Linux schedulers – overview continued

CFS, BFS, Deadline, MuQSS, etc.

What’s new in process scheduling?

Table of contents

- Linux schedulers – overview (continued)
  - CFS
  - BFS
  - Performance comparison
  - Deadline
  - MuQSS
- What’s new in process scheduling?
  - CPU Idle Loop
  - Thermal pressure
  - Energy Aware Scheduling (EAS)
  - Arm big.LITTLE CPU chip
  - Core scheduling

Completely Fair Scheduler
Ingo Molnar, April 2007

Ingo Molnar, April 2007

I wrote the first line of code of the CFS patch this week, 8am Wednesday morning, and released it to lkml 62 hours later, 22pm on Friday.

I’d like to give credit to Con Kolivas for the general approach here: he has proven via RSDL/SD that ‘fair scheduling’ is possible and that it results in better desktop scheduling. Kudos Con!

The CFS patch uses a completely different approach and implementation from RSDL/SD. My goal was to make CFS’s interactivity quality exceed that of RSDL/SD, which is a high standard to meet ☺

18 files changed, 1454 inserstions(+), 1133 deletions(-)

Schedulers: the plot thickens, https://lwn.net/Articles/230574/, J. Corbet, April 2007

The new scheduler which got into the game so abruptly caused a big storm on Linux mailing lists. Ultimately, CFS was incorporated into the **2.6.23 kerne**l, and Con Kolivas gave up work on the Linux kernel (temporarily).

Completely Fair Scheduler

- A completely new concept for the scheduler.
- Trying to implement **Processor Sharing** (there is no state in which one process has gotten more of the CPU than another, since all tasks are running on it at once, so all processes have a **fair share** of the CPU).
- On a real CPU **unfairness** will inevitably be **nonzero** when there are more tasks than CPUs. When one task is running on a CPU, this increases the amount of CPU time that the CPU **owes** to all other tasks. CFS schedules the task with the **largest unfairness** onto the CPU first.
- **Unfairness** is measured by the **virtual runtime**.
- In practice, the **virtual runtime** of a task is its **actual runtime normalized** to the total number of **running tasks**.
- Virtual time flows at a **priority-dependent** speed.
- Instead of queues of tasks – a **red-black tree**, sorted after the **virtual runtime** and the selection of the task, that run for the **shortest time** period.

Completely Fair Scheduler

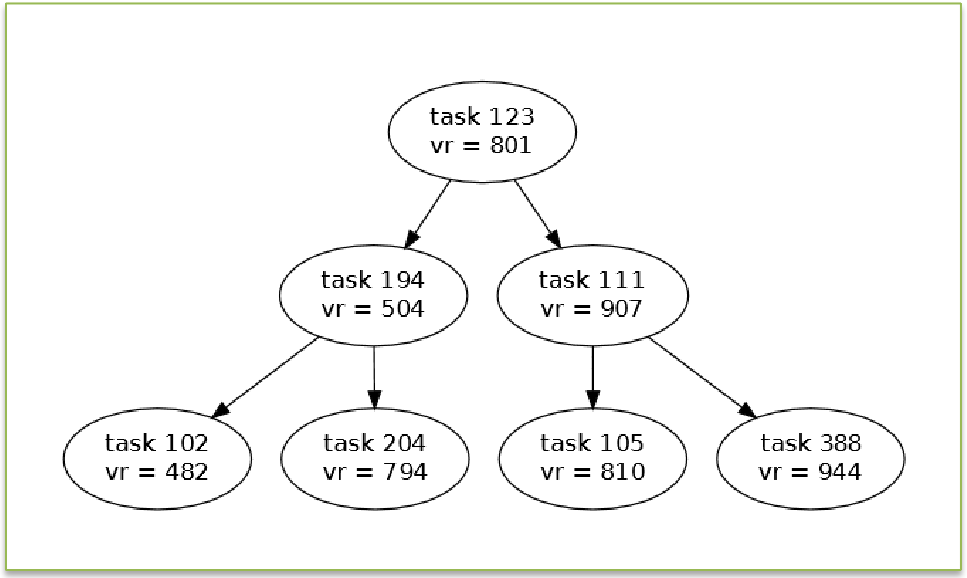
- It is also possible to group tasks and share processor time fairly among defined „**entities**” – **process groups**.
- When implementing CFS, the code was reorganized to separate sections responsible for the scheduling policy (the **struct sched_class** has been created, there is a pointer to this structure in **struct task_struct**).
- Like the O(1) scheduler, CFS maintains **separate data structures for each CPU**. This reduces the wait for the lock to be removed, but requires explicit processor load balancing.
- When a new task is created, it is assigned the minimum current vruntime (**min_vruntime**).

- With CFS, tasks have no concept of timeslice but rather run until they are no longer the most unfairly treated task. To reduce context switching overhead, CFS divides time into a minimum granularity.

```
struct task_struct  {
    …
    const struct sched_class *sched_class;
    …
}
```

Completely Fair Scheduler

Red-black tree
for CFS scheduler
process selection **O(1)** insertion
O(log(n))



- When a task has finished running on the CPU, all of the other tasks in the tree need to have their **unfairness** increase.
- To prevent having to update **all** of the tasks in the tree the scheduler maintains a per-task **vruntime** statistic.
- This is the **amount of total nanoseconds** that the task has spent **running** on a **CPU weighted** by its **niceness**.
- Thus, instead of updating all other tasks to be more **unfair** when a task has **finished** running on the CPU, we update the **leaving** task to be more fair than others by increasing its **virtual runtime**.
- The scheduler always selects the **most unfairly treated** task by selecting the task with the **lowest vruntime**.

http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

Completely Fair Scheduler

sched_latency_ns – the time when one era should take place, i.e. all tasks from the queue should be completed. The default is 20 ms.

sched_min_granularity_ns – minimum time, which on average should be given to a task in an era. The default is 4 ms.

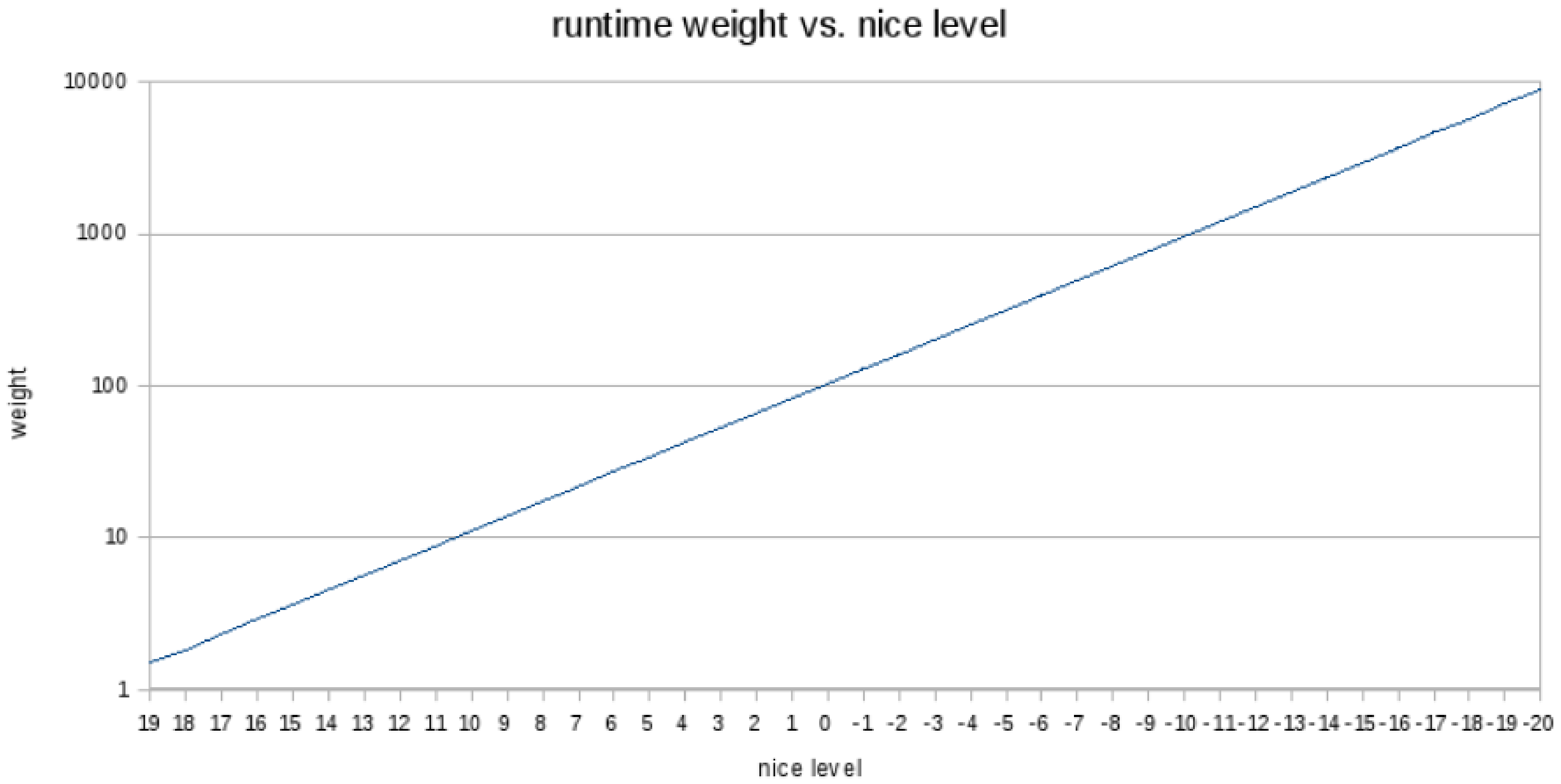
epoch = max(sched_latency_ns; sched_min_granularity_ns * nr_running)
ideal_slice = epoch * (weight/∑weight)

Variable quantum lengths, but a fixed period of rotation of the era, guarantees small delays. Under heavy load, the quanta are not shortened below the minimum value, at the expense of responsiveness.

Processes with different **priorities** receive different **weights**. The virtual time is scaled with these weights – the scheduler takes into account the differences in *nice* values of processes. Priority weights are allocated as geometric progression:
prio_to_weight[n] ≈ prio_to_weight[n+1] * 1.25
Processes with a given priority difference will always receive CPU time in a constant proportion.

Completely Fair Scheduler

weights of processes



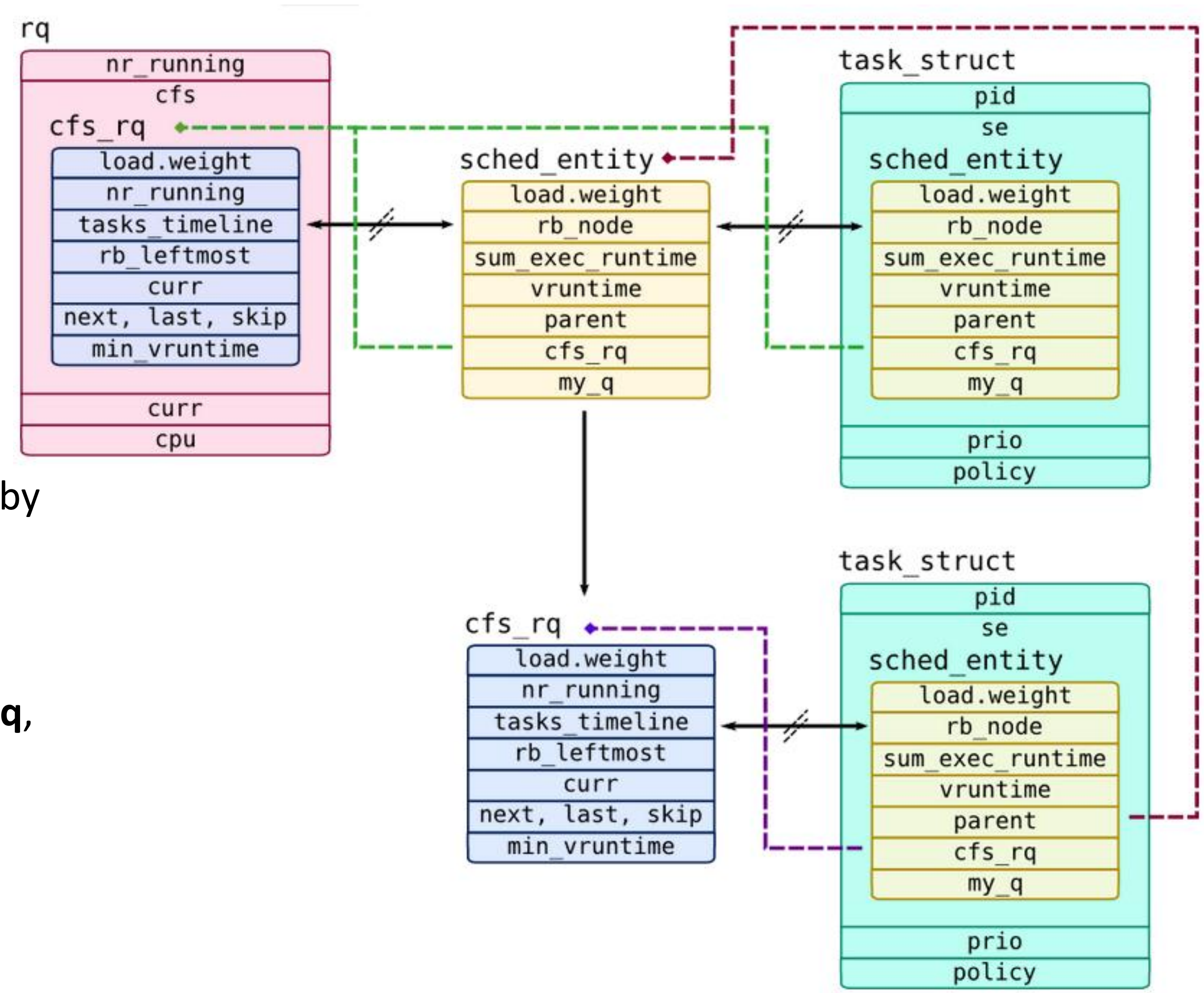
A process with *nice* -20 will get about 6000 times more CPU time than a process with *nice* 19 (for comparison: 160 time more than in the old scheduler)

CFS scheduler doesn’t deal with **tasks**, but with **scheduler entities** of type **struct sched_entity**.

Sched entity may represent a **task** or a **queue** of **entities** of type **struct cfs_rq** (which is referenced by field **my_q**), thus allowing to build hierarchies of entities and allocate resources to task groups (**Cgroups)**.

Processor run queue, represented by type **struct rq**, contains field **cfs** which is instance of **struct cfs_rq** and contains queue of all high-level entities.

Each entity has **cfs_rq** pointer which points to CFS **runqueue** to which that entity belongs.



In this example processor run queue has two **scheduler entities**: one **CFS queue** with single task (which refers to top-level cfs_rq through parent pointer) in it and one **top-level task**.

Source: https://myaut.github.io/dtrace-stap-book/kernel/sched.html

Completely Fair Scheduler

SKIP

- CFS doesn't allocate timeslices. Instead it accounts total time which task had spend on CPU and saves it to **sum_exec_runtime** field.
- When task is dispatched onto CPU, its **sum_exec_runtime** is saved into **prev_sum_exec_runtime**, so calculating their difference will give time period that task spent on CPU since last dispatch. **sum_exec_runtime** is expressed in nanoseconds but it is not directly used to measure task's runtime.
- To implement priorities, CFS uses task **weight** (in field **load.weight**) and divides runtime by tasks weight, so tasks with higher weights will advance their runtime meter (saved into **vruntime** field) slower.
- Tasks are sorted according to their **vruntime** in a red-black tree called **tasks_timeline**, while left-most task which has lowest **vruntime** of all tasks and saved into **rb_leftmost**.
- CFS has special case for tasks that have been woken up. Because they can be sleeping too long, their **vruntime** may be too low and they will get unfairly high amount of CPU time. To prevent this, CFS keeps **minimum** possible **vruntime** of all tasks in **min_vruntime** field, so all waking up tasks will get **min_vruntime** minus a predefined "**timeslice**" value.
- CFS also have a **scheduler buddies** – helper pointers for a dispatcher:
  - **next** – task that was recently awoken,
  - **last** – task that recently was evicted from CPU and
  - **skip** – task that called **sched_yield()** giving CPU to other entities.

Source: https://myaut.github.io/dtrace-stap-book/kernel/sched.html

Completely Fair Scheduler

easy exchange of schedulers or modularization?

Con Kolivas accused CFS of having used his scheduler modularization ideas included in the **plugsched** framework (enabling easy exchange of schedulers), although they were previously criticized.

> Linus Torwalds (https://yarchive.net/comp/linux/security.html)
>
> The arguments that 'servers' have a different profile than 'desktop' is pure and utter garbage, and is perpetuated by people who don't know what they are talking about (...) Yes, there are differences in tuning, but those have nothing to do with the basic algorithm. They have to do with goals and trade-offs, and most of the time we should aim for those things to **auto-tune**.

Ingo refuted Con's allegations that his proposed reorganization of the scheduler code was not intended to allow the interchangeability of schedulers, but only to improve the quality of the code.

Con Kolivas returns in 2009
in FAQ about BFS

Why „Brain Fuck”?

Because ...

... it throws out everything about what we know is good about how to design a modern scheduler in scalability.

... it's so ridiculously simple.

... it performs so ridiculously well on what it's good at despite being that simple.

... it's designed in such a way that mainline would never be interested in adopting it, which is how I like it.

... it will make people sit up and take notice of where the problems are in the current design.

... it **throws out the philosophy that one scheduler fits all** and shows that you can do a -lot- better with a **scheduler designed for a particular purpose**. I don't want to use a steamroller to crack nuts.

... it actually means that more CPUs means better latencies.

... I must be fucked in the head to be working on this again.

I'll think of some more becauses later.

Con Kolivas – on lwn.net (and wiki) about BFS

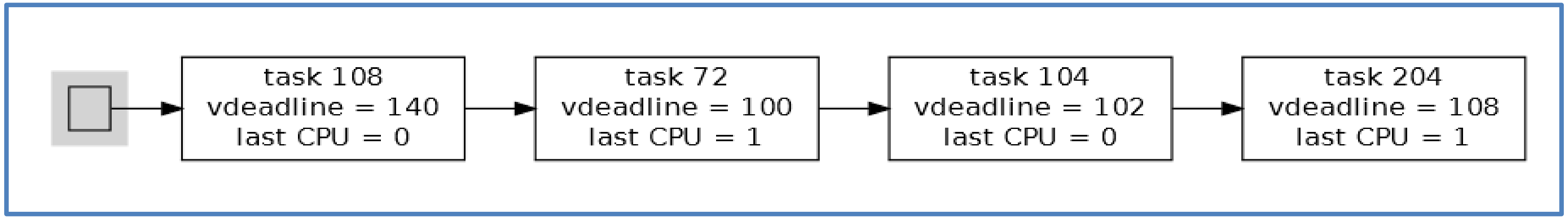
The main focus of BFS is to achieve excellent desktop **interactivity** and **responsiveness** without heuristics and tuning knobs that are difficult to understand, impossible to model and predict the effect of, and when tuned to one workload cause massive detriment to another.

BFS is best described as a **single runqueue**, O(n) lookup, **earliest effective virtual deadline first** design.

The reason for going back to a single runqueue design is that once **multiple runqueues** are introduced, per-CPU or otherwise, there will be **complex interactions** as each runqueue will be responsible for the scheduling latency and fairness of the tasks only on its own runqueue, and to achieve **fairness** and **low latency** across **multiple CPUs**, any advantage in throughput of having CPU local tasks causes other disadvantages.

A significant feature of BFS is that all accounting is done purely based on **CPU used** and **nowhere is sleep time used in any way** to determine entitlement or interactivity.

Con Kolivas – on lwn.net (i na wiki) about BFS



task 108
vdeadline = 140
last CPU = 0

task 72
vdeadline = 100
last CPU = 1

task 104
vdeadline = 102
last CPU = 0

task 204
vdeadline = 108
last CPU = 1

The task put into the queue receives a time quantum of **rr_interval** and **deadline**

jiffies + (prio_ratio * rr_interval)

where **prio_ratio**, like the weights in CFS, depends geometrically on the priority.

If the calculated deadline is earlier than the deadline of the process performed on one of the processors, then the new process immediately preempts it.

The system selects the task by reviewing the entire list of tasks in **O(n)** (!). If it encounters a task with an **expired deadline**, it immediately runs it. Otherwise, the one with the **nearest deadline** begins to run.

Simple additional mechanisms that favor tasks on the same processor as they were previously run.

Only two configuration parameters:

rr_interval – time quantum, default 6 ms,

iso_cpu – percentage of processor time that user processes simulating RT tasks can take up at maximum, default 70%.

Charts removed, conclusions left

Comparing performance of Linux schedulers

Experiments by Ingo Molnar (2009-09-06)

- First and foremost, let me say that I'm happy that you are hacking the Linux scheduler again.
- **It's perhaps proof that hacking the scheduler is one of the most addictive things on the planet** ;-)
- I understand that BFS is still early code and that you are not targeting BFS for mainline inclusion – but BFS is an interesting and bold new approach, cutting a _lot_ of code out of kernel/sched*.c, so it raised my curiosity and interest :-)
- The testbox i picked fits into the upper portion of what I consider a sane range of systems to tune for – and should still fit into BFS's design bracket as well according to your description: it's a dual quad core system with hyperthreading.

http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)

- Alas, as it can be seen in the graphs, I can not see **any BFS performance improvements**, on this box.
- In the **kbuild** test BFS is showing significant weaknesses up to 16 CPUs.
- BFS performed very poorly in the **pipe** test: at 8 pairs of tasks it had a runtime of 45.42 seconds - while sched-devel finished them in 3.8 seconds.
- **Messaging**: mainline sched-devel is significantly faster for smaller and larger loads as well. With 20 groups mainline ran 61.5% faster.
- **OLTP peformance**: for sysbench OLTP performance sched-devel outperforms BFS on each of the main stages.
- General interactivity of BFS seemed good to me - except for the pipe test when there was significant lag over a minute. I think it's some starvation bug, not an inherent design property of BFS, so I'm looking forward to re-test it with the fix.
- I agree with the general goals described by you in the BFS announcement - **small desktop systems matter more than large systems**. We find it critically important that the mainline Linux scheduler performs well on those systems too.

http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Con replies (7 September 2009) ...

Hard to keep a project under wraps and get an audience at the same time, it is. I do realise it was inevitable LKML would invade my personal space no matter how much I didn't want it to, but it would be rude of me to not respond.

/me sees Ingo run off to find the right combination of hardware and benchmark to prove his point.

[snip lots of bullshit meaningless benchmarks showing how great cfs is and/or how bad bfs is, along with telling people they should use these artificial benchmarks to determine how good it is, demonstrating yet again why benchmarks fail the desktop]

I'm not interested in a long protracted discussion about this since I'm too busy to live linux the way full time developers do, so I'll keep it short, and perhaps you'll understand my intent better if the FAQ wasn't clear enough.

Do you know what a normal desktop PC looks like? No, a more realistic question based on what you chose to benchmark to prove your point would be: Do you know what normal people actually do on them?

Feel free to treat the question as rhetorical.

Regards, -ck

/me checks on his distributed computing client's progress, fires up his next H264 encode, changes music tracks and prepares to have his arse whooped on quake live.

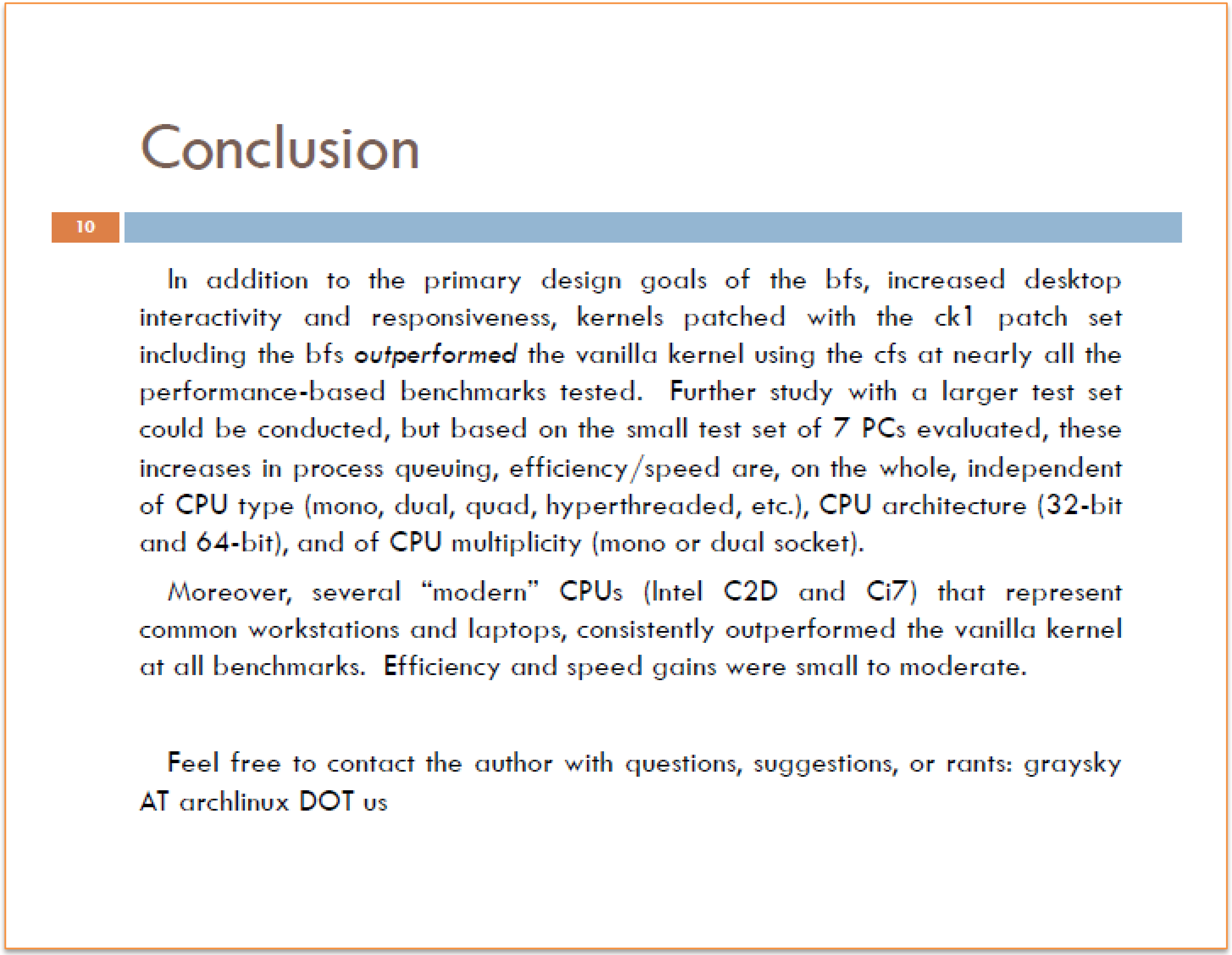
https://lwn.net/Articles/351504/

T. Groves, J. Knockel, E. Schulte Conclusions

- The results indicate that CFS outperformed BFS with minimizing turnaround time but that BFS outperformed CFS for minimizing latency. This indicates that BFS is better for interactive tasks that block on I/O or user input and that CFS is better for batch processing that is CPU bound.
- Many distros like Ubuntu already have separate kernel packages for desktops and servers optimized for those common use cases. To improve the average desktop experience, distros could patch their kernel to use the BFS scheduler. If desktop users do perform a lot of batch processing, **distros could provide two different kernel packages alternatives**.

http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

CPU schedulers compared – Graysky, 2012



Conclusion

10

In addition to the primary design goals of the bfs, increased desktop interactivity and responsiveness, kernels patched with the ck1 patch set including the bfs ***outperformed*** the vanilla kernel using the cfs at nearly all the performance-based benchmarks tested. Further study with a larger test set could be conducted, but based on the small test set of 7 PCs evaluated, these increases in process queuing, efficiency/speed are, on the whole, independent of CPU type (mono, dual, quad, hyperthreaded, etc.), CPU architecture (32-bit and 64-bit), and of CPU multiplicity (mono or dual socket).

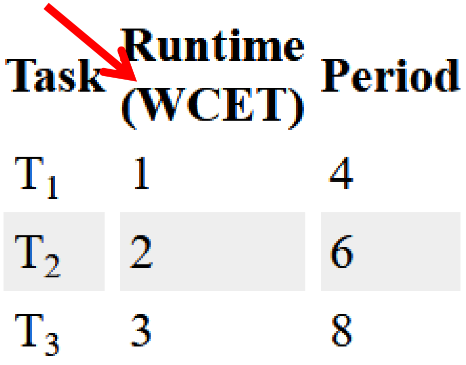
Moreover, several "modern" CPUs (Intel C2D and Ci7) that represent common workstations and laptops, consistently outperformed the vanilla kernel at all benchmarks. Efficiency and speed gains were small to moderate.

Feel free to contact the author with questions, suggestions, or rants: graysky AT archlinux DOT us

http://repo-ck.com/bench/cpu_schedulers_compared.pdf

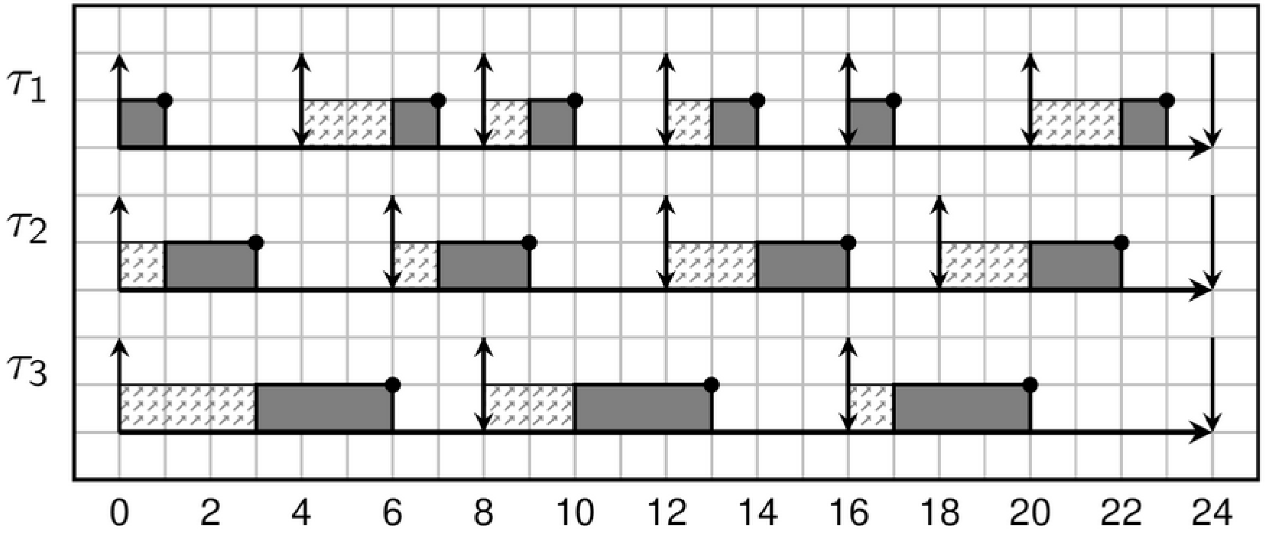
The Deadline scheduler

Worst-case execution time



Task	Runtime (WCET)	Period
T_1	1	4
T_2	2	6
T_3	3	8

https://lwn.net/Articles/743740/



it is not possible to use a fixed-priority scheduler to schedule this task set while meeting every deadline; regardless of the assignment of priorities, one task will not run in time to get its work done.

Deadline scheduling gets away with the notion of process priorities. Instead, processes provide three parameters: runtime, period, and deadline. A SCHED_DEADLINE task is guaranteed to receive "**runtime**" microseconds of execution time every "**period**" microseconds, and these "runtime" microseconds are available within "**deadline**" microseconds from the beginning of the period. The task scheduler uses that information to run the process with the **earliest deadline first** (EDF).

The Deadline scheduler

- /Documentation/echeduler/sched-deadline.rst.
- SCHED_DEADLINE in Wikipedia.
- Deadline scheduling: coming soon?, Jonathan Corbet, December 2013.
- Deadline scheduling in the Linux kernel, J.Lelli, C. Scordino, L. Abeni, D.Faggioli, Software – Practice& Experience, vol. 46 (6), June 2016.

  Container-Based Real-Time Scheduling in the Linux Kernel, L. Abeni, A. Balsini, T. Cucinotta, EWiLi'18, October 4th, 2018.

- Deadline scheduling part 1 — overview and theory, D. Oliveira, 2018.
- Deadline scheduler part 2 — details and usage, D.Oliveira, 2018.
- https://lwn.net/Kernel/Index/#Realtime-Deadline_scheduling.
- SCHED_DEADLINE desiderata and slightly crazy ideas, D. Oliveira, J. Lelli, DevConf.cz, January 2020.
- Capacity awareness for the deadline scheduler, M. Rybczyńska, May 2020.

  The current implementation of the deadline scheduler does not work well on asymmetric CPU configurations like Arm's big.LITTLE. Dietmar Eggemann posted a patch set to address this problem by adding the notion of **CPU capacity** (the number of instructions that can be executed in a given time) and taking it into account in the admission-control and task-placement algorithms.

The MuQSS CPU scheduler

- Based on https://lwn.net/Articles/720227/ by Nur Hussein, April 2017.
- Original message from Con Kolivas : http://ck-hack.blogspot.com/2016/10/muqss-multiple-queue-skiplist-scheduler.html announcing patch for Linux 4.7 (October 2016).

- **MuQSS – The Multiple Queue Skiplist Scheduler** (pronounced *mux*)
- The main goal is to tackle 2 major **scalability limitations** in BFS:
  - The **single runqueue** which means all CPUs would fight for lock contention over the one runqueue (problems start when the **number of CPUs increases** beyond 16),
  - The **O(n) look up** which means linear increase in overhead for task lookups as **number of processes increases**. Also, iterating over a linked list led to cache-thrashing behavior.

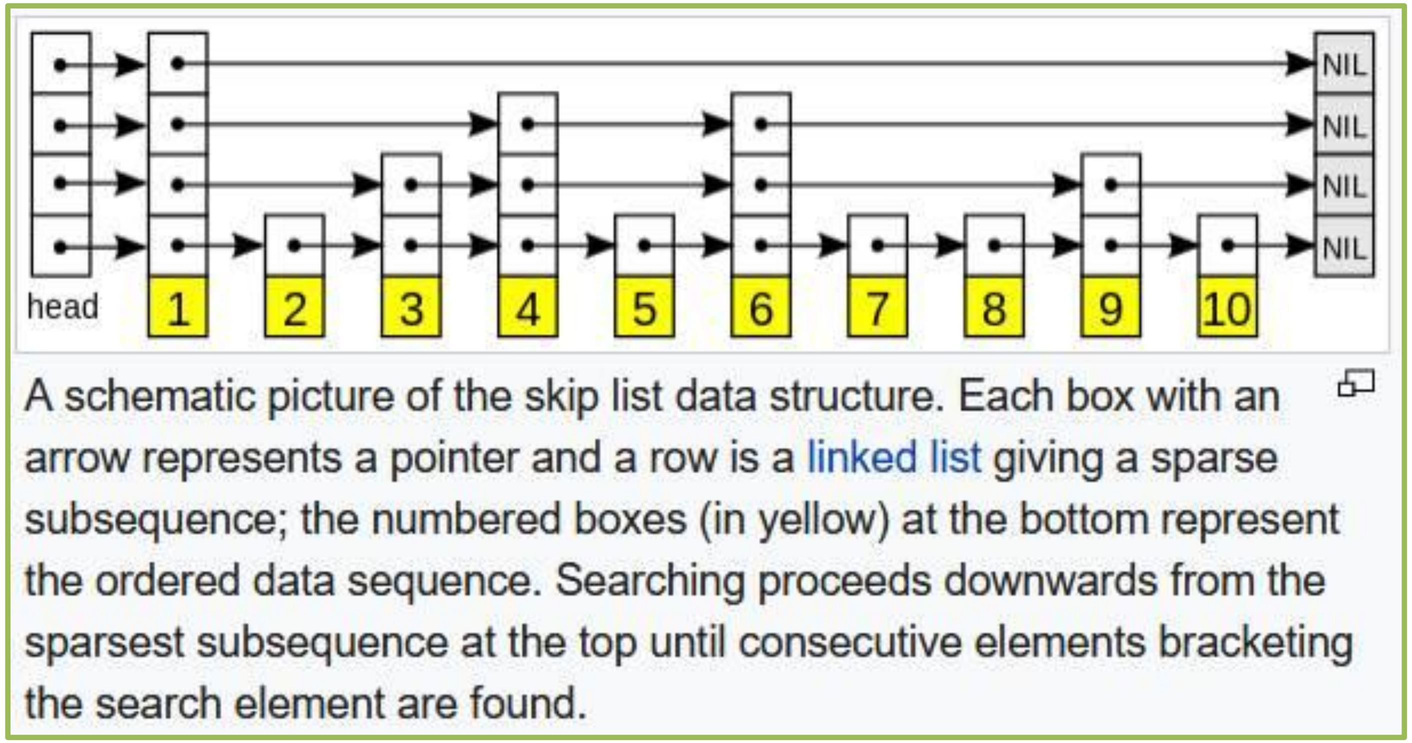
The MuQSS CPU scheduler

MuQSS is BFS with **multiple run queues, one per CPU**.

The queues have been implemented as **skip lists**.

Wikipedia, https://en.wikipedia.org/wiki/Skip_list

Skip list is a data structure that allows **O(log n) search** complexity as well as **O(log n) insertion** complexity within an ordered sequence of **n** elements.



A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes (in yellow) at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

The MuQSS CPU scheduler

Virtual deadline is calculated as in BFS, **niffies** are used instead of **jiffies** (nanosecond-resolution monotonic counter)

virtual_deadline = niffies + (prio_ratio * rr_interval)

The scheduler can find the next eligible task to run in **O(1)**, insertion is done in **O(log n)**.

The scheduler will use a **non-blocking** "trylock" attempt when popping the chosen task from the relevant run queue, but will move on to the next-nearest deadline on another queue if it fails to acquire the queue lock (no lock contention among different CPU queues).

Only 3 configuration parameters:

rr_interval – CPU quantum, which defaults to 6 ms,

interactive – a tunable to toggle the deadline behavior. If disabled, searching for the next task to run is done independently on each CPU, instead of across all CPUs.

iso_cpu – percentage of CPU time, across a rolling five-second average, that isochronous tasks (SCHED_ISO) will be allowed to use.

Introduction – A short story of sched_ext

- Patchset v1: 2022-11-30, patchset v2: 2023-01-27
- Kernel Report 2023: The extensible scheduler class (write complete CPU schedulers in BPF)
  - It allows users to write a custom scheduling policy using BPF without modifying the kernel code.
  - BPF provides a safe kernel programming environment.
  - BPF verifier ensures that your custom scheduler has neither a memory bug nor an infinite loop.
  - BPF scheduler can be updated without reinstalling the kernel and rebooting a server.
  - Developed by engineers from **Meta** and **Google**.
  - Why: easy experimentation, faster scheduler development, ad hoc schedulers for special workloads.
  - Why ***not***: added mainteance burden, benchmark gaming, vendors may require specific schedulers, ABI concerns, redirection of work on core scheduler.
  - Rejected by scheduler maintainer (**Peter Zijlstra**).
- September 30, 2024: Linus has released 6.12-rc1 – sched_ext got merged.
- https://github.com/sched-ext/scx/

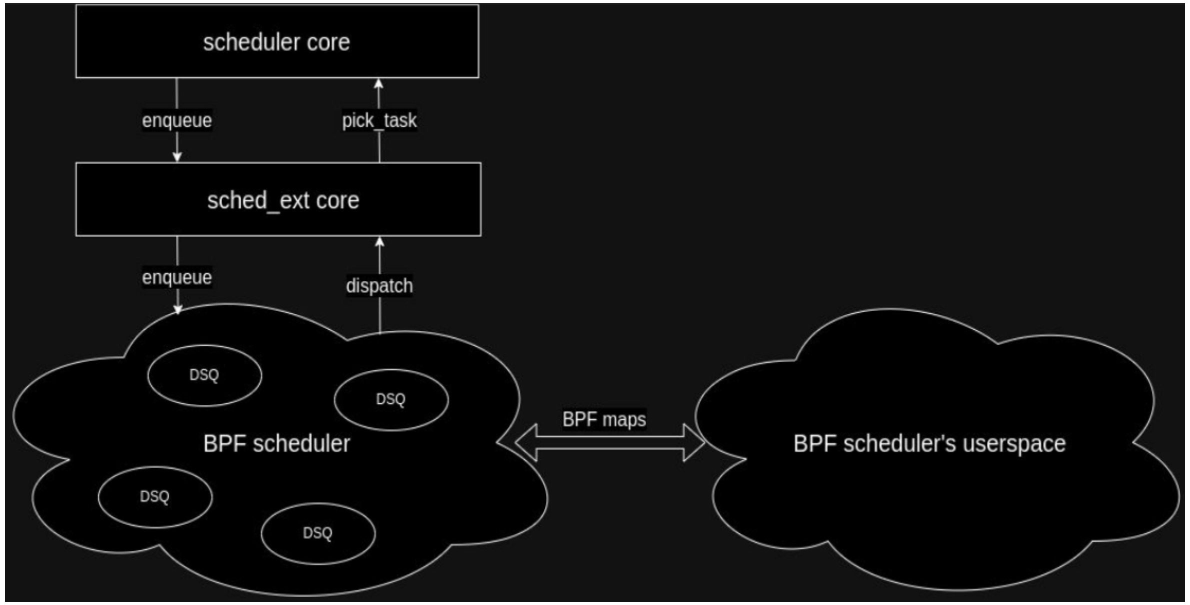
David Vernet

Peter Zijlstra

Changwoo Min's introduction to the sched_ext scheduling class

Sched_ext: pluggable scheduling using BPF

- A new scheduling class, called **SCHED_EXT**, that can be selected with a sched_setscheduler() call like most others.
- SCHED_EXT is placed between the idle class (SCHED_IDLE) and the completely fair scheduler (SCHED_NORMAL) in the priority stack. As a result, no SCHED_EXT scheduler can take over the system.
- The BPF-written scheduler is global to the system as a whole.
- If there is no BPF scheduler loaded, then any processes that have been put into the SCHED_EXT class will be run as if they were in SCHED_NORMAL.
- Once a BPF scheduler is loaded, it will take over the responsibility for all SCHED_EXT tasks.
- A BPF program implementing a scheduler will manage a set of dispatch queues, each of which may contain runnable tasks that are waiting for a CPU to execute on.
- The BPF side of the scheduler is mostly implemented as a set of callbacks to be invoked via an operations structure, each of which informs the BPF code of an event or a decision that needs to be made.



Sched_ext: pluggable scheduling using BPF

- Sched_ext: pluggable scheduling in the Linux kernel, Kernel Recepies, Oct. 5, 2023, David Vernet.
  - Scheduling is a notoriously difficult problem. An effective scheduler should fully utilize a system, while also optimizing for cache locality, while also accounting for real time constraints, while also accounting for battery life and power management, while also ensuring fairness, etc.
  - The landscape of the tech industry has changed a lot in the last 15 years. Back in the late 2000s, cores were typically homogeneous, and were spaced further apart from one another. Modern systems are by comparison much more complex. Heterogeneous architectures are the norm for mobile devices, and are becoming more common in x86. Cache hierarchies are also less uniform, with Core Complex (CCX) chips having multiple shared L3 caches within a single socket. Use cases have evolved as well. Applications such as mobile and VR have latency requirements to avoid missing deadlines that impact user experience, and stacking workloads in data centers is constantly pushing the demands on the scheduler in terms of workload isolation and resource distribution.
  - While CFS is a great scheduler, there are opportunities to continue to improve it for such use cases. With sched_ext, we can easily experiment and find scheduling algorithms that address these use cases by allowing developers to implement scheduling policies in BPF programs.

Sched_ext: pluggable scheduling using BPF

- LSF/MM/BPF Summit 2024, 13 May, More features and use cases for sched_ext, David Vernet.
- LPC, Vienna, 18-20 Sept., 2024 – First public gathering of sched_ext community, many presentations
  - Sched_ext at LPC 2024, Jonathan Corbet, Sept. 26, 2024
  - „Hey, pssst, try this.” The underground culture around custom CPU schedulers.
  - The current status and future potential of sched_ext, David Vernet
- Scheduling with superpowers: Using sched_ext to get big perf gains, Kernel recepies, Oct. 1, 2024, David Vernet.
  - Since last year the project has grown significantly; both in terms of its technical capabilities, as well as in the number of contributors and users of the project.
  - sched_ext now runs at massive scale at **Meta**, and will also soon run as the default scheduler on **Steam Deck devices** (the year of Linux gaming is upon us at last)!
  - Some cutting edge sched_ext schedulers enable great performance on certain workloads.
  - New features available in sched_ext, like **cpufreq** integration, which can improve both datacenter and handheld workloads.

Quotes of the week – posted by J. Corbet on Jan 2024

I ended up writing a Linux scheduler in Rust using sched-ext during Christmas break, just for fun. I'm pretty shocked to see that it doesn't just work, but it can even outperform the default Linux scheduler (EEVDF) with certain workloads (i.e., gaming).

Writing a Linux scheduler in Rust that runs in user-space, OSPM, 30 May 2024
Crafting a Linux kernel scheduler that runs in user-space using Rust, LPC, 18-20 Sept, 2024

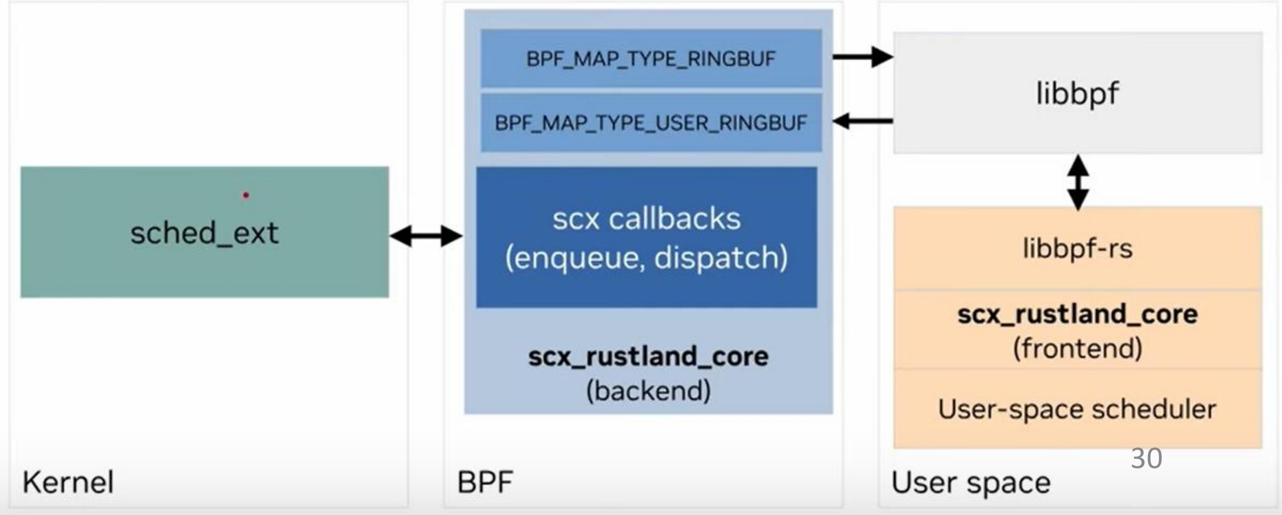
Andrea Righi

Principal System Software Engineer at NVIDIA

Andrea Righi introduced **scx_rustland**, a framework to write CPU schedulers for Linux that run as user-space programs.

Initially a project aimed at teaching operating systems concepts to undergraduate students, it led Righi to appreciate the convenience of the change/build/run workflow to modify the kernel's behavior without rebooting.

This effort was able to prove that user-space scheduling is not only possible, but can even reach respectable performance, such as video gaming at 60 frames per second while compiling the Linux kernel at the same time.


BPF_MAP_TYPE_RINGBUF
BPF_MAP_TYPE_USER_RINGBUF
libbpf
sched_ext
scx callbacks (enqueue, dispatch)
libbpf-rs
scx_rustland_core (frontend)
scx_rustland_core (backend)
User-space scheduler
Kernel
BPF
User space

scx_rustland_core architecture

Earliest Eligible Virtual Deadline First (EEVDF)

- An EEVDF CPU scheduler for Linux, Jonathan Corbet, March 9, 2023.
- Completing the EEVDF scheduler, J. Corbet, April 11, 2024.
- Work by **Peter Zijlstra**.
- (**Oct 30, 2023**) Merged as an option for the **6.6 kernel**.
- It should provide **better performance and fairness** while **relying less on fragile heuristics**. The merge message notes that there may be some rare performance regressions with some workloads, and that work is ongoing to resolve them.
- (**April 05, 2024**) Peter Zijlstra posted a patch series intended to finish the EEVDF work. Beyond some fixes, this work includes a significant behavioral change and a new feature intended to **help latency-sensitive tasks**.
- One place where there is a desire for improvement is in the handling of **latency**.
- The amount of CPU time given to any two processes (with the same nice value) will be the same, but the low-latency process will get it in a larger number of shorter slices.
- Currently a **default scheduler** (replaced CFS).

Earliest Eligible Virtual Deadline First (EEVDF)

- There are many constraints beyond the fair allocation of CPU time that are placed on the scheduler.
  - It should maximize the benefit of the system's memory caches.
  - It should preserve battery life (Power management).
  - Improvement in the handling of latency is required.
- **CFS** does not give processes a way to express their latency requirements; nice values (priorities) can be used to give a process more CPU time, but that is not the same thing.
- For each process, EEVDF calculates the difference between the time that process should have gotten (a task's virtual run time) and how much it actually got (its actual running time); that difference is called **lag**.
- For any process with a **negative lag**, there will be a time in the future where the time it is entitled to catches up to the time it has actually gotten and it will become eligible again; that time is deemed the **eligible time**.
- The **virtual deadline** is the earliest time by which a process should have received its due CPU time. This deadline is calculated by adding a process's allocated time slice to its eligible time. A process with a 10ms time slice, and whose eligible time is 20ms in the future, will have a virtual deadline that is 30ms in the future.
- EEVDF will run the process with the **earliest virtual deadline first**.

Earliest Eligible Virtual Deadline First (EEVDF)

1. CPU-bound tasks (A, B, and C) start at the same time. Before any of them runs, they will all have a **lag of zero**:

   A: 0ms B: 0ms C: 0ms

2. Since none of the tasks have a negative lag, **all are eligible**. If the scheduler picks A to run first with a **30ms** (to pick a number) **time slice**, and if A runs until the time slice is exhausted, the lag situation will look like this:

   A: -20ms B: 10ms C: 10ms

   Over those 30ms, each task was entitled to 10ms of CPU time. A got 30ms, so it accumulated a lag of -20ms; the other two tasks, which got no CPU time at all, ended up with 10ms of lag.

3. **Task A is no longer eligible**, so the scheduler will have to pick one of the others next. If B is given (and uses) a 30ms time slice, the situation becomes:

   A: -10ms B: -10ms C: 20ms

   Each task has earned 10ms of lag corresponding to the CPU time it was entitled to, and B burned 30ms by actually running. Now only **C is eligible**, so the scheduler's next decision is easy.

Earliest Eligible Virtual Deadline First (EEVDF)

- **The sum of all the lag values** in the system is always **zero**.
- Any task can **request shorter time slices**, which will cause it to be run sooner and, possibly, more frequently. If, however, the requested time slice is too short, the task will find itself frequently preempted and will run slower overall.
- A task can use the **sched_setattr()** system call, passing the desired slice time (in nanoseconds) in the **sched_runtime** field of the **sched_attr** structure.
- EEVDF allows one task to **preempt** another if its **virtual deadline is earlier**. This provides more consistent timings for short-time-slice tasks, while slowing long-running tasks slightly.
- When a task sleeps, it is normally removed from the run queue so that the scheduler need not consider it. In EEVDF an ineligible process that goes to sleep will be left on the queue, but marked for "deferred dequeue". Since it is ineligible, it will not be chosen to execute, but its **lag will increase** according to the virtual run time that passes. Once the lag goes positive, the scheduler will notice the task and remove it from the run queue. The result of this implementation is that a **task that sleeps briefly will not be able to escape a negative lag value**, but **long-sleeping tasks will eventually have their lag debt forgiven**. A positive lag value is, instead, retained indefinitely until the task runs again.

Last discussions on schedulers

- Scheduler related topics discussed at OSPM 2023
  - Part 1, Part 2
- Scheduler related topics discussed at OSPM 2024
  - Part 1, Part 2

Role of CPU Idle loop

- Intel - CPU idle time management, Rafael J. Wysocki
- When there are no other tasks to run on a CPU, the **idle task** runs on it.
- The idle task's code is the **idle lo**op.
- The idle loop calls into cpuidle to allow the CPU to be put into an **energy-saving state** (if this makes sense).
- Cpuidle uses a **governor** to decide which idle state to put the CPU into (and whether of not to stop the scheduler tick on it).
- Three cpuidle governors are available (in the mainline), but 2 of them are practicaly relevant (**menu** and **teo**).
- Idle state parameters that are used by the governors for making decisions are the ta**r**get **residency** and the **exit latency**.

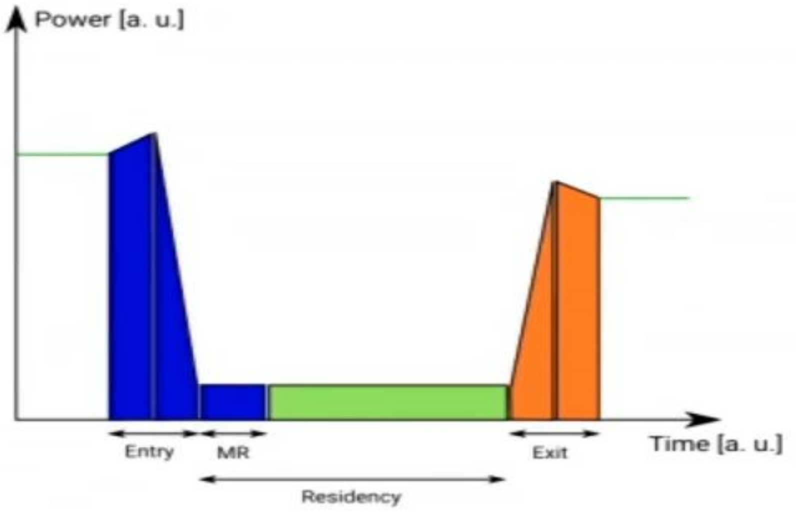
CPU Idle Loop

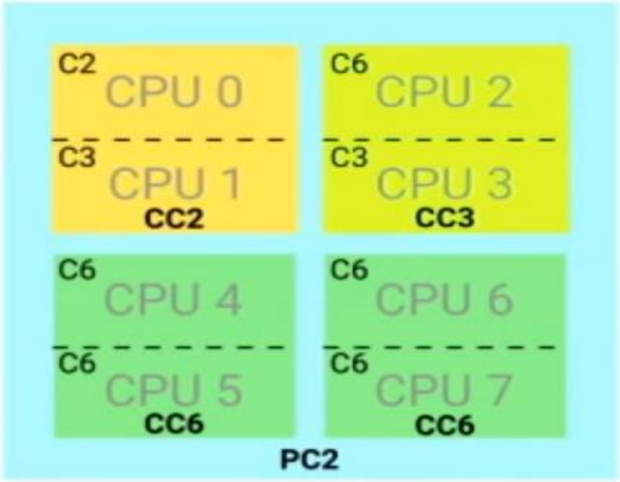
CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.

What’s a CPU to do when it has nothing to do?, Tom Yates, October 2018.

Although increasingly deep idle states consume decreasing amounts of power, they have increasingly large costs to enter and exit. It is in the kernel's best interests to predict how long a CPU will be idle before deciding how deeply to idle it. This is the job of the **idle loop**. The scheduler then calls the **governor**, which does its best to predict the appropriate idle state to enter.

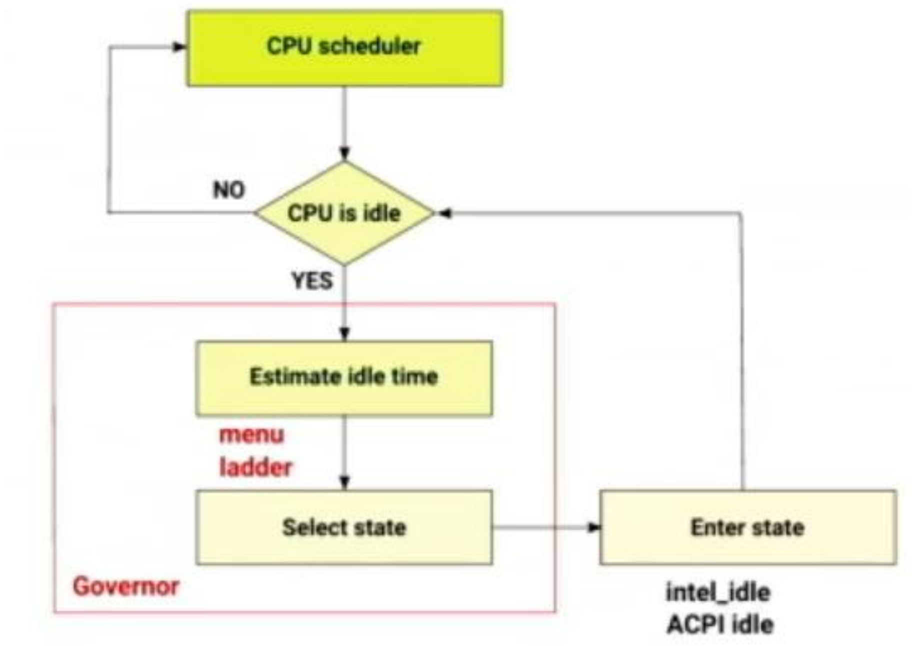
He reworked the idle loop for kernel 4.17 so that the decision about **stopping the tick** is taken *after* the governor has made its recommendation of the idle state.



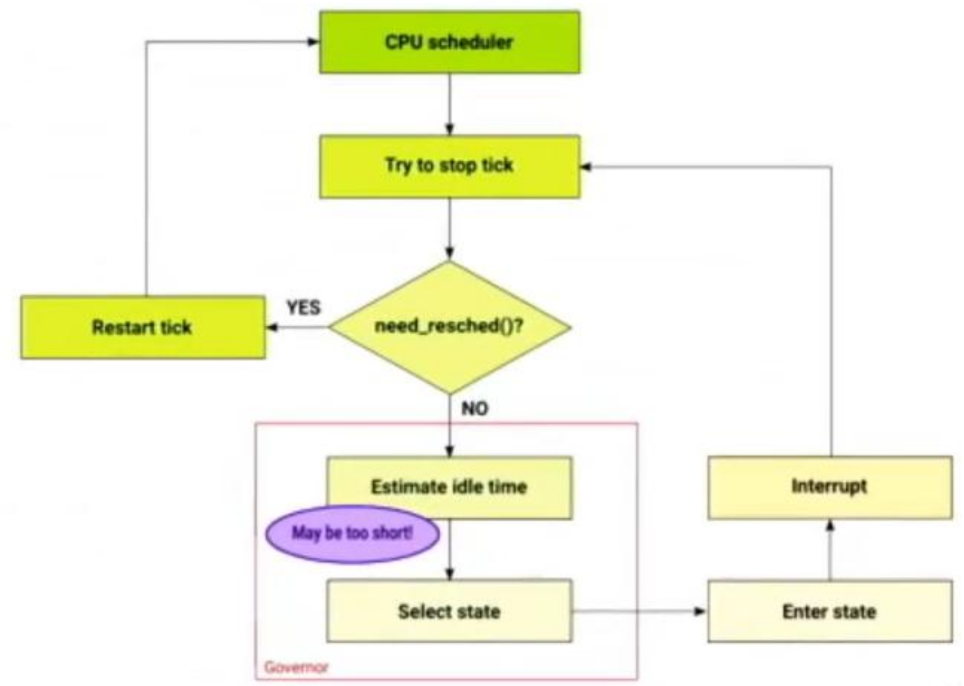


CPU Idle Loop

CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.



High-level CPU idle time management control flow

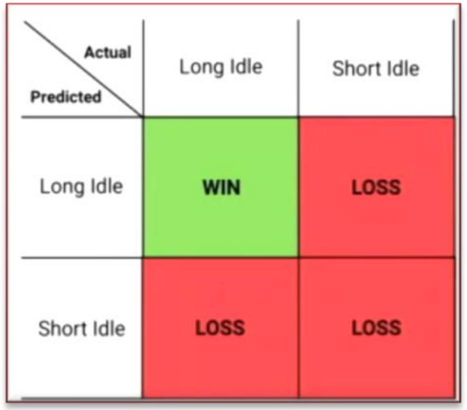


But there is a CPU scheduler tick timer ..
Original idle loop design issue

```
jmd@students$ cat /sys/devices/system/cpu/cpuidle/current_governor_ro
menu
```

CPU Idle Loop

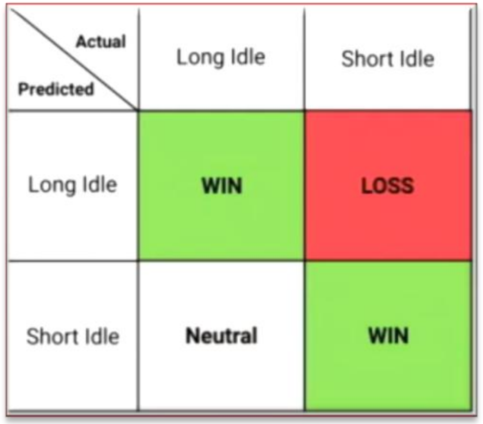
Before



Actual Predicted	Long Idle	Short Idle
Long Idle	WIN	LOSS
Short Idle	LOSS	LOSS

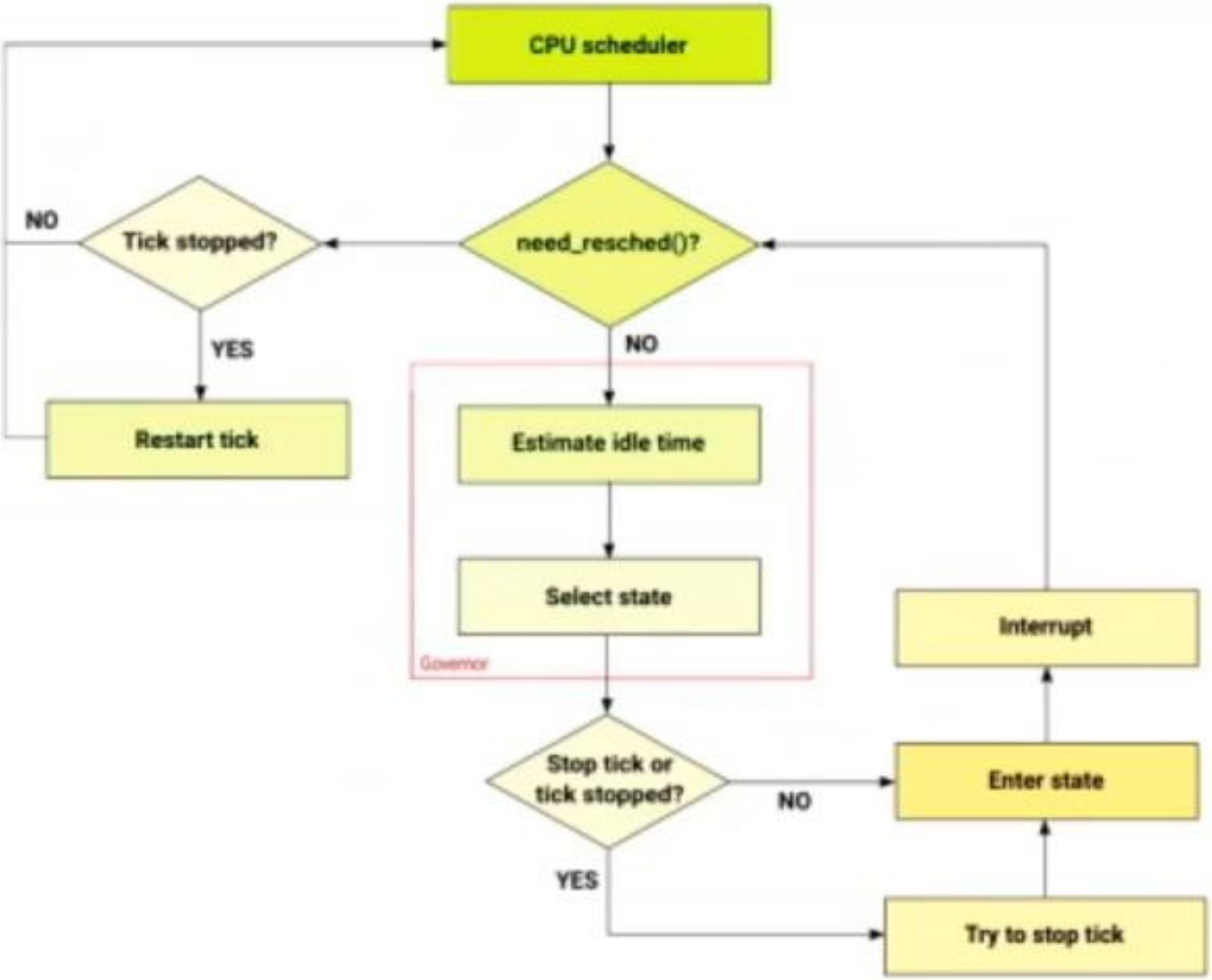
Short idle duration prediction problem

After



Actual Predicted	Long Idle	Short Idle
Long Idle	WIN	LOSS
Short Idle	Neutral	WIN

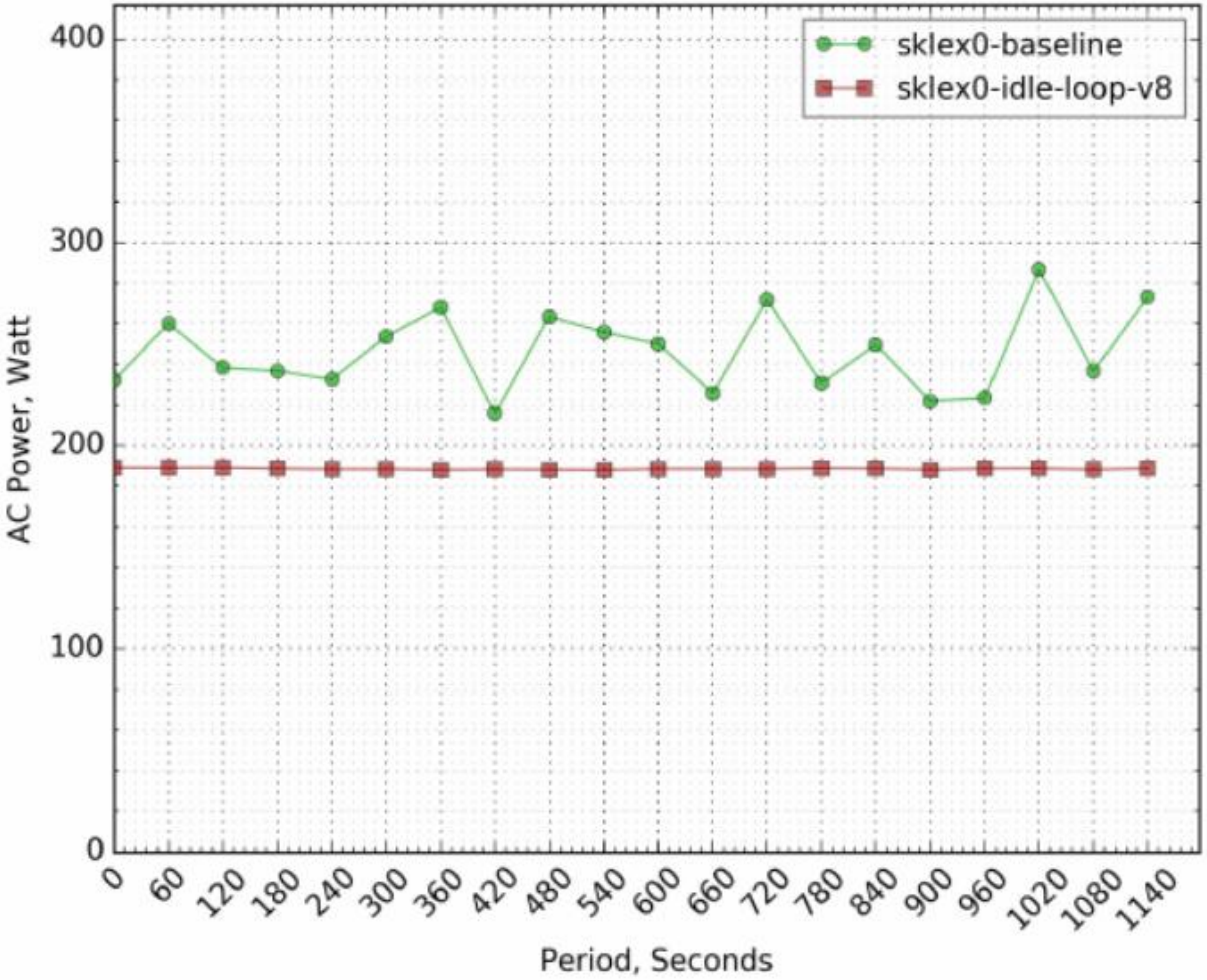
CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.



Redesigned idle loop (Linux 4.17 and later)

Idle power (Intel OTC Server Power Lab)

The green line is with the old idle loop, the red is with the new: power consumption is less under the new scheme, and moreover it is much more predictable than before.



CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.

CPU Idle Loop

- Improving idle behavior in tickless systems, Marta Rybczyńska, December 2018.

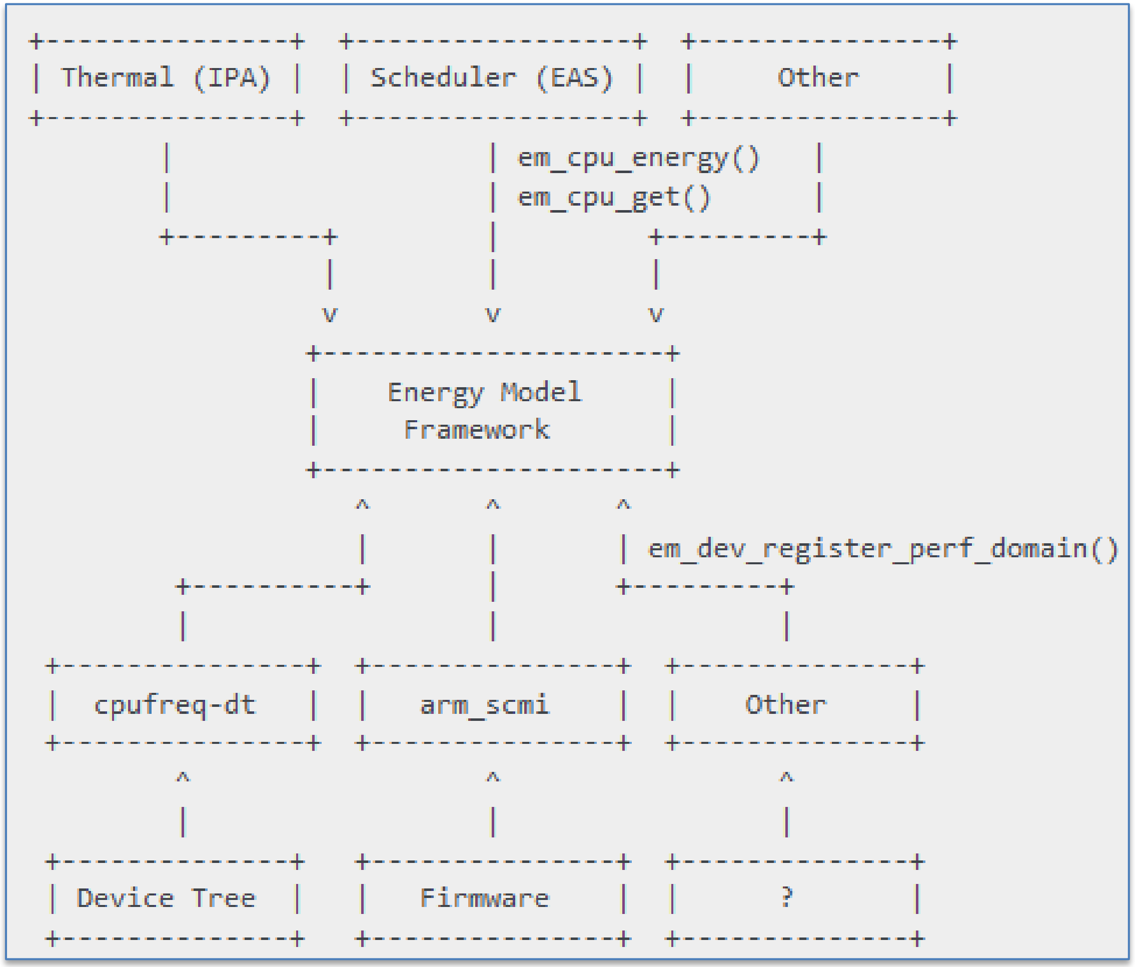
  Linux currently provides two cpu idle governors, **ladder** and **menu**. Wysocki implemented the third, **timer events oriented** (TEO) – similar to menu, but takes into cosideration different factors.

- Fixing SCHED_IDLE, Viresh Kumar, November 2019 – The 5.4 kernel release includes a few improvements to the existing SCHED_IDLE scheduling policy that can help users improve the scheduling latency of their high-priority (interactive) tasks if they use the SCHED_IDLE policy for the lowest-priority (background) tasks.

- Intel - CPU idle time management, Rafael J. Wysocki, Intel, OSPM 20, May 31, 2024. Problems with governors – they don’t take latency into account.

Energy Model of devices

Energy models of devices



```
+---------------+  +-----------------+  +---------------+
| Thermal (IPA) |  | Scheduler (EAS) |  |     Other     |
+---------------+  +-----------------+  +---------------+
        |                   | em_cpu_energy()   |
        |                   | em_cpu_get()      |
        +---------+         |         +---------+
                  |         |         |
                  v         v         v
                 +---------------------+
                 |    Energy Model     |
                 |     Framework       |
                 +---------------------+
                    ^       ^       ^
                    |       |       | em_dev_register_perf_domain()
         +----------+       |       +---------+
         |                  |                 |
 +---------------+  +---------------+  +--------------+
 |  cpufreq-dt   |  |   arm_scmi    |  |    Other     |
 +---------------+  +---------------+  +--------------+
         ^                  ^                 ^
         |                  |                 |
 +--------------+   +---------------+  +--------------+
 | Device Tree  |   |   Firmware    |  |      ?       |
 +--------------+   +---------------+  +--------------+
```

- The **Energy Model** (**EM**) framework serves as an interface between drivers knowing the **power consumed by devices** at various performance levels, and the kernel subsystems willing to use that information to make **energy-aware decisions**.
- The source of the information about the power consumed by devices can vary from one platform to another. These power costs can be estimated using **device tree dat**a in some cases. In others, the **firmware** will know better. Alternatively, **user space** might be best positioned.
- In order to avoid each and every client subsystem to re-implement support for each and every possible source of information on its own, the **EM framework intervenes as an abstraction layer** which standardizes the format of power cost tables in the kernel.
- The power values might be expressed in **micro-Watts** or in an **abstract scale**.
- In case of **CPU** the EM framework manages power cost tables per **performance domain** in the system. A performance domain is a **group of CPUs** whose performance is scaled together.
- Performance domains generally have a 1-to-1 mapping with **CPUFreq policies**.

Energy Aware Scheduling

- Energy Aware Scheduling (EAS) gives the scheduler the ability to predict the impact of its decisions on the **energy consumed by CPUs**.
- EAS relies on an **Energy Model** of the CPUs to select an energy efficient CPU for each task, with a minimal impact on throughput.
- EAS operates only on **heterogeneous CPU topologies** (such as Arm big.LITTLE and other multi-core SoCs) because this is where the potential for saving energy through scheduling is the highest.
- Definitions
  - **energy** = [joule] (resource like a battery on powered devices)
  - **power** = energy/time = [joule/second] = [watt]
- The goal of EAS is to **minimize energy**, while still getting the job done. That is, we want to
  - **maximize**: performance [inst/s] / power [W] which is equivalent to
  - **minimizing**: energy [J] / instruction.
- It is essentially an alternative optimization objective to the current **performance-only** objective for the scheduler. This alternative considers two objectives: **energy-efficiency** and **performance**.
- The use-cases where EAS can help the most are those involving a **light/medium CPU utilization**.

Energy Aware Scheduling

- Researched since 2013. In 2019 added to **Linux 5.0**.
- Energy Aware Scheduling (EAS) on ARM wiki. Arm, Linaro and key partners are contributing jointly to the development of EAS. Energy-Aware Scheduling Project on linaro.org.
- An Unbiased Look at the Energy Aware Scheduler, Vitaly Wool, Embedded Linux Conference, 2018.
- Evaluating vendor changes to the scheduler, Jonathan Corbet, May 2020.

  The benchmark results for each of these patches were remarkably similar. They all tended to **hurt performance** by 3-5% while **reducing energy** use by 8-11%.

- Saving frequency scaling in the data center, J. Corbet, May 2020.

  Frequency scaling — adjusting a CPU's operating frequency to save power when the workload demands are low — is common practice across systems supported by Linux. It is, however, viewed with some suspicion in data-center settings, where power consumption is less of a concern and there is a strong emphasis on **getting the most performance** out of the hardware.

- Imbalance detection and fairness in the CPU scheduler, J. Corbet, May 2020.

Scheduling – thermal pressure

Telling the scheduler about thermal pressure, Marta Rybczyńska, May 2019.

Even with radiators and fans, a system's **CPUs** can **overheat**. When that happens, the kernel's thermal governor will **cap the maximum frequency** of that CPU to allow it to cool. The scheduler, however, is **not aware that the CPU's capacity** has changed; it may schedule more work than optimal in the current conditions, leading to a performance degradation.

The solution adds an interface to inform the scheduler about thermal events so that it can assign tasks better and thus improve the overall system performance.

The term **thermal pressure** means the difference between the maximum processing capacity of a CPU and the currently available capacity, which may be reduced by overheating events.

The two approaches, the **thermal pressure approach** and **energy-aware scheduling** (**EAS**), have different scope: thermal pressure is going to work better in asymmetric configurations where capacities are different and it is more likely to cause the scheduler to move tasks between CPUs.

The two approaches should also be independent because thermal pressure should work even if EAS is not compiled in.

Enhancements and adjustments of the thermal control subsystem, Rafael J. Wysocki , Linux Plumbers Conference (LPC), September 19, 2024.

Scheduling – Arm big.LITTLE CPU chip

Scheduling for asymmetric Arm systems, Jonathan Corbet, November 2020.

The **big.LITTLE architecture** placed **fast** (but power-hungry) and **slower** (but more power-efficient) CPUs in the same **system-on-chip** (**SoC**); significant scheduler changes were needed for Linux to be able to properly distribute tasks on such systems.

Putting tasks on the wrong CPU can result in poor performance or excessive power consumption, so a lot of work has gone into the problem of **optimally distributing workloads** on big.LITTLE systems.

When the scheduler gets it wrong, though, performance will suffer, but things will still work.

Future Arm designs, include systems where some CPUs can run **both 64-bit and 32-bit** tasks, while others are **limited to 64-bit tasks** only. The result of an incorrect scheduling choice is no longer a matter of performance; it could be catastrophic for the workload involved.

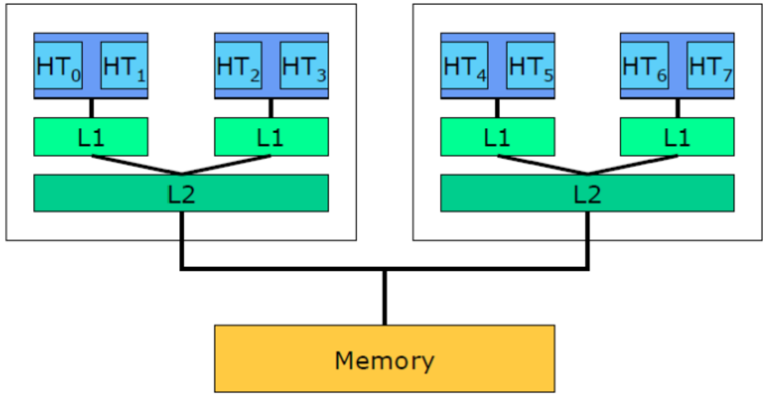
What should happen if a 32-bit task attempts to run on a 64-bit-only CPU?

- Kill the task or
- recalculate the task's CPU-affinity mask?

Cortex A57/A53 MPCore big.LITTLE CPU chip

Core scheduling

Symmetric Multi-threading (SMT)
Hyperthreading



Core scheduling, Jonathan Corbet, February 2019.

SMT (**simultaneous multithreading**) increases performance by turning one physical CPU into two virtual CPUs that share the hardware; while one is waiting for data from memory, the other can be executing. Sharing a processor this closely has led to security issues and concerns for years, and many security-conscious users disable SMT entirely.

On kernels where core scheduling is enabled, a **core_cookie** field is added to the task structure. These cookies are used to define the trust boundaries; two processes with the same cookie value trust each other and can be allowed to run simultaneously on the same core. (Peter Zijlstra)

Completing and merging core scheduling, Jonathan Corbet, May 2020.

A set of virtualization tests showed the system running at **96%** of the performance of an unmodified kernel with **core scheduling enabled**; the 4% performance hit hurts, but it's far better than the **87%** performance result measured for this workload with **SMT turned off** entirely.

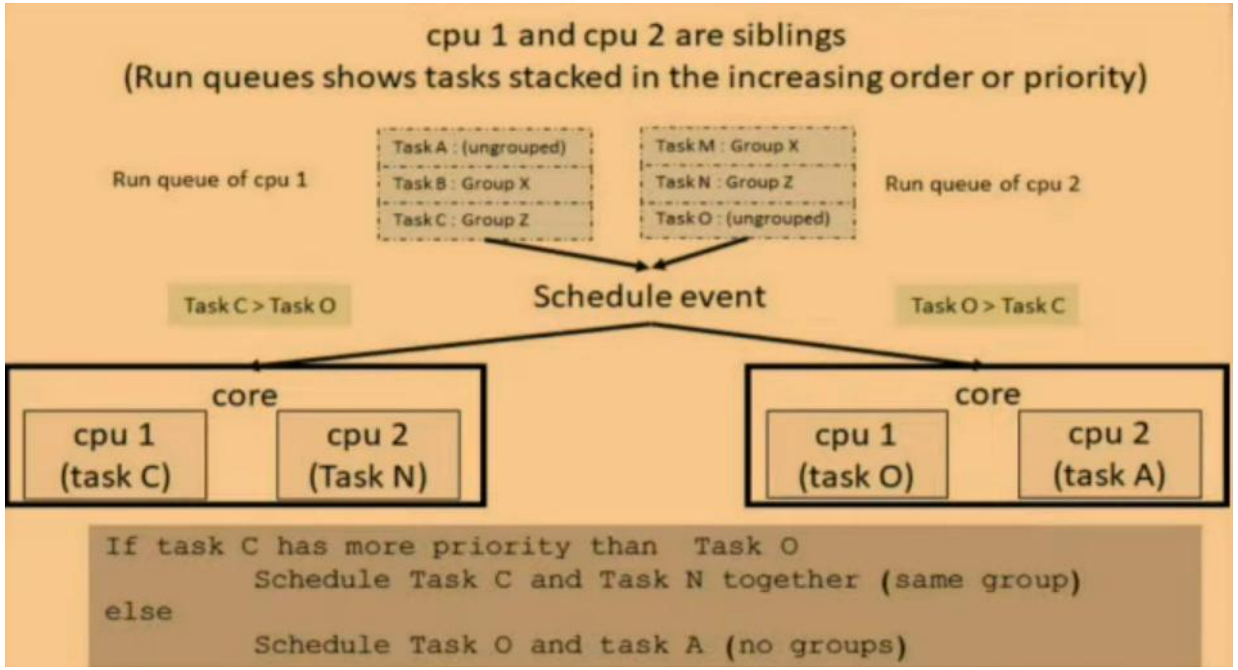
The all-important kernel-build benchmark showed almost no penalty with core scheduling, while turning off SMT cost 8%.

Core scheduling

Core Scheduling Looks Like It Will Be Ready For Linux 5.14 To Avoid Disabling SMT/HT, Michael Larabel, May 2021.

Core scheduling lands in 5.14, Jonathan Corbet, 2021.

Core scheduling should be effective at mitigating user-space to user-space and user-to-kernel attacks when the functionality is properly used. But the default kernel policy will not change over how tasks are scheduled but is up to the administrator for identifying tasks that can or cannot share CPU resources.



```
If task C has more priority than  Task O
      Schedule Task C and Task N together (same group)
else
      Schedule Task O and task A (no groups)
```

https://www.youtube.com/watch?v=_8_xUf47-jE

Conclusions

Scheduler performance varies dramatically according to hardware and workload, and as a result we strongly encourage Linux distributions to take an increased level of responsibility for selecting appropriate default schedulers that best suit the intended usage of the system.