SCHED_DEADLINE: What’s next (?)

Daniel Bristot de Oliveira
Juri Lelli

Real-time Linux Summit 2019

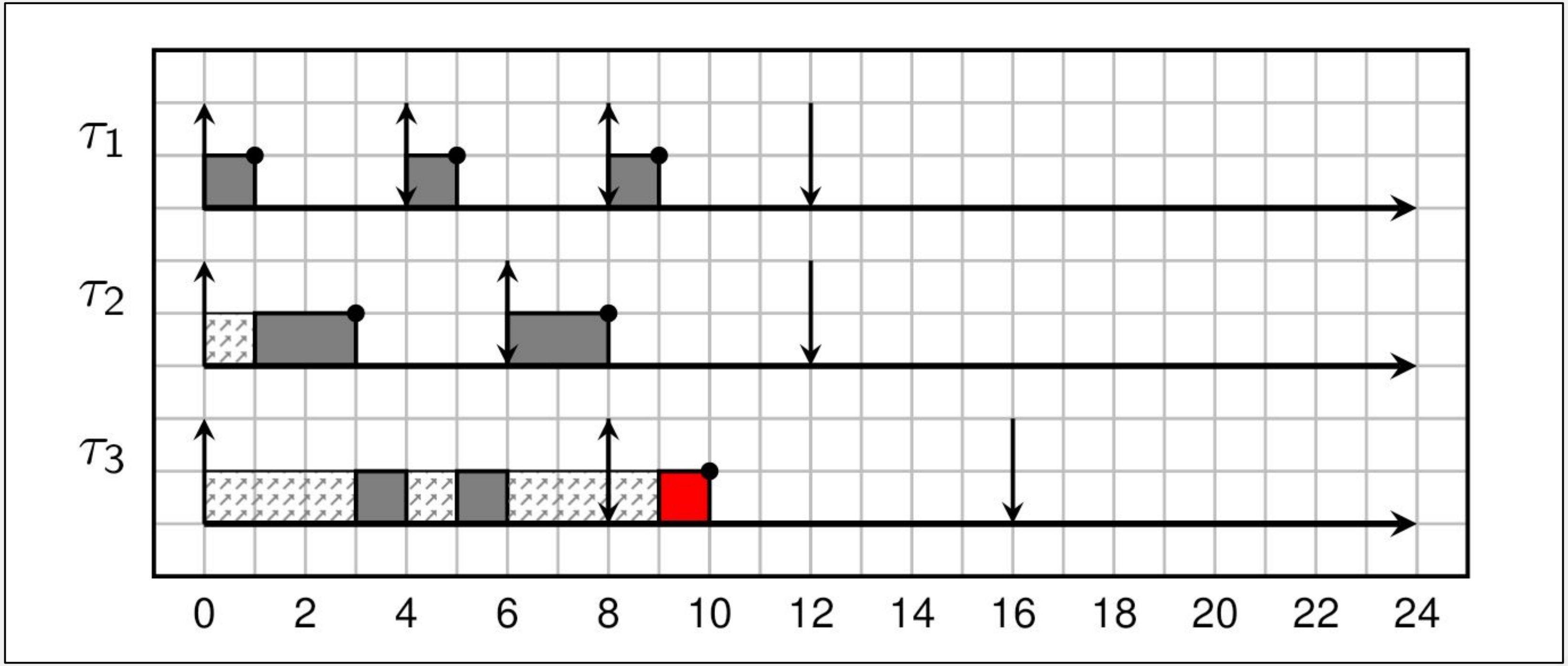
Before start

Real-time scheduling - an exercise

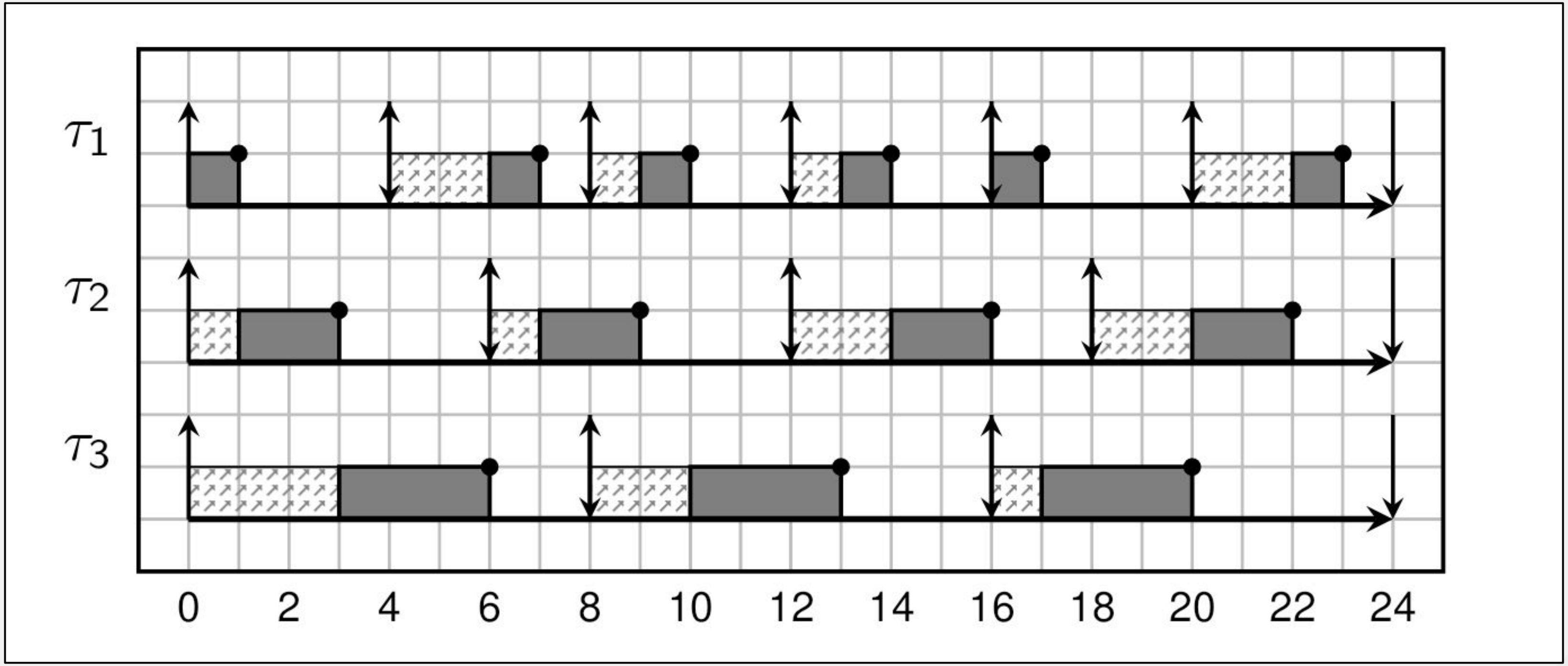
In a system with the following periodic real-time tasks:

Task	WCET	Period = Deadline	U
t1	1	4	0.250
t2	2	6	0.330
t3	3	8	0.375
∑(U)			0.958 (< 1)

Task-level Fixed Priority: Fixed priority RM



Job-level Fixed Priority - EDF



That is why people like deadline scheduler.

Other advantages of sched deadline

- User do not need to “chose” the priorities
    - The user set the runtime and period of tasks
- Miss behave tasks do not cause damage on the system
- The workload of the system is known
  - This allowed the development of other features like:
    - GRUB: That allows a task to run for a longer by using the time not used by other task!
    - GRUP-PA: That allows a processor to scale down the frequency when the system is not overloaded.
    - Always without missing deadlines!
  -

But there is still some work to be done!

Non-root usage

```
To: linux-rt-users@xxxxxxxxxxxxxxx

Subject: SCHED_DEADLINE as user

From: <xxxxxxxxxxxxxxxx>

Date: Wed, 15 Aug 2018 14:08:20 +0800

...

i wonder, what's the preferred way to obtain SCHED_DEADLINE privileges
as non-root user?
for SCHED_RR/SCHED_FIFO i'm typically using pam_limits/limits.conf, but
i haven't found any resources on how SCHED_DEADLINE can be obtained ...

... it's a showstopper for using it in audio applications, which are running
as user.
```

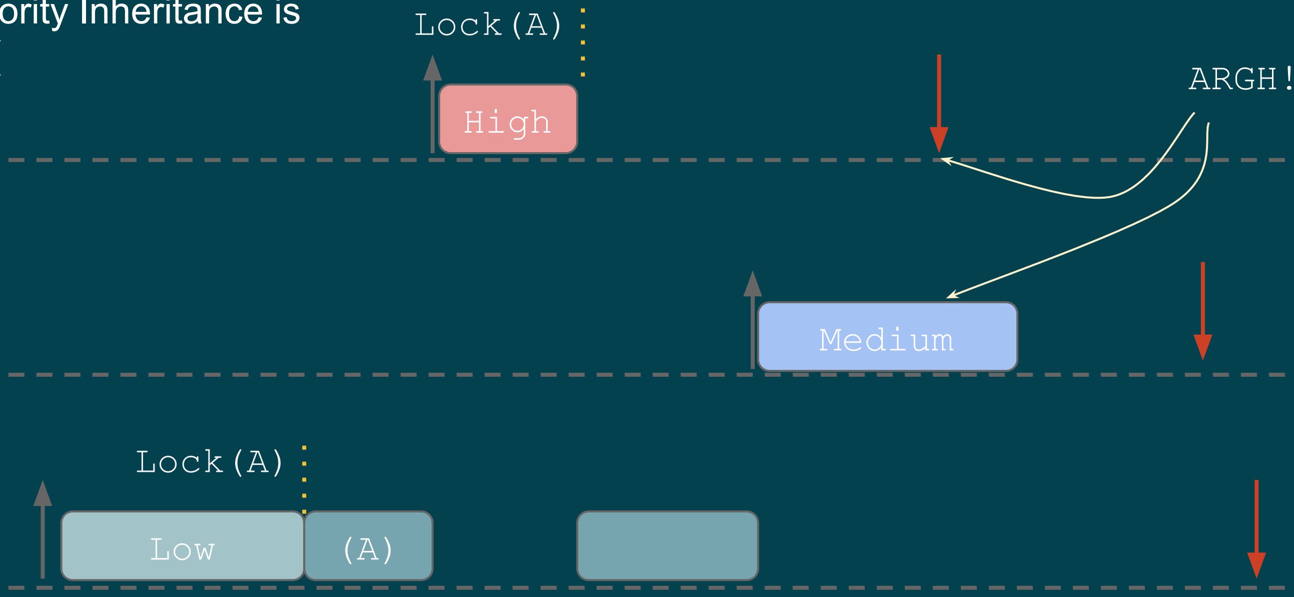
Non-root usage

- Only ROOT can `sched_setattr()` to `SCHED_DEADLINE`
- Lack of a sane and safe Priority Inheritance mechanism
  - Today: deadline inheritance w/o runtime enforcement
  - We need: bandwidth inheritance w/ enforcement (proxy exec.)

Better Priority Inheritance (AKA proxy execution)

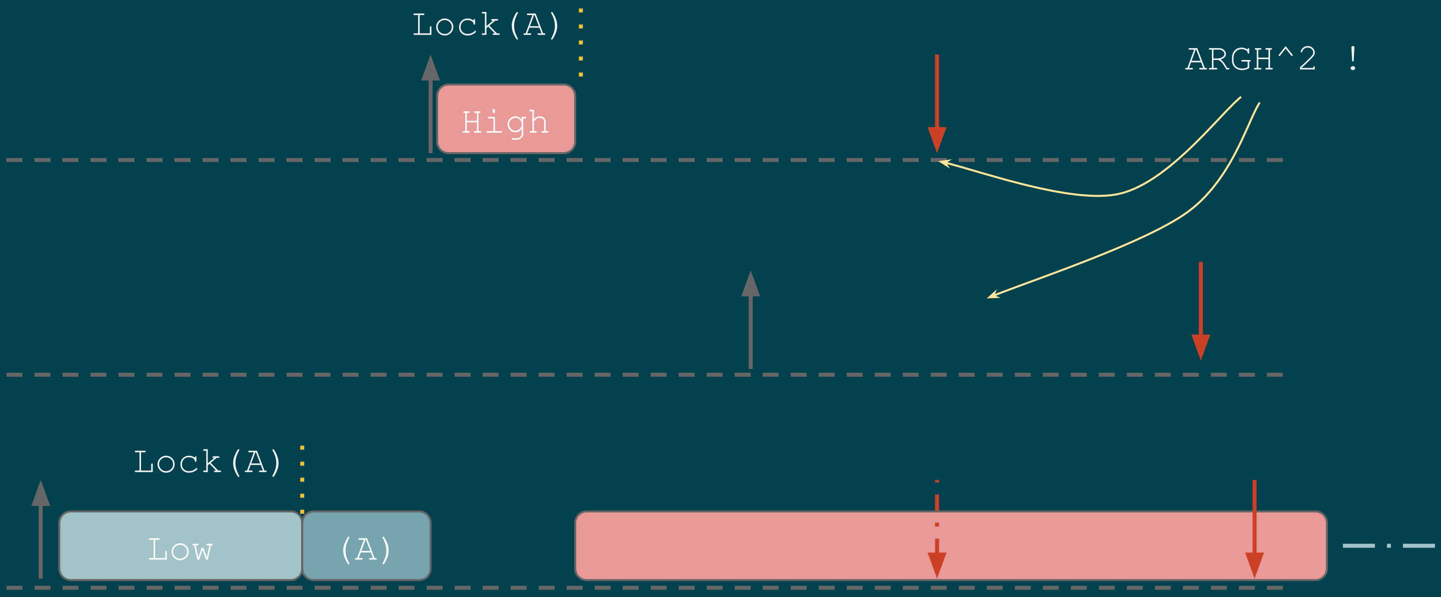
Proxy execution

No Priority Inheritance is bad :-(



Priority inheritance for Sched Deadline = Deadline inheritance

Proxy execution



It can be worse than not having priority inheritance at all!

Proxy execution

- What's the problem ?
- Current Priority Inheritance mechanism is not safe for !root
  - Deadline inheritance (… also slightly incorrect)
  - Priority boosted tasks are outside runtime enforcement

- We would need to **inherit donors' bandwidth** (runtime/period)
- And keep **runtime enforcement on** while doing that
- Basically let the mutex owner **execute using the scheduling context** of a (several) donor(s)

Proxy execution

High’s task_struct

SCHEDULING

Info for implementing a policy, e.g.

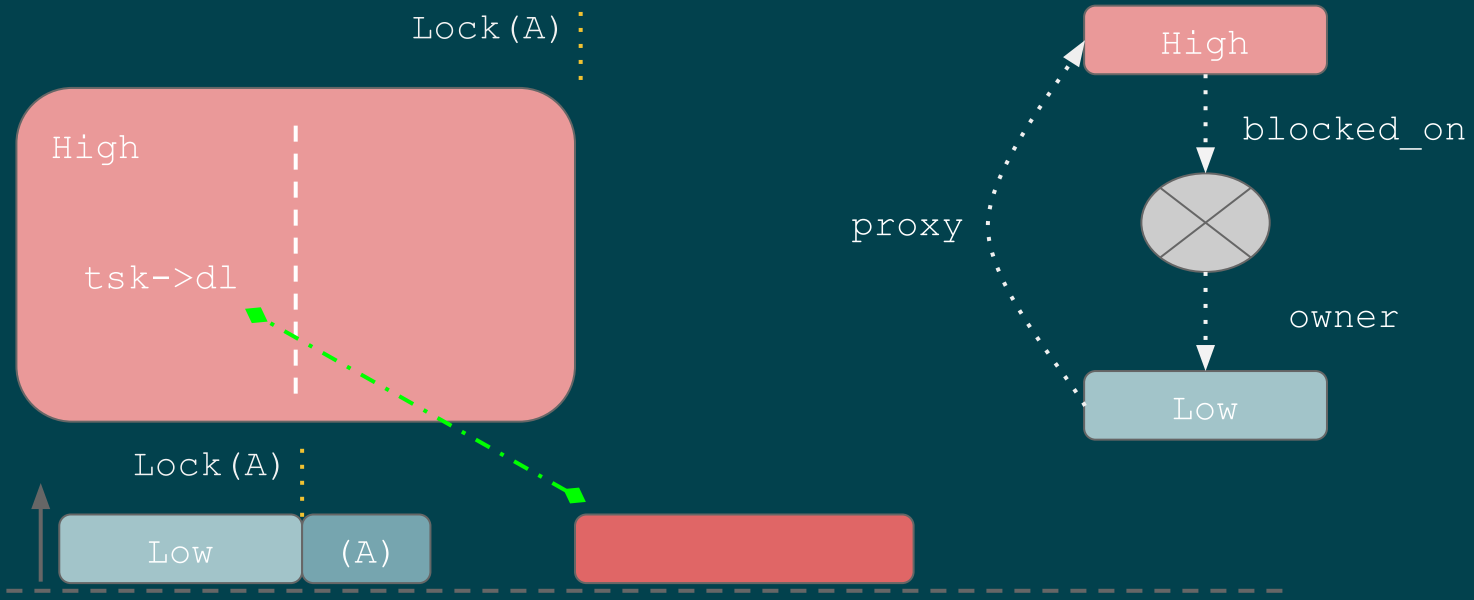
- tsk->se
- tsk->rt
- tsk->dl

EXECUTION

Info for running the task, e.g.

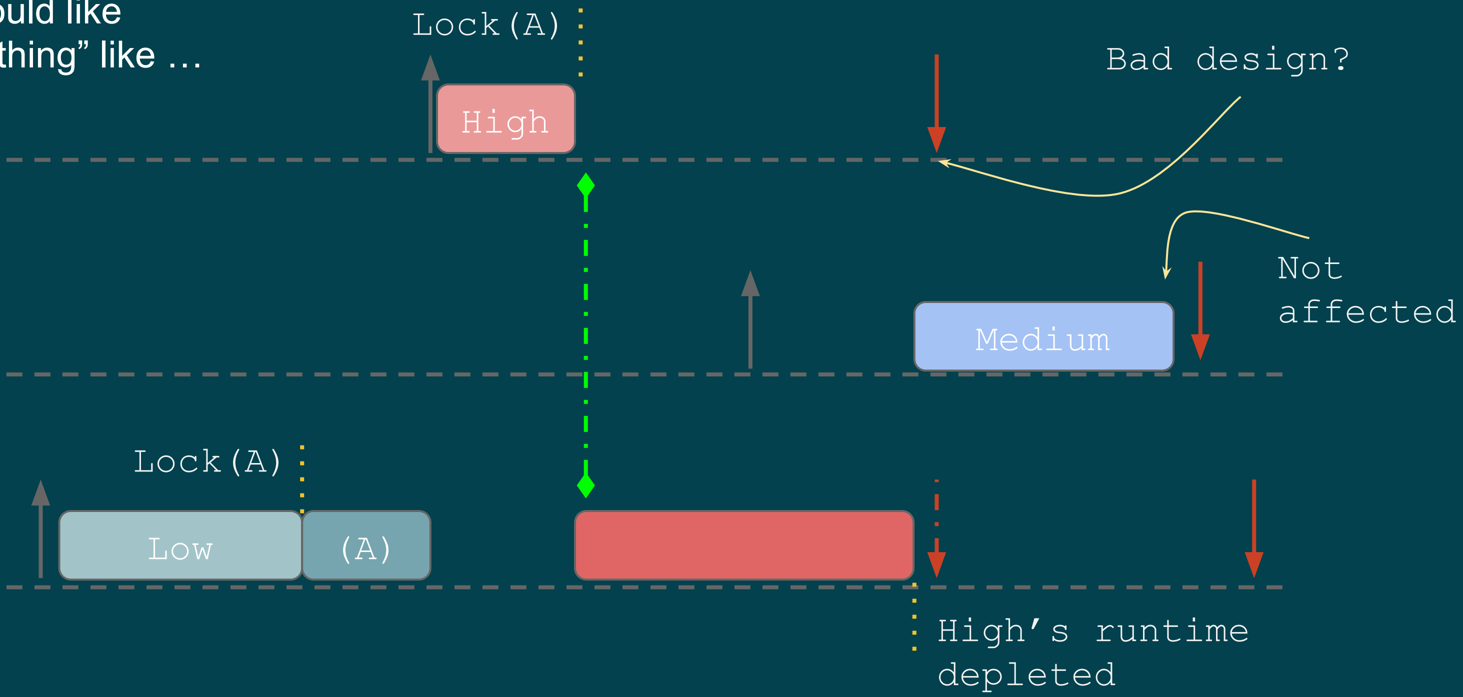
- affinity

Proxy execution



Proxy execution

We would like
“something” like …



Proxy execution

- **More general** than Priority Inheritance for SCHED_DEADLINE
- Could be applied to **other synch mechanisms** (e.g., cond. var., yield_to like calls)
- “Boosted” task could inherit **additional properties**, e.g.
  - NICE
  - RT prio
  - Utilization clamping values
  - …

Cgroups support

Cgroups support

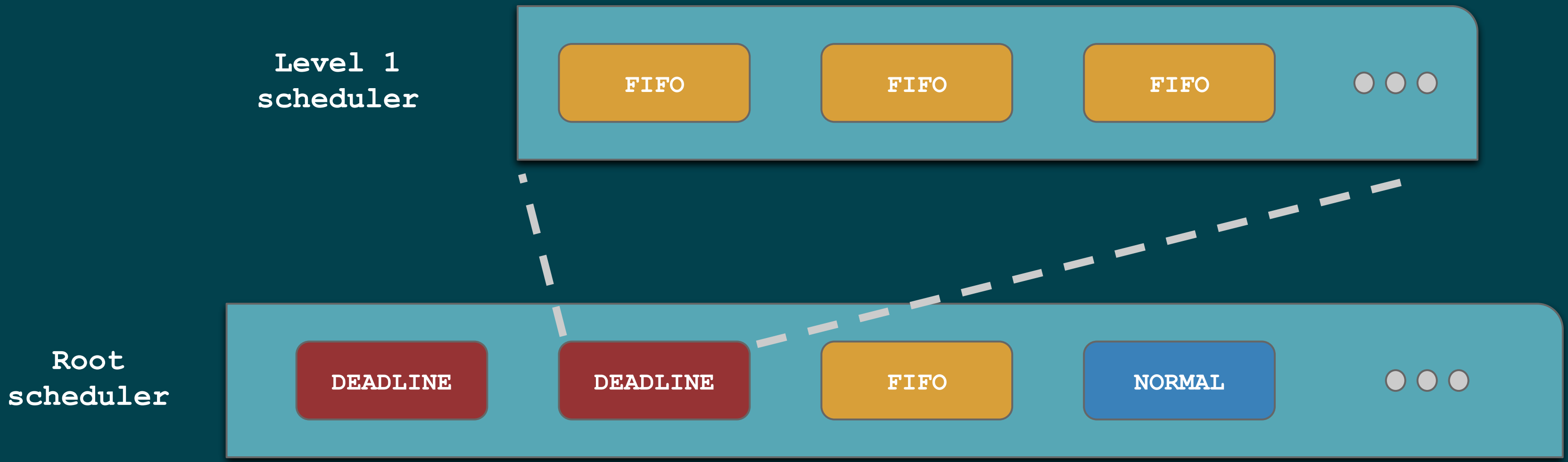
- Cgroups based bandwidth management
- Hierarchical scheduling

Cgroups support

- Cgroups based bandwidth management
  - System administrator could reserve a fraction of total bandwidth to users
  - Users would add tasks to this reservation
    - Sharing the same reservation

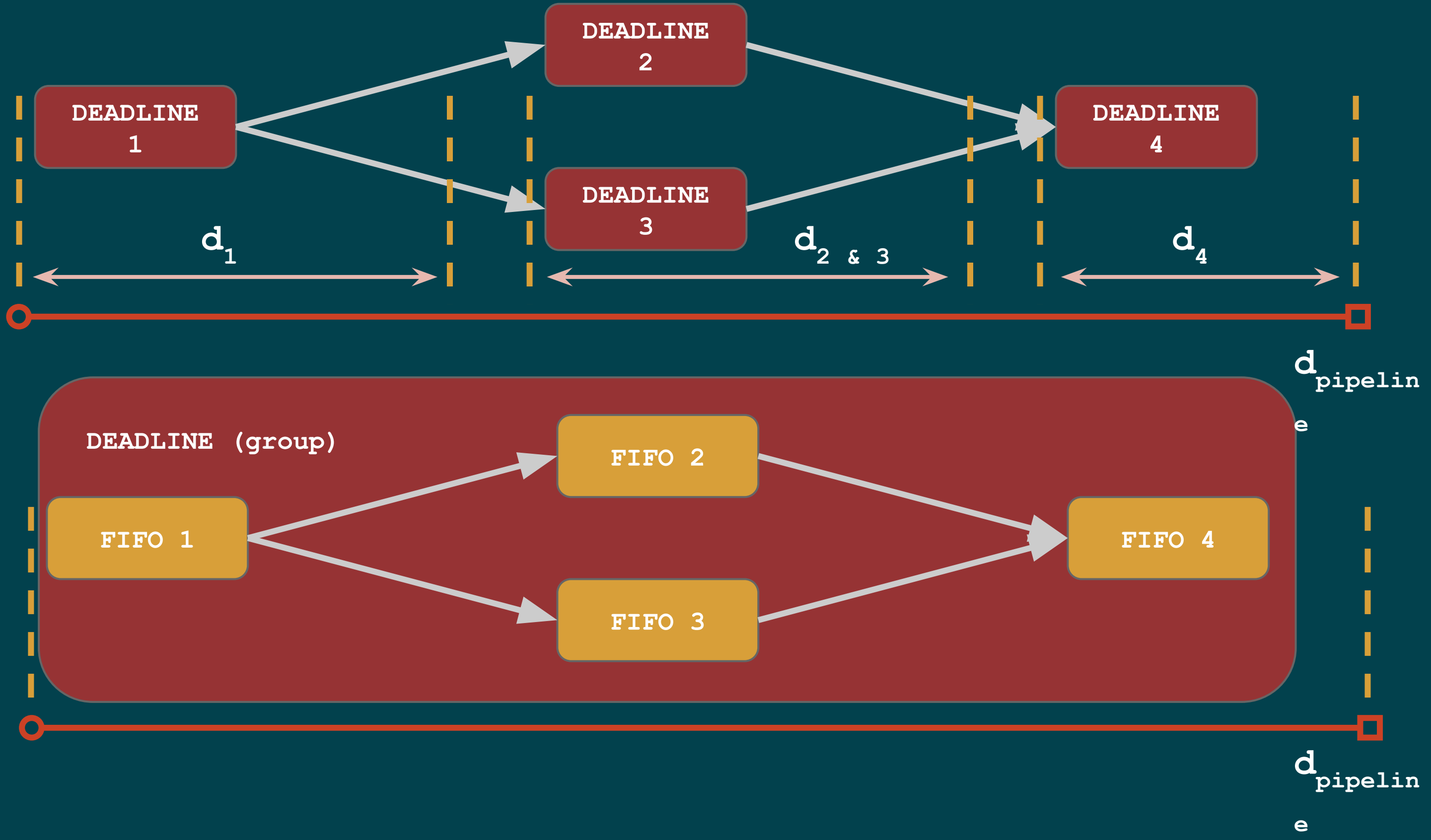
Cgroups support

- Hierarchical scheduling - Hierarchical Constant Bandwidth Server (H-CBS)
    - Nest SCHED_{FIFO,RR} entities within SCHED_DEADLINE

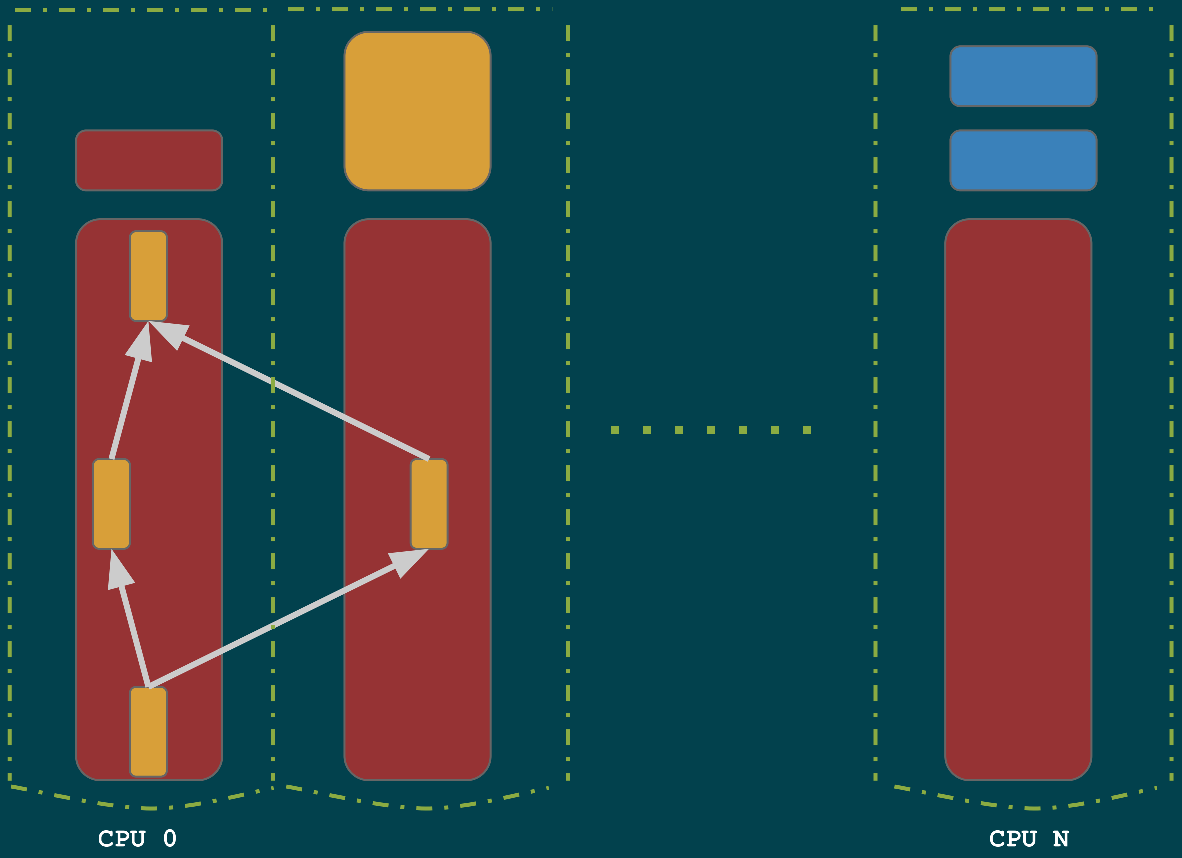


This allows the creation of pipelines

Cgroups support



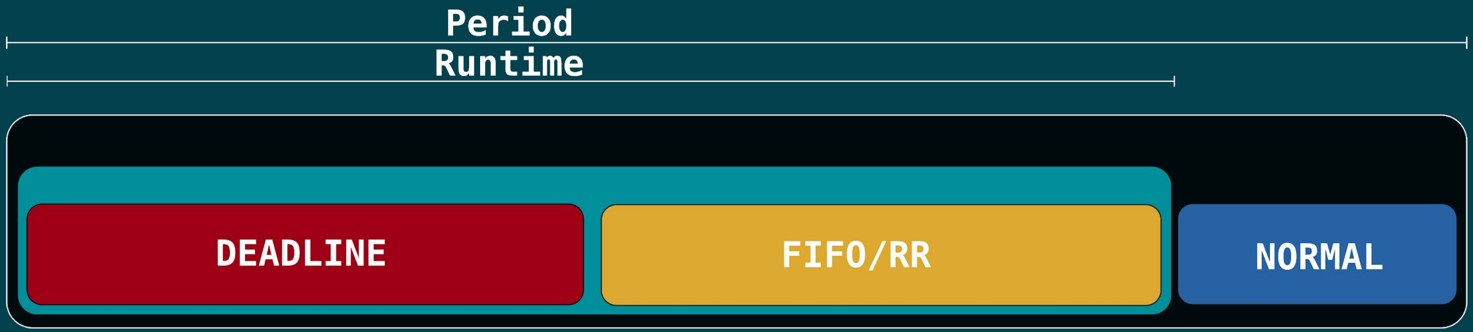
Cgroups support



Re-working RT Throttling to use DL servers

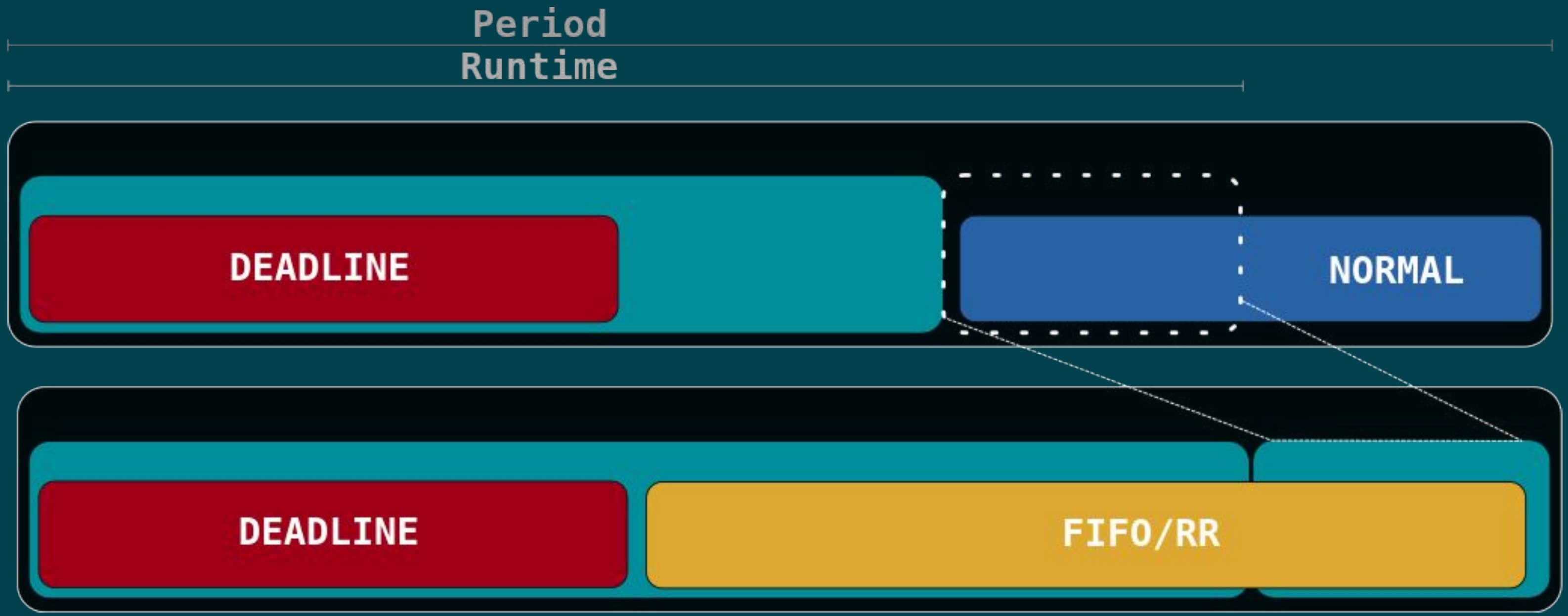
RT Throttling

- The real-time throttling mechanism is a safeguard for misbehaving real-time tasks
- The idea is to avoid real-time tasks starving non-rt tasks
- By default, real-time tasks can run:
  - kernel.sched_rt_runtime_us / kernel.sched_rt_period_us
    - 950000 / 1000000



RT Throttling

- For SMP, it is also possible to share runtime among the runqueues of the same sched domain (RT_RUNTIME_SHARE).



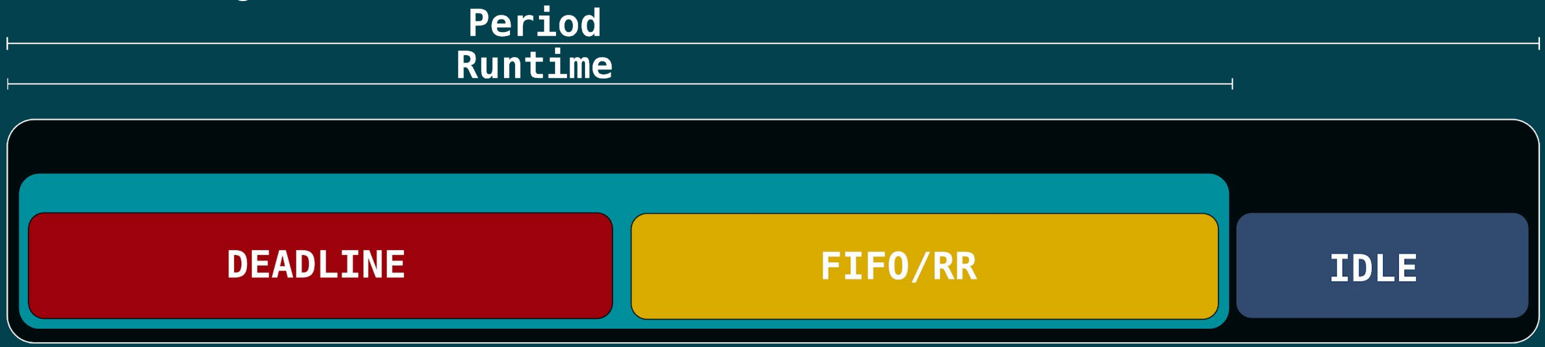
Everything works!

No?

What is the deal?

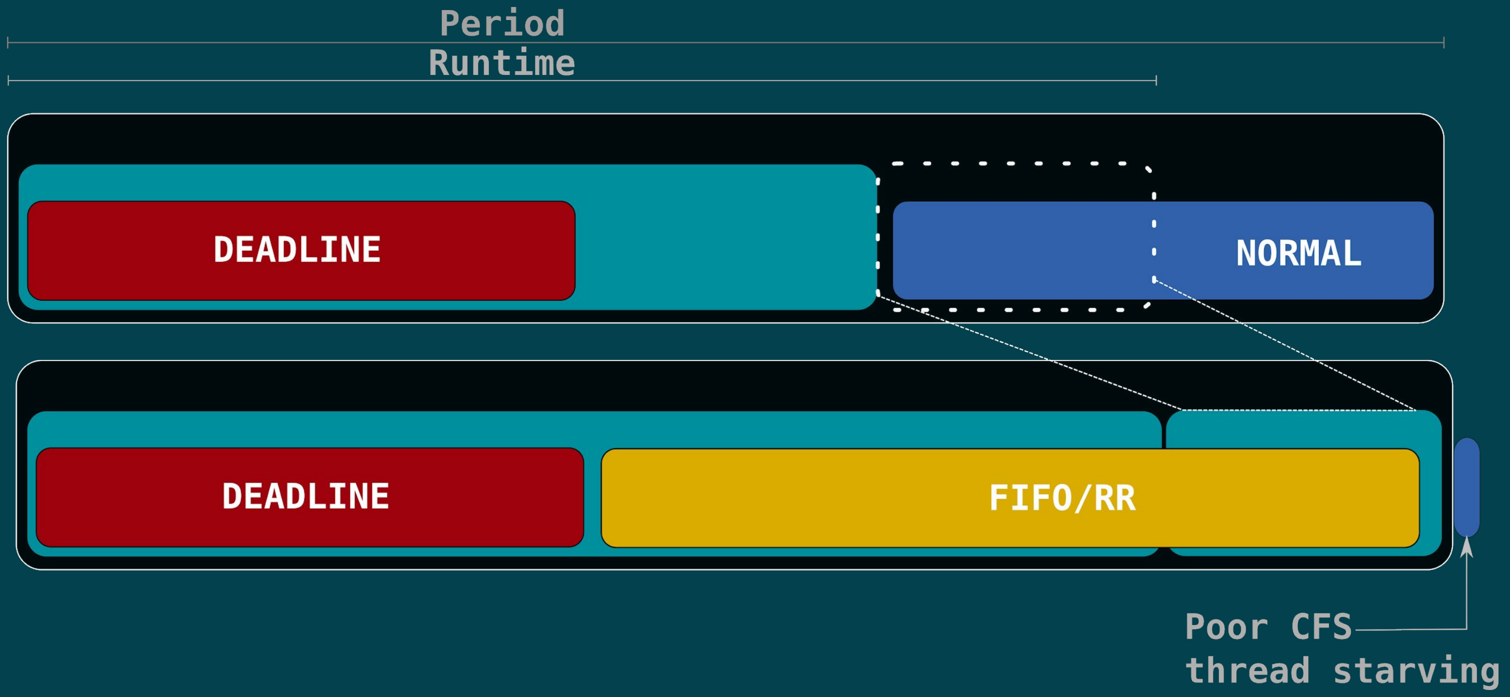
RT Throttling Pitfalls

- In the absence of normal tasks:
  - Single core or NO_RT_RUNTIME_SHARE



RT Throttling Pitfalls

- In the presence of per-cpu kernel threads:
  - RT_RUNTIME_SHARE

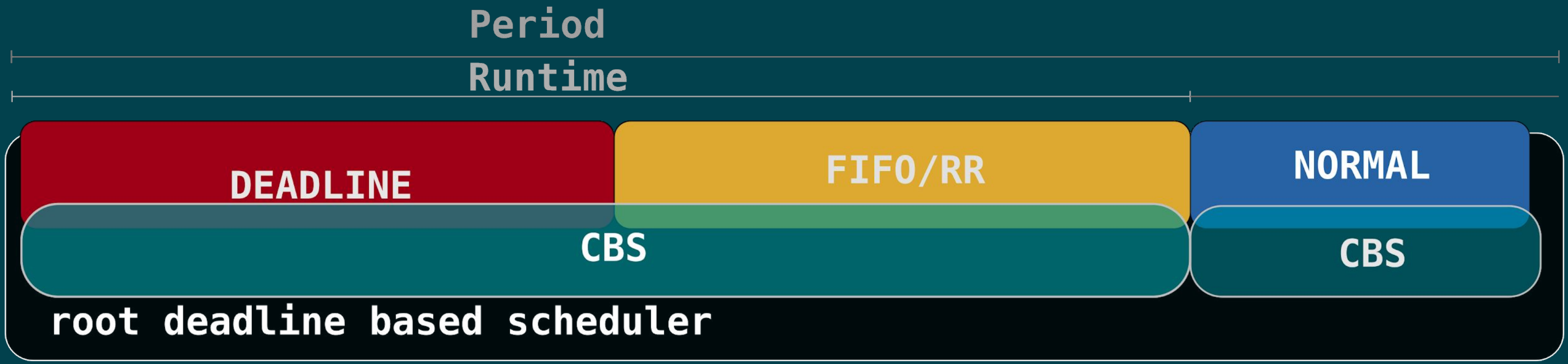


RT Throttling rework

- Change the way we implement RT Throttling
- Instead of throttling, provide bandwidth (a reservation) for RT and NON-RT schedulers:
  - RT/DL schedulers: 950/1000 ms
  - Non-rt schedulers: 50/1000 ms
  - Per-cpu schedulers (partitioned)
- Prioritize the servers according to the timing behavior

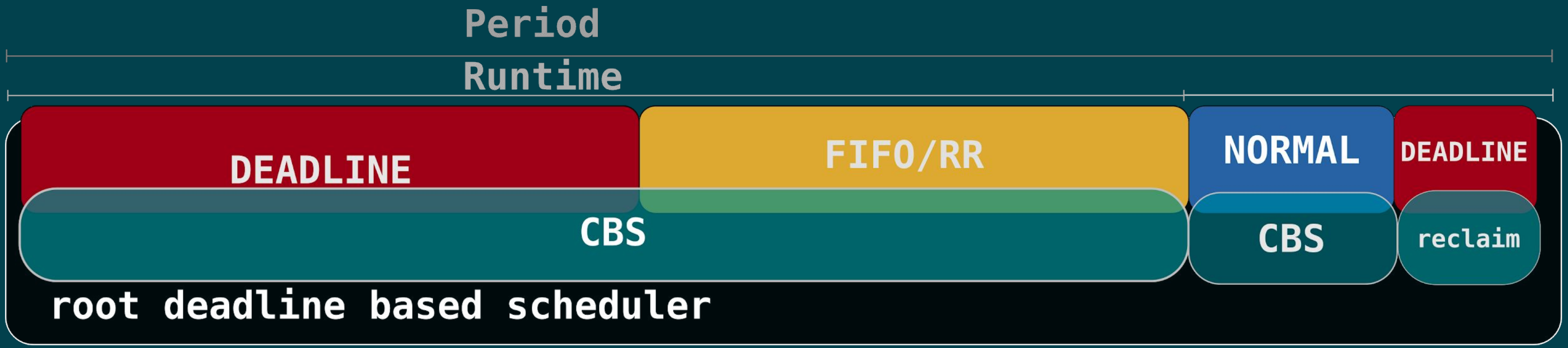
DL Server

- Suggestion from upstream is to have
  - A CBS Server scheduled for DL and RT (950ms/1000ms)
  - A CBS to normal (50ms/1000ms)
  - scheduling by the deadline:

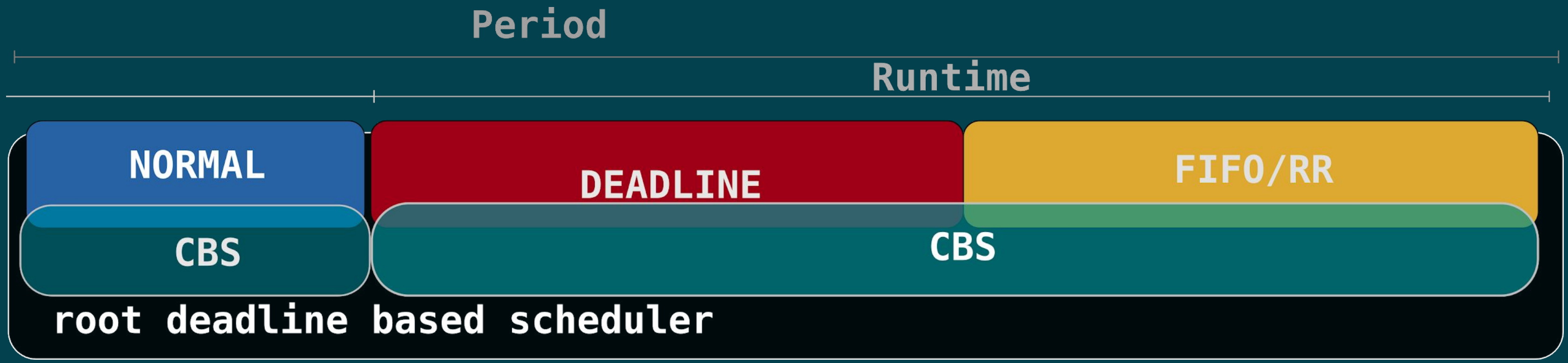


DL Server + Reclaiming

- We also need to implement reclaiming



Things are not that simple



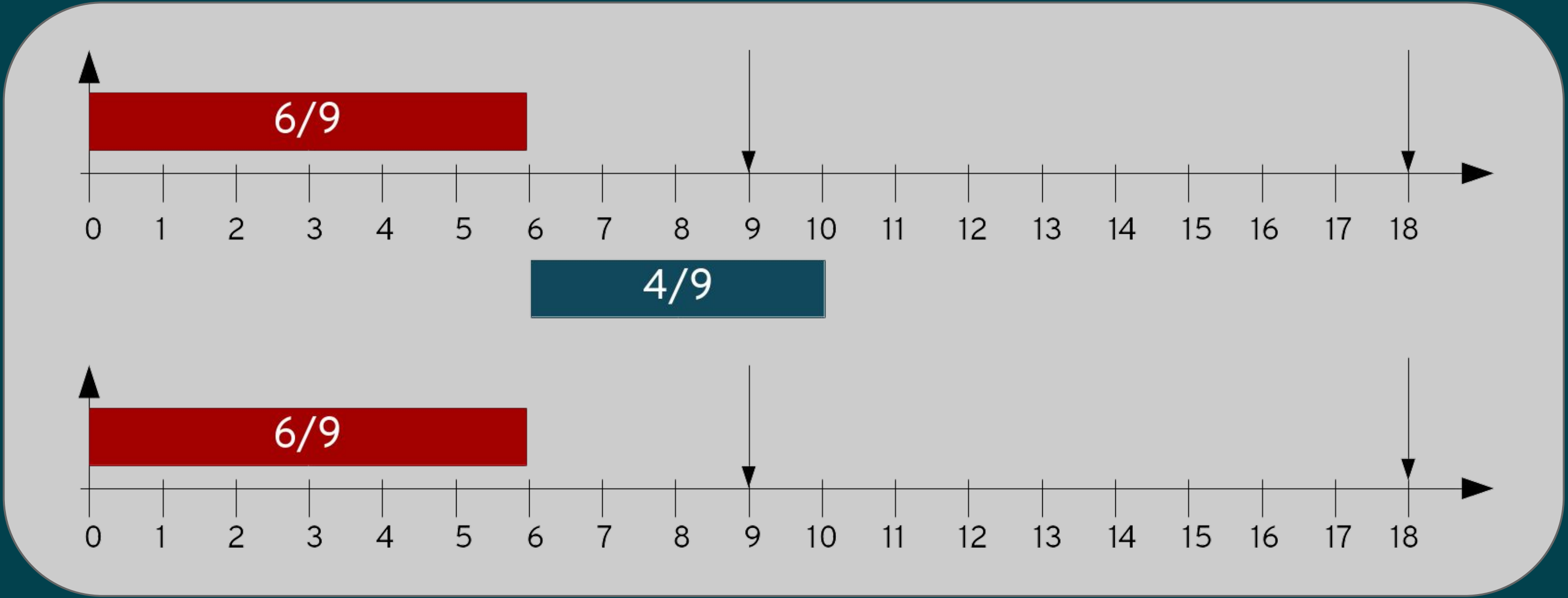
More studies required

- Pure SCHED_DEADLINE does not apply:
- GRUB also does not directly apply:
  - GRUB is fair:
    - Can cause the NORMAL reservation to use more than runtime/period in the presence of suspending RT tasks.
- Points to explore:
  - Use EDZL?
  - Put only RT tasks in the server, with reclaiming?

Schedulability improvements

The semi-partitioned scheduler

There are some cases in which a feasible task set is not scheduled by neither global or partitioned schedulers. For instance:

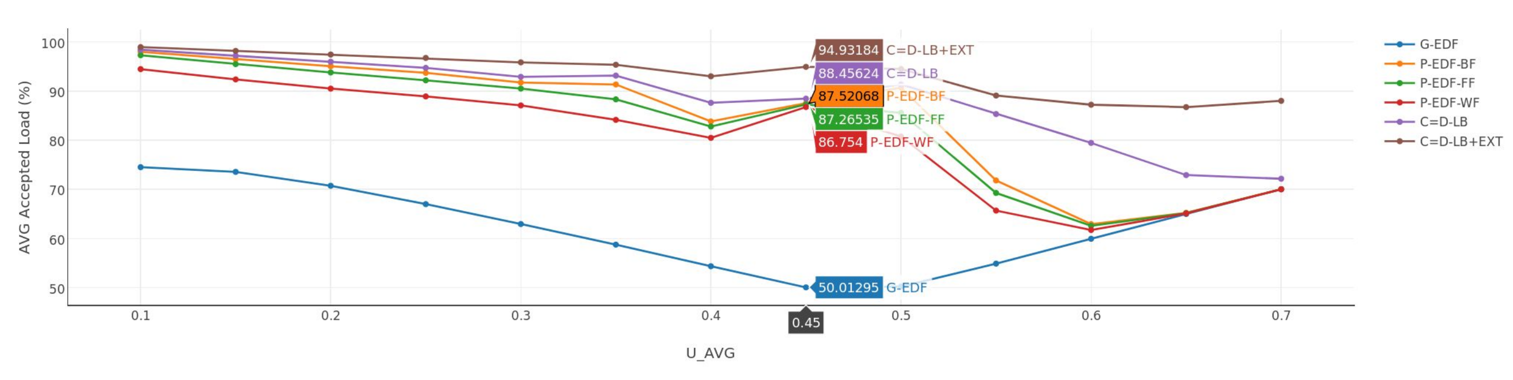


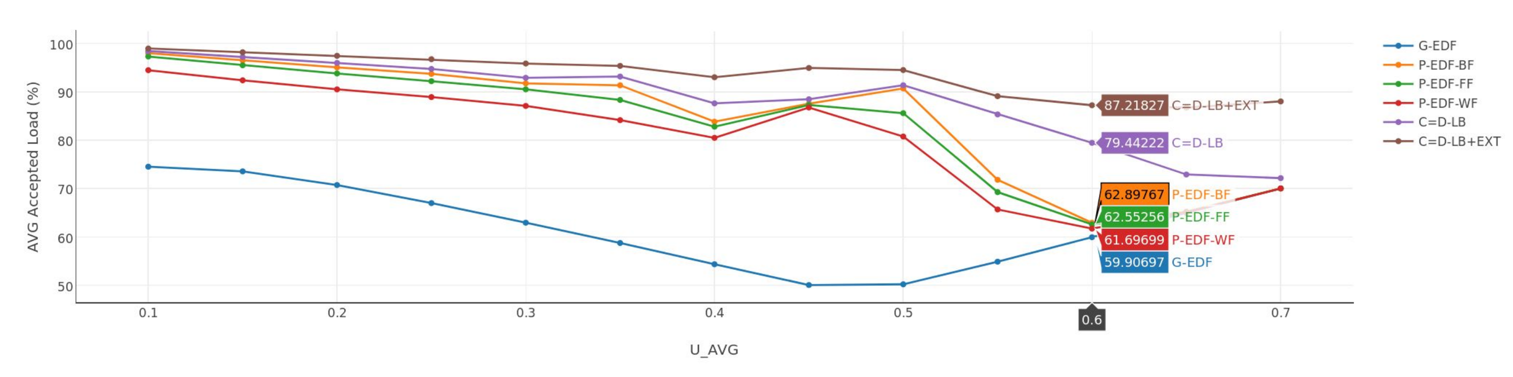
What does the academy have to say about it?

- B. Brandenburg and M. Gül, “Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations” shows that:
    - “usually ≥ 99% schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).”
    - This work, however, is not applicable for Linux because the workload is static

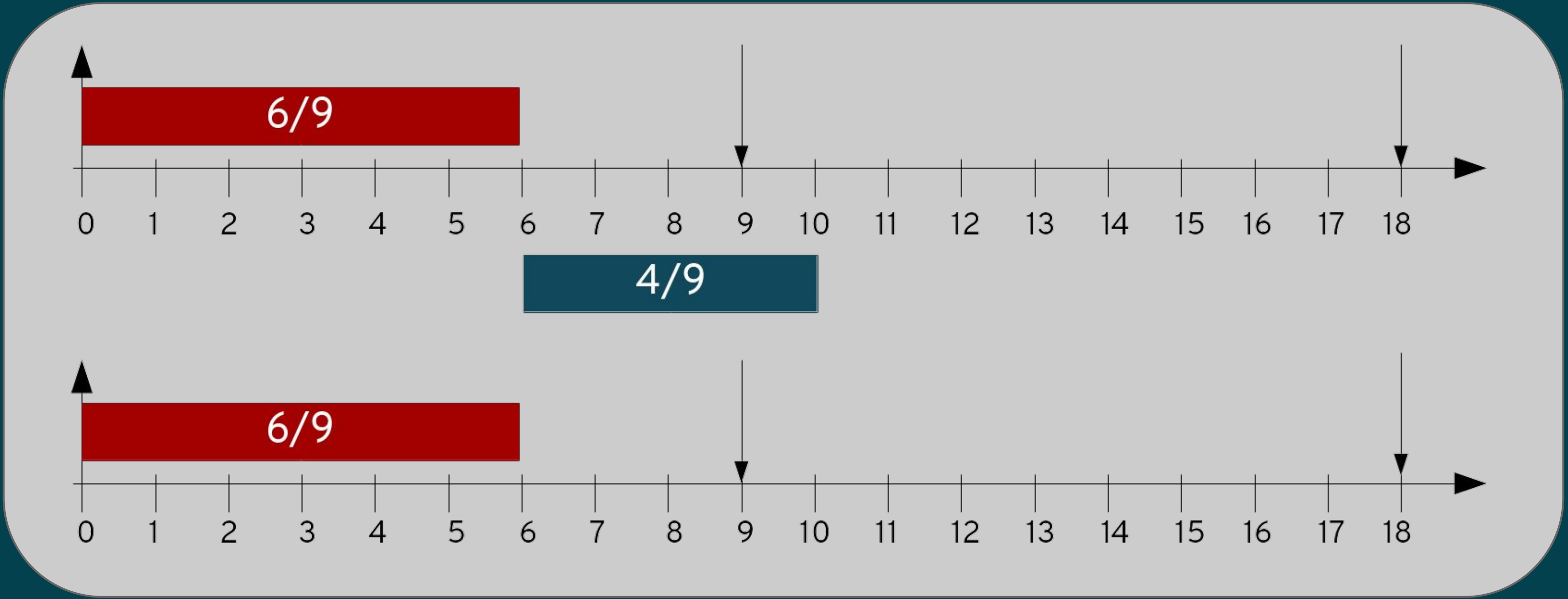
- D. Casini, A. Biondi, G. Buttazzo, “Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing.”
    - This paper relaxes the first, to be able to deal with dynamic workload.

How good is this online semi-partitioned scheduler?

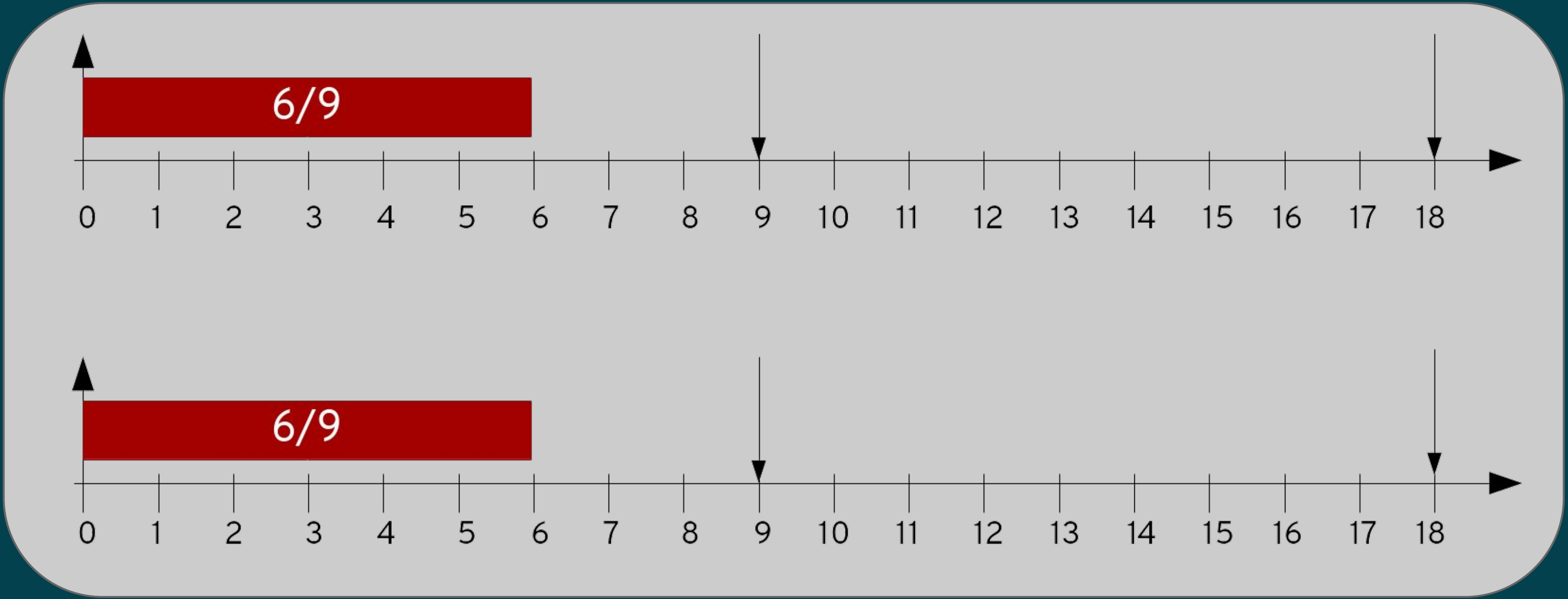




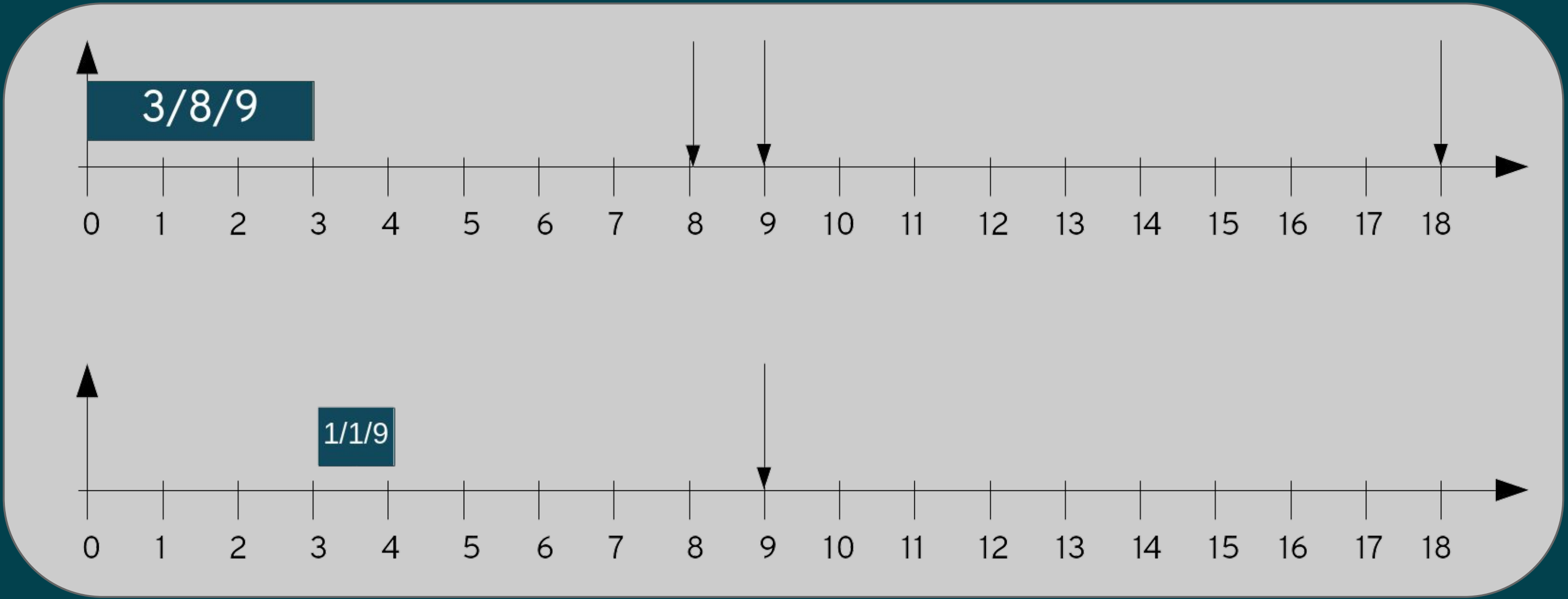
How does semi-partitioned place tasks?



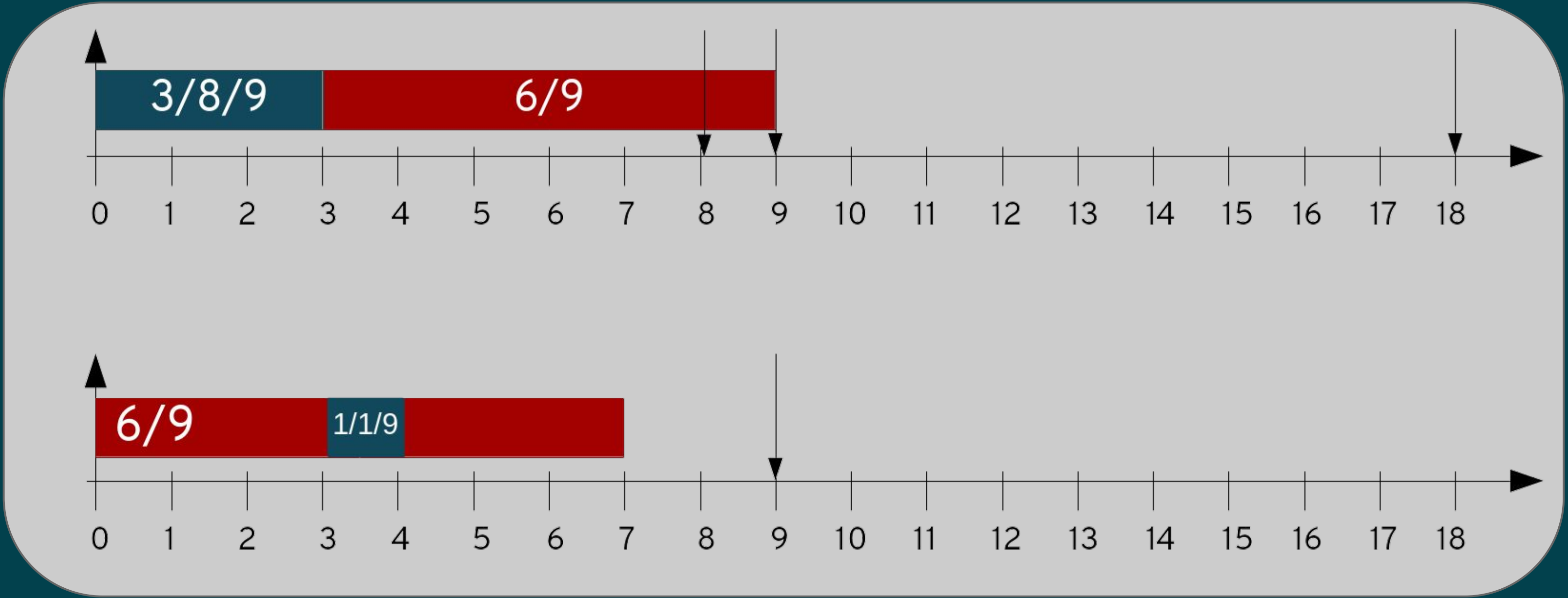
Pin as much task as possible



When it is not possible to pin, it splits a task.



Voilà!


3/8/9
6/9
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
6/9
1/1/9
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

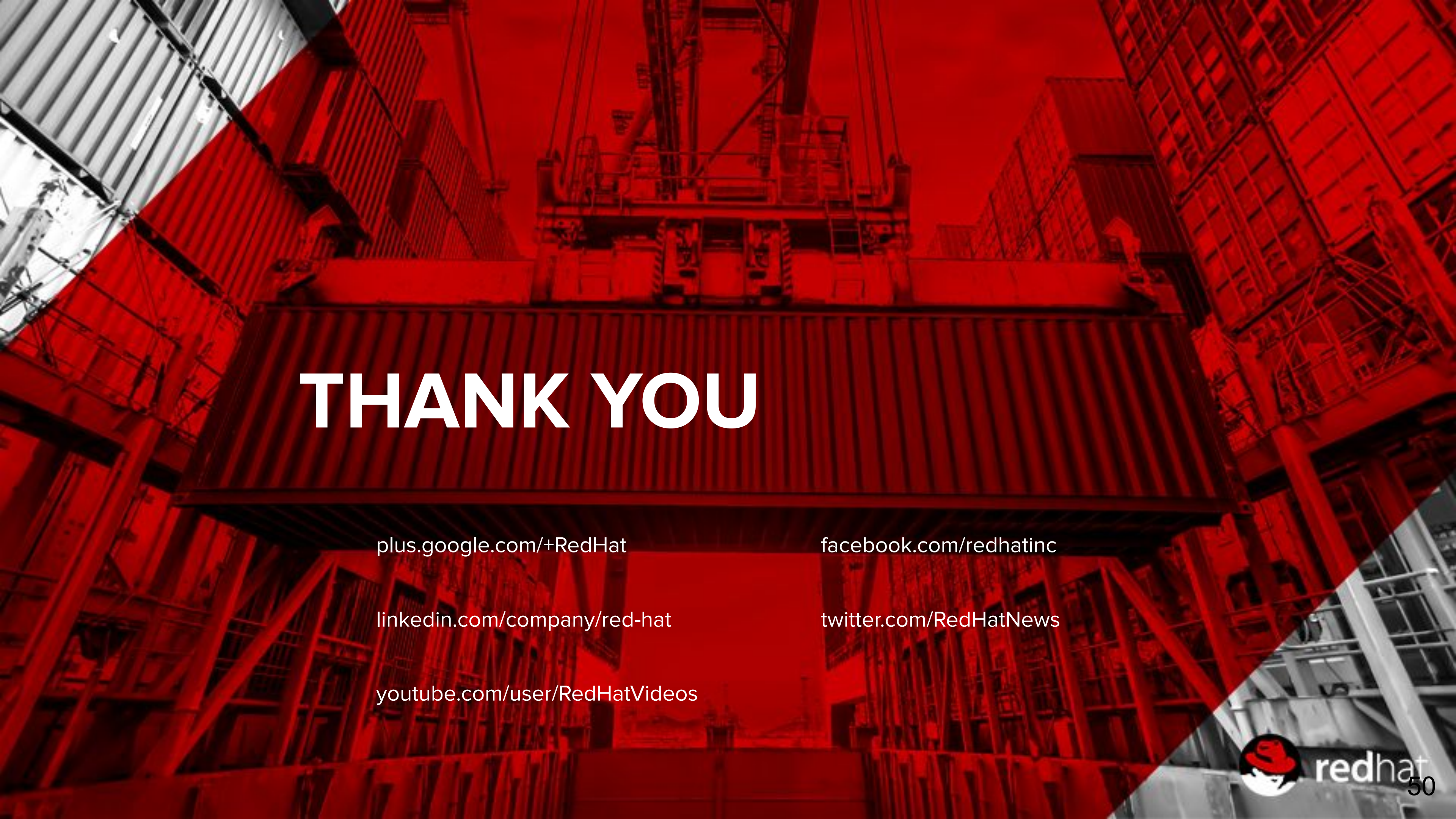
Semi-partitioned benefits

- Good points:
    - The majority of problems are reduced to single-core!
    - Less overhead:
        - The heuristics run only when setting attr/affinity/hotplug
        - There is no need to pull tasks, just push!
        - Migrations are bounded to M, for the system!
    - Tasks are mostly pinned to a single CPU!
    - Affinities come for FREE! YAY!

Semi-partitioned benefits

- Bad point:
    - Average response time is higher!
- Things we need to “think more”
    - The - real - admission control must to run in the kernel
    - The design of the scheduler considers implicit deadline - likewise the current… so.

Let us know what else you need!



THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos