Deadline Scheduler

Open Issues

Daniel Bristot de Oliveira
Red Hat, Inc.
Who is Daniel?
Real-time systems

Systems which deal with external events with timing constraints
  - Real from real/external-world
  - Time from timing constraints

The response of an event is correct if and only if:
  - The logical response is correct
  - It is produced within a deadline
Real-time scheduler modeling

- A system is viewed as a “model”
  - A system is composed by a set of $n$ tasks
  - A task is a set of infinity recurring jobs.
  - Each task is characterized by some parameters:
    - $C$ or $Q = \text{WCET or Budget}$
    - $T$ or $P = \text{Period or Minimum inter-arrival time}$
    - $D = \text{Deadline}$
on Linux, DL tasks are characterized by:

- `dl_period` = Period [sporadic || periodic]
- `dl_deadline` = Relative deadline [ by default == period ... but can be < ]
- `dl_runtime` = Execution time;
Regarding Period:

- **Periodic**
- **Sporadic**
- **Aperiodic**
Regarding Deadline:

- Implicit
- Constrained
- Arbitrary
Why EDF scheduler?
Fixed Priority  versus  Deadline
EDF is optimal!

*Under optimal conditions*
EDF is optimal (\(U \leq 1\)) with
- If tasks does not misbehave
- Job does not suspend (dequeue/enqueue) during an activation
- Implicit deadline (deadline == period)
- Uniprocessor

Note:
\[
[U]\text{utilization} = \frac{C}{T} \text{ (or } \frac{Q}{P}, \text{ runtime/period)}
\]
\[
[D]\text{ensity} = \frac{C}{D} \text{ (or } \frac{Q}{D}, \text{ runtime/deadline)}
\]
So, let’s explore each point!
What if a task runs longer than it said (C) it was suppose to run?

Or

What if the utilization goes higher than 100%?
The domino effect
To avoid the domino problem...

- Admission control to avoid overload:
  - The sum of the Utilization of all tasks cannot be higher than 
    $\frac{rt\_period - rt\_runtime}{rt\_period}$ (by default 95%).

- CBS to avoid a misbehaving task to run more than runtime.
CBS: Constant Bandwidth Server

- Throttle a misbehaving task that uses more than allowed
- Try to provide runtime CPU time every period.
  - It relies on non-suspending tasks.
CBS & Suspending task

By assuming non-suspending tasks...

- It is implicitly assumed that, when queued, the **absolute U** of a task is bound to its **relative U** (U=runtime/period).
- In other words: The task will never overload the system.
- If the task suspends/blocks, that might not be the case...
For example, a task with $U = 3/9$ blocks with $2/8$
Returning with $U=2/3$

Notation: C/P & C/D/P
CBS & Self-suspending tasks

- CBS wakeup rule (ensures that a task will not overload the system):
  
  If the **deadline** is in the **past**:
  
  new absolute **runtime** and absolute **deadline** is set.

  If the **deadline** is in the **future**:
  
  If the **possible U < allowed U**
  
  Go ahead and run, my little reservation.

  else

  Reset runtime, set the new deadline
Replenish the runtime and reset period

Notation: C/P & C/D/P
In the presence of another deadline task...

Original deadline

New deadline

Notation: C/P & C/D/P
What do we care more, having runtime/period after a wakeup or try to make the deadline?
Revised CBS & Self-suspending tasks
- CBS wakeup rule (ensures that a task will not overload the system):
  If the **deadline** is in the **past**:
    new absolute **runtime** and absolute **deadline** is set.
  If the deadline is in the future:
    If the **absolute U < relative U**
      Go ahead and run, my little CBS.
    else
      Truncate runtime, new runtime = \((C / P) \times \text{laxity}\)
Using the revised CBS:

\[ U_a = \frac{2}{8} \]

\[ U_a = \frac{3}{9} \]

\[ U_b = \frac{4}{9} \]

As \( U_a \geq \frac{2}{3} \Rightarrow \frac{3}{9} \)

\( U_a = \frac{1}{3} \)

Notation: C/P & C/D/P
Should we consider using the revised CBS?
Constrained deadline

- Linux’s deadline scheduler accepts task with deadline $\leq$ period.
- In the presence of an implicit deadline task, the admission test is not valid to “guarantee” the deadline, even on single-core systems.
- For example, two tasks with $3/10$ (60%) but deadline of 5:

\[ \begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 & \quad 12 & \quad 13 & \quad 14 & \quad 15 & \quad 16 & \quad 17 & \quad 18 \\
\end{align*} \]

Notation: C/P & C/D/P
That is easy!
We should use runtime/deadline, not runtime/period!
No, it is too pessimistic...

Notation: C/P & C/D/P
There is one case in which we decided to use it, with revised CBS...
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Self-suspended constrained deadline task

Notation: C/P & C/D/P
Revised CBS & Suspending & Constrained DL

- CBS wakeup rule (ensures that a task will not overload the system):
  
  If the **deadline** is in the **past**:
  
  If the next period is in the future:
  
  Throttle waiting the next period;

  else

  new absolute **runtime** and absolute **deadline** is set.

  If the deadline is in the future:

  If the absolute **Density** is < relative **Density**

  Go ahead and run, my little CBS.

  else

  **Truncate runtime**, new runtime = \( \frac{C}{D} \times \text{laxity} \)
Self-suspending constrained deadline task

Notation: C/P & C/D/P
Mamma mia!
Things are confuse for deadline < period & Self-suspending!?!?!?!?
Suspending + constrained deadline is a REAL open issue.
Let’s talk about multi-processor scheduling
Multi-processor scheduling

A scheduler can be classified as:

- **Partitioned**: When each scheduler manages a single CPU
- **Global**: When a single scheduler manages all M CPUs of the system
- **Clustered**: When a single scheduler manages a disjoint subset of the M CPUs
- A CPU cannot belong to two “domains”.

Scheduler domain

<table>
<thead>
<tr>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>Clusters</th>
<th>Clusters</th>
</tr>
</thead>
</table>

Global

CPUs
Let’s talk about global scheduling!
Global scheduling
Global scheduling adds a lot of anomalies. For instance, there is no critical instant.
- Release all tasks at same time is not the worst case anymore
“Obvious things” are not obvious anymore:
- Reducing the load of a schedulable taskset does not turn guarantee the task set will still schedulable...
Dhall’s effect

Notation: C/P & C/D/P
Reducing the load...

Notation: C/P & C/D/P
Increasing a little bit... BOOM!

Notation: C/P & C/D/P
Taking Dhall’s effect in account, an admission test would be:

- $\sum(U) \leq M - (M - 1) \times U_{\text{max}}$
- Where $U_{\text{max}}$ is the highest $U$ of all tasks
Solution: Partitioned + Clustered
What if those small tasks were per-cpu tasks?
So should we always use partitioned?
How about this scenario?

Notation: C/P & C/D/P
Neither partitioned nor global are optimal...
Is there anything else we could?
The word is: semi-partitioned
Let’s take this scenario:

Notation: C/P & C/D/P
Let’s pin some tasks:

Notation: C/P & C/D/P
Then, we split the other one....

Notation: C/P & C/D/P
Hey hey hey! Didn’t you say constrained deadline tasks are a problem?
They are not always a problem:

Notation: C/P & C/D/P
And voilà!

Notation: C/P & C/D/P
How good is this idea?
Empirically, near-optimal hard real-time schedulability — usually ≥99% schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).
Online semi-partitioned comparison:
Online semi-partitioned comparison:
Affinity! For almost free

Affinity for global scheduling is a problem
For semi-partitioned... it is not.
- Just one more input to the heuristics
- Possible to make a “per-cpu fake load” to reserve time for CFS
- DL Server to schedule CFS: Hierarchical scheduler
  - A re-implementation of RT Throttling:
    - [PATCH] sched/rt: RT_RUNTIME_GREED sched feature
    - https://lkml.org/lkml/2016/11/7/55
We still have arguments for another talk
But I am being throttled...
Questions?
Thank you! Obrigado! Grazie Mille!