#define __NR_sched_setattr           380
#define __NR_sched_getattr           381
#endif

static volatile int done;

struct sched_attr {
     __u32 size;

     __u32 sched_policy;
     __u64 sched_flags;

     /* SCHED_NORMAL, SCHED_BATCH */
     __s32 sched_nice;

     /* SCHED_FIFO, SCHED_RR */
     __u32 sched_priority;

     /* SCHED_DEADLINE (nsec) */
     __u64 sched_runtime;
     __u64 sched_deadline;
     __u64 sched_period;
};

int sched_setattr(pid_t pid,
               const struct sched_attr *attr,
               unsigned int flags)
{
     return syscall(__NR_sched_setattr, pid, attr, flags);
}

int sched_getattr(pid_t pid,
               struct sched_attr *attr,
               unsigned int size,
               unsigned int flags)
{
     return syscall(__NR_sched_getattr, pid, attr, size, flags);
}

void *run_deadline(void *data)
{
     struct sched_attr attr;
     int x = 0;
     int ret;
     unsigned int flags = 0;

     printf("deadline thread started [%ld]\n", gettid());

     attr.size = sizeof(attr);
     attr.sched_flags = 0;
     attr.sched_nice = 0;
     attr.sched_priority = 0;

     /* This creates a 10ms/30ms reservation */
     attr.sched_policy = SCHED_DEADLINE;
     attr.sched_runtime = 10 * 1000 * 1000;
     attr.sched_period = attr.sched_deadline = 30 * 1000 * 1000;

     ret = sched_setattr(0, &attr, flags);
     if (ret < 0) {
             done = 0;
             perror("sched_setattr");
             exit(-1);
     }

     while (!done) {
             x++;
     }

     printf("deadline thread dies [%ld]\n", gettid());
     return NULL;
}

int main (int argc, char **argv)
{
     pthread_t thread;

     printf("main thread [%ld]\n", gettid());

     pthread_create(&thread, NULL, run_deadline, NULL);

     sleep(10);

     done = 1;
     pthread_join(thread, NULL);

     printf("main dies [%ld]\n", gettid());
     return 0;
}
```