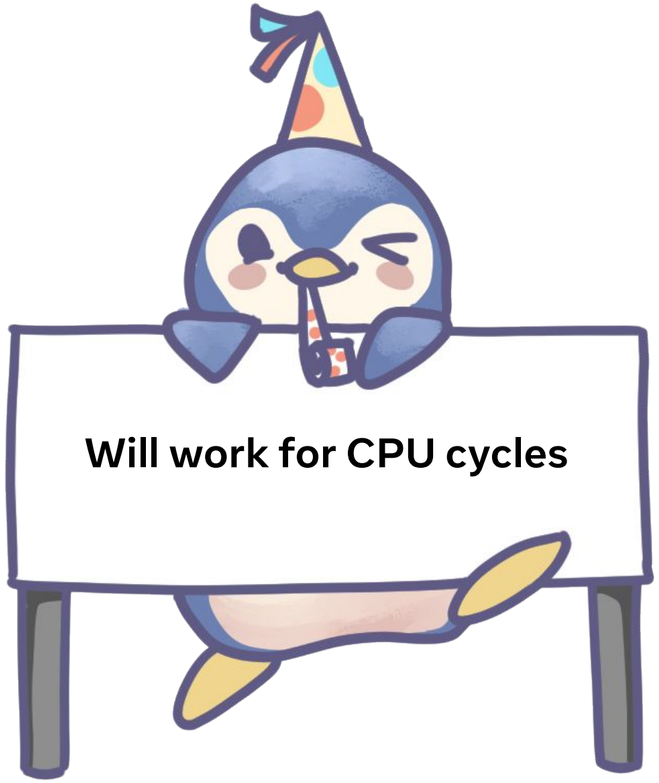
# Sched Ext

**The extensible sched_class**


David Vernet
Kernel engineer

Meta

# Agenda

01 Background and motivation

02 Building schedulers with sched_ext

03 Example schedulers

04 Current status and future plans

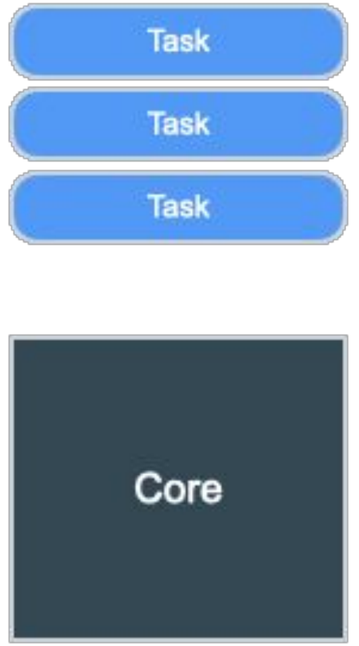
05 Questions?

# 01 Background and motivation

# What is a CPU scheduler?

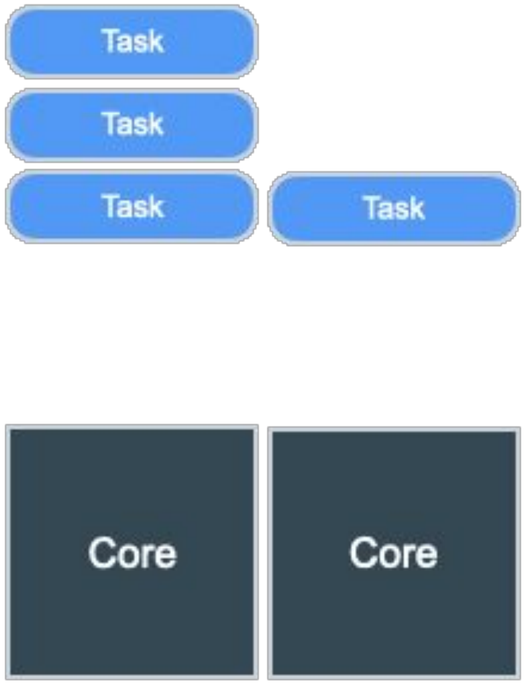
# CPU schedulers multiplex threads onto core(s)

- Manages the finite resource of CPU between all of the execution contexts on the system
- Decide who gets to run next, where they run, and for how long
- Does context switching

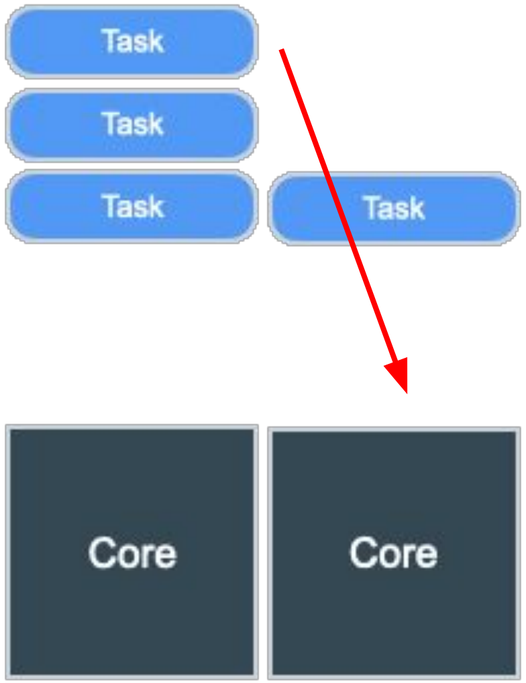


# What about multiple cores?

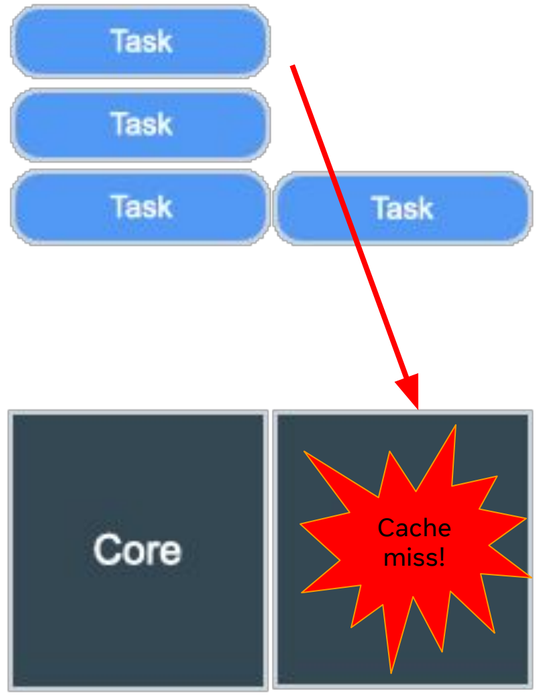
# No problem, just move tasks between cores when one becomes available



# No problem, just move tasks between cores when one becomes available



# Except that caches exist, there’s a latency penalty for migrations, etc...



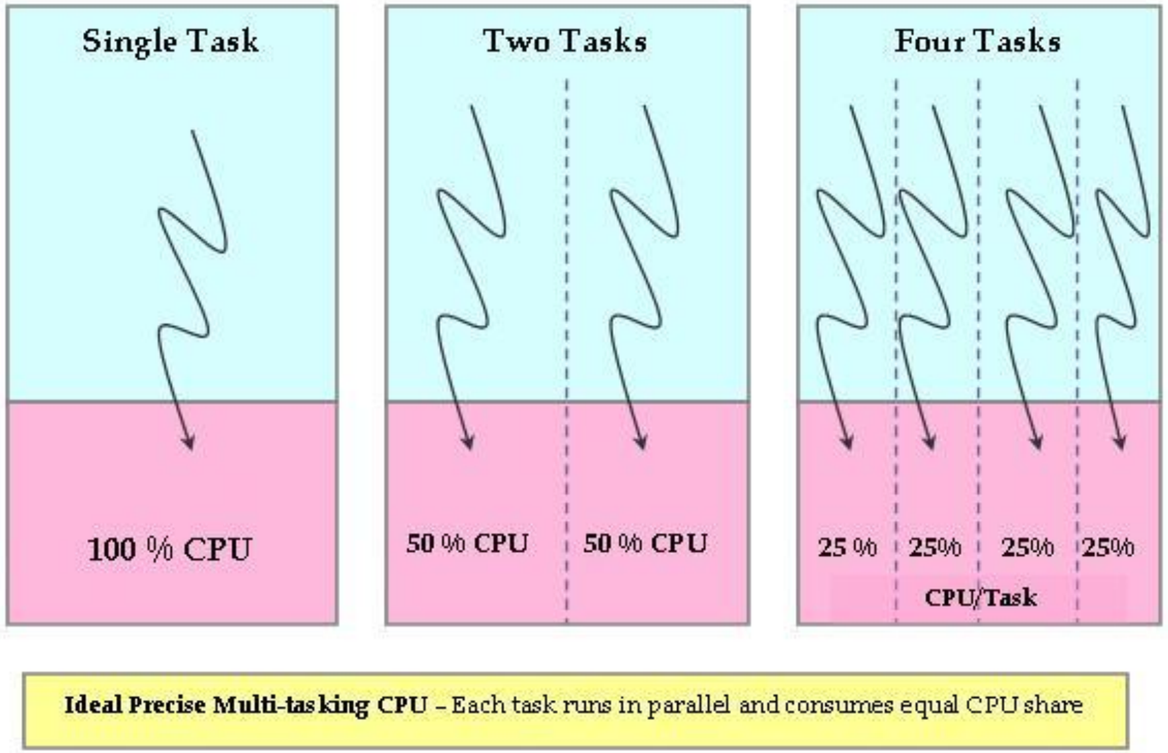
# Things get very complicated very quickly

- Very challenging technical problem
  - **Fairness**: Everyone should get *some* CPU time
  - **Optimization**: Make optimal use of system resources, minimize critical sections
  - **Low overhead**: Should run for as short as possible
  - **Generalizable**: Should work on every architecture, for every workload, etc.

# CFS: The **C**ompletely **F**air **S**cheduler

# CFS is a “fair, weighted, virtual time scheduler”

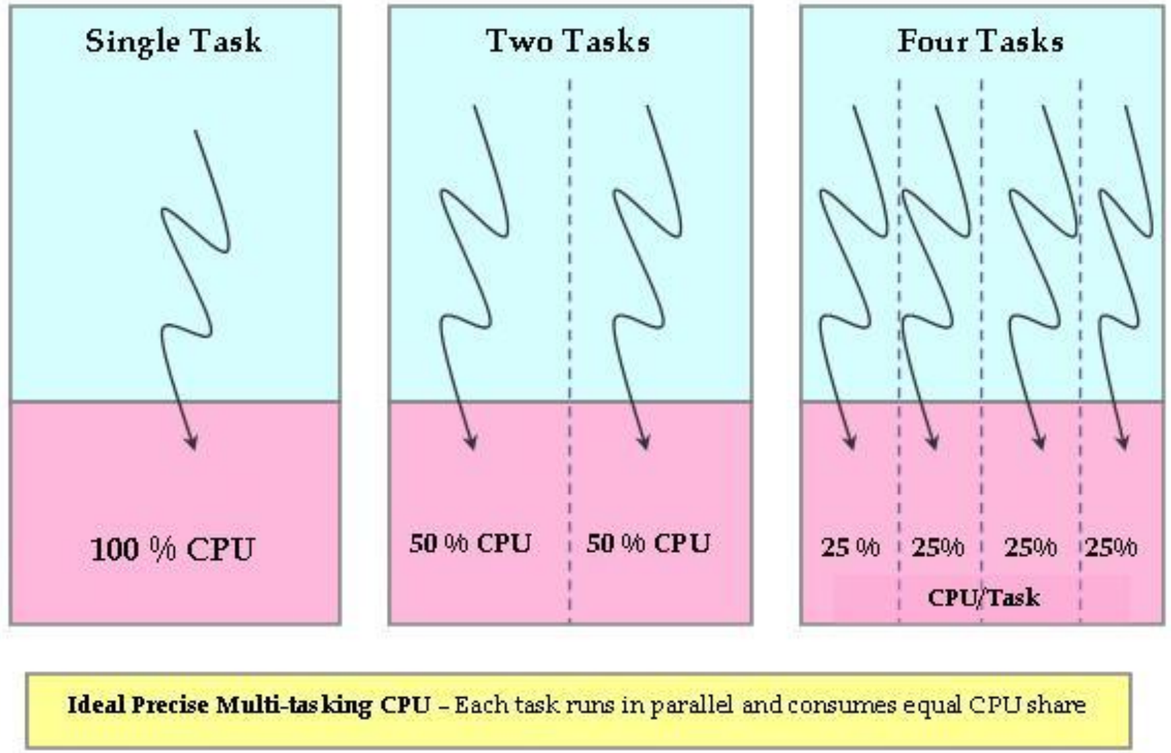
- Threads given proportional share of CPU, according to their weight and load
  - In example on right, all threads have equal weight
- Conceptually quite simple and elegant
  - Also has drawbacks, more on this later



# CFS is a “fair, weighted, virtual time scheduler”

- Threads given proportional share of CPU, according to their weight and load
  - In example on right, all threads have equal weight
- Conceptually quite simple and elegant
  - Also has drawbacks, more on this later

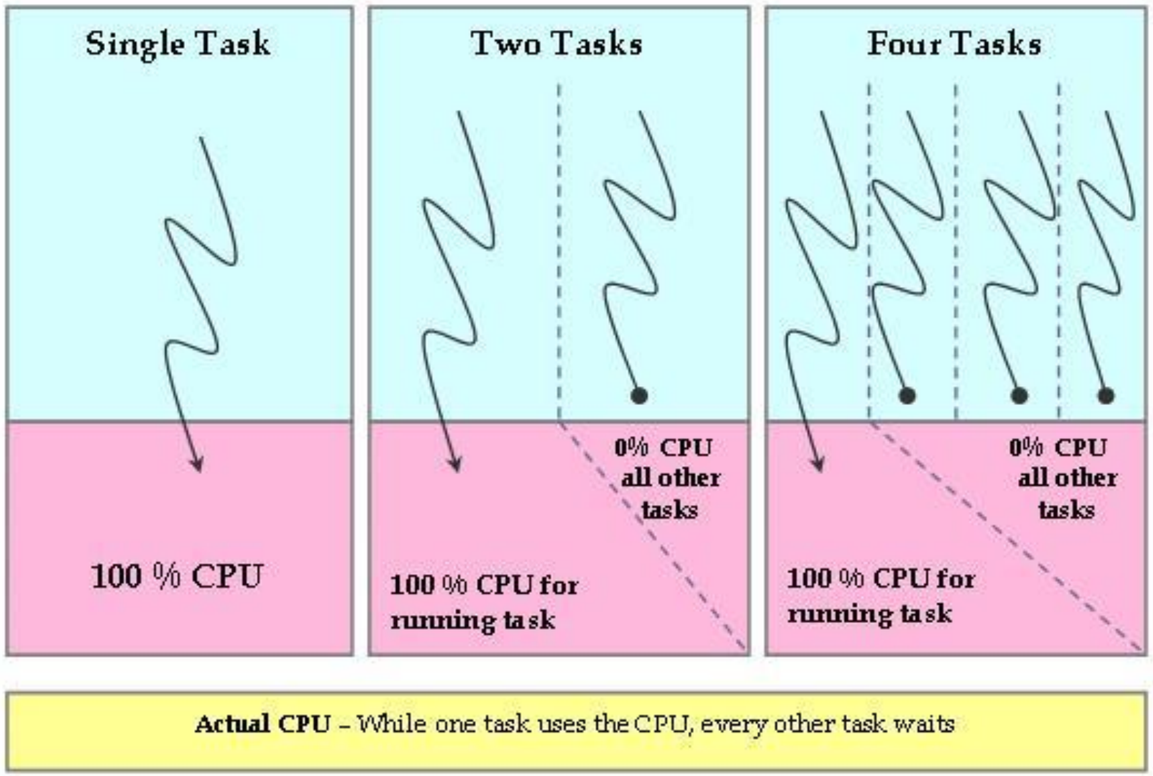
Conceptual →




# CFS is a “fair, weighted, virtual time scheduler”

- Threads given proportional share of CPU, according to their weight and load
  - In example on right, all threads have equal weight
- Conceptually quite simple and elegant
  - Also has drawbacks, more on this later

Actual →




# CFS has been in the kernel since 2007

**All of lore.kernel.org**

search help / color / mirror / Atom feed

*** [patch] CFS scheduler, -v6**
**@ 2007-04-25 21:47 Ingo Molnar**
2007-04-26 2:14 ` Gene Heskett
` (6 more replies)
0 siblings, 7 replies; 89+ messages in thread
From: Ingo Molnar @ 2007-04-25 21:47 UTC (permalink / raw)
To: linux-kernel
Cc: Linus Torvalds, Andrew Morton, Con Kolivas, Nick Piggin,
Mike Galbraith, Arjan van de Ven, Peter Williams,
Thomas Gleixner, caglar, Willy Tarreau, Gene Heskett, Mark Lord,
Zach Carter, buddabrod

i'm pleased to announce release -v6 of the CFS scheduler patchset. The
main goal of CFS is to implement "high quality desktop scheduling" as
well as technically possible.

The CFS patch against v2.6.21-rc7 or against v2.6.20.7 can be downloaded
from the usual place:

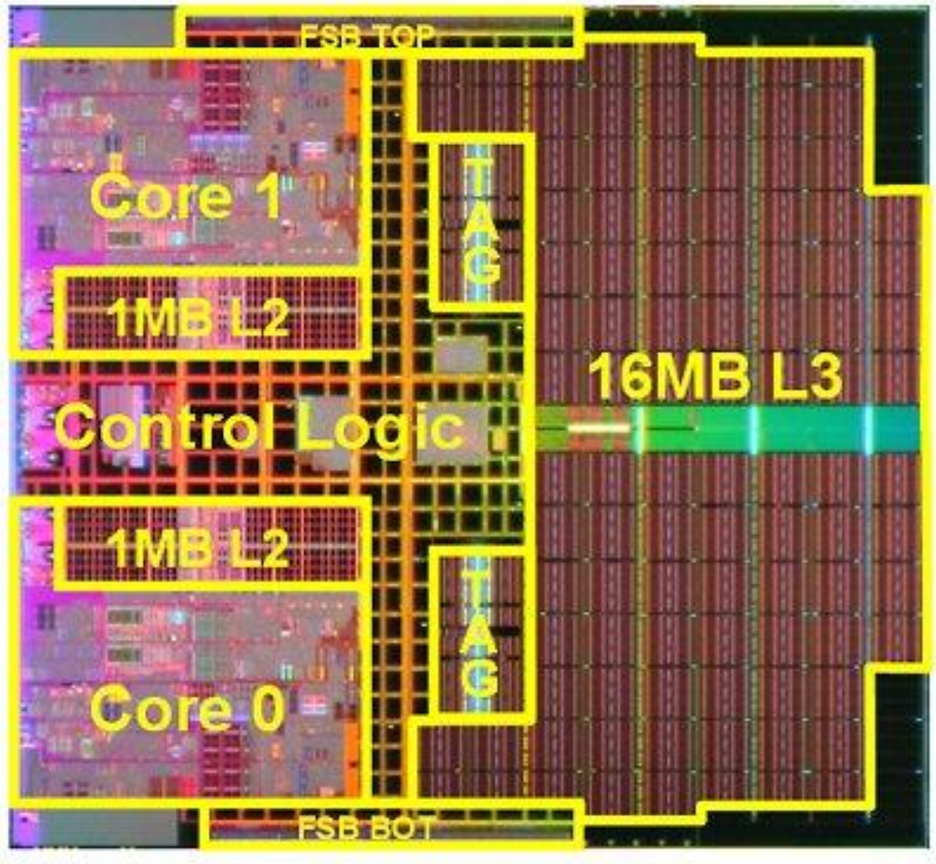
http://redhat.com/~mingo/cfs-scheduler/

i got lots of -v5 feedback (thanks and please keep the reports coming!)
so the -v6 release includes many bugfixes and improvements:

19 files changed, 317 insertions(+), 744 deletions(-)

# CFS was built in a simpler time

- Much smaller CPUs
- Topologies much more homogeneous
- Cores spaced further apart, migration cost typically high
- Power consumption and die area wasn’t as important

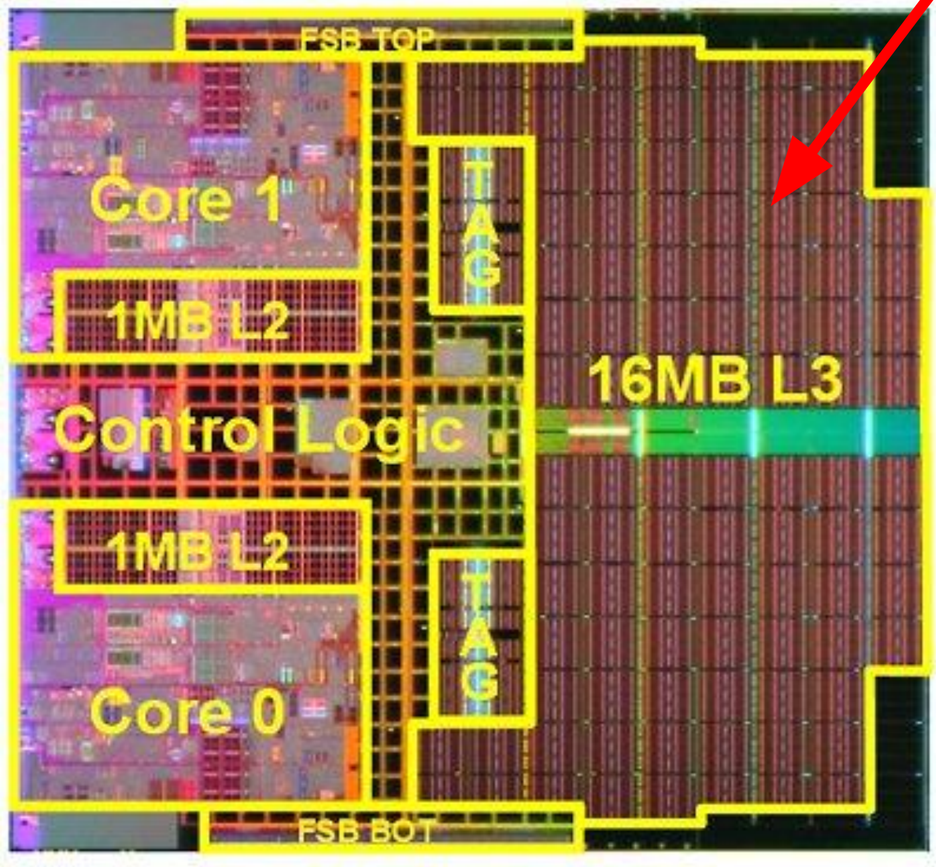


# CFS was built in a simpler time

- Much smaller CPUs
- Topologies much more homogeneous
- Cores spaced further apart, migration cost typically high
- Power consumption and die area wasn’t as important
- The fundamental assumptions behind heuristics may be easier to justify

Just two cores

Just one L3 cache



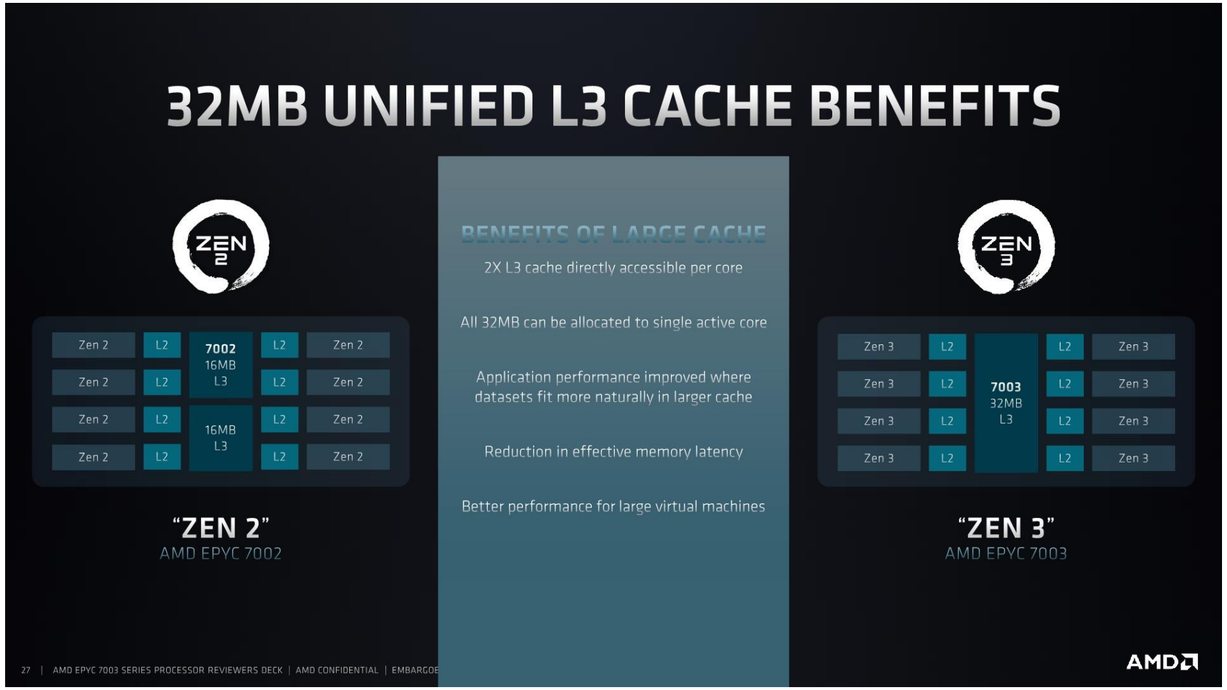

Intel Xeon MP 71xx die

# New reality: complex hardware topologies, and heterogeneity

- **CCD**'s (Core Complex Dies) aggregate groups of **CCX**'s (Core Complexes)
  - A CCX is a cluster of cores that share an L3 cache
  - Can have multiple CCXs per NUMA node
  - Can have multiple CCXs per CCD

# Architectures *much* more complicated now

- Heterogeneity is becoming the norm
- Non-uniform memory accesses between sockets
- Non-uniform memory accesses between CCDs
- Non-uniform memory accesses between CCXs
- Non-uniform memory accesses between CCXs in the same CCD

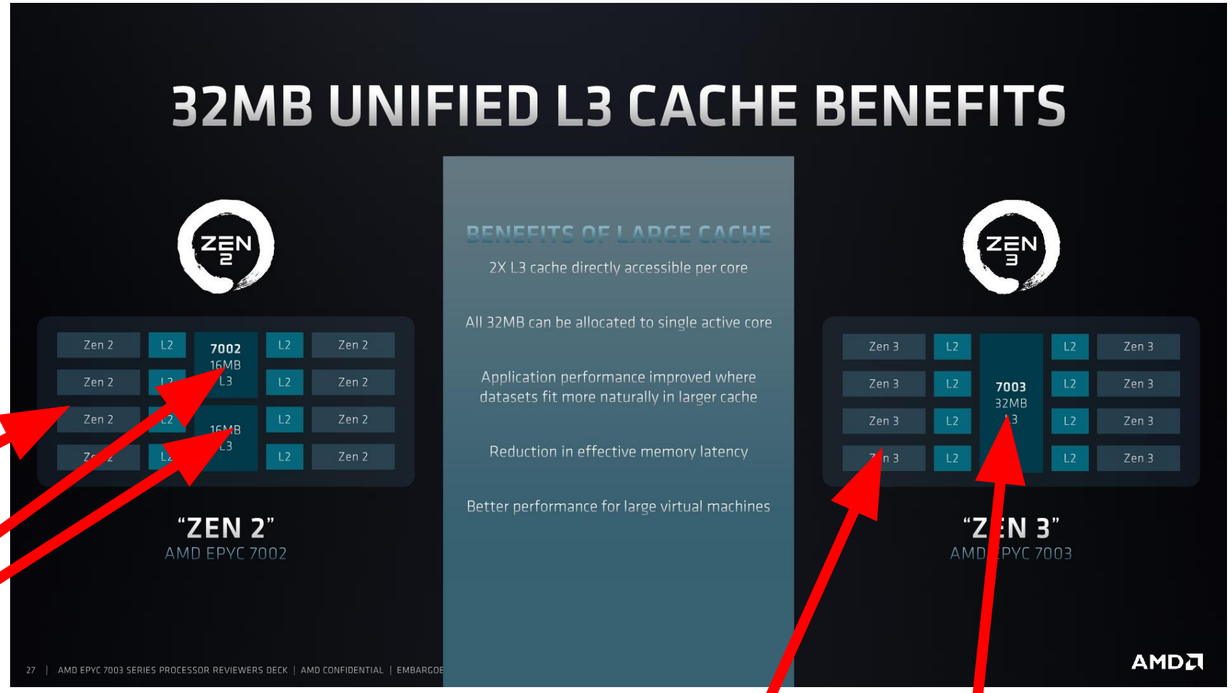



AMD Zen 2 Rome

AMD Zen 3 Milan

# Architectures *much* more complicated now

- Heterogeneity is becoming the norm
- Non-uniform memory accesses between sockets
- Non-uniform memory accesses between CCDs
- Non-uniform memory accesses between CCXs
- Non-uniform memory accesses between CCXs in the same CCD




AMD Zen 2 Rome

4 cores per "CCX"
8 cores per "CCD"

2 L3 caches per CCD!

8 cores per "CCX"
8 cores per "CCD"

1 L3 cache per CCD!

# CFS is great, but has some drawbacks

- Experimentation is difficult: need to recompile + reboot + rewarm caches
- *Very* complex, often takes O(years) for people to fully onboard
- Generalizable scheduler
  - Often leaves some performance on the table for some workloads / architectures
  - Impossible to make everyone happy all of the time
- Difficult to get new features upstreamed
  - Can’t regress the scheduler
  - High bar for contributions (understandably)
  - Results in lots of out of tree schedulers, vendor hooks, etc

# Result: usually lots of heuristics in the scheduler

- Scheduler did something I didn’t like, tweak the behavior to accommodate
  - Err on the side of keeping a task local to promote cache locality
  - Be more likely to schedule someone who was previously your hypertwin
- Don’t apply well to every workload or architecture
- Often result in non-intuitive behavior
  - Setting sched_migration_cost_ns knob to 0 may still not migrate a task to use an idle core
    - SHARED_RUNQ patchset is meant to help address this: https://lore.kernel.org/all/20230809221218.163894-1-void@manifault.com/

# Quick aside on BPF

# BPF: The safe way to run kernel code

- Kernel feature that allows custom code to run safely in the kernel
- Started in the early days as a way to do custom packet filtering, now a much, much larger and richer ecosystem
- Far too much to cover here. Conceptually, just think “safe JIT in the kernel”

# Introducing: sched_ext

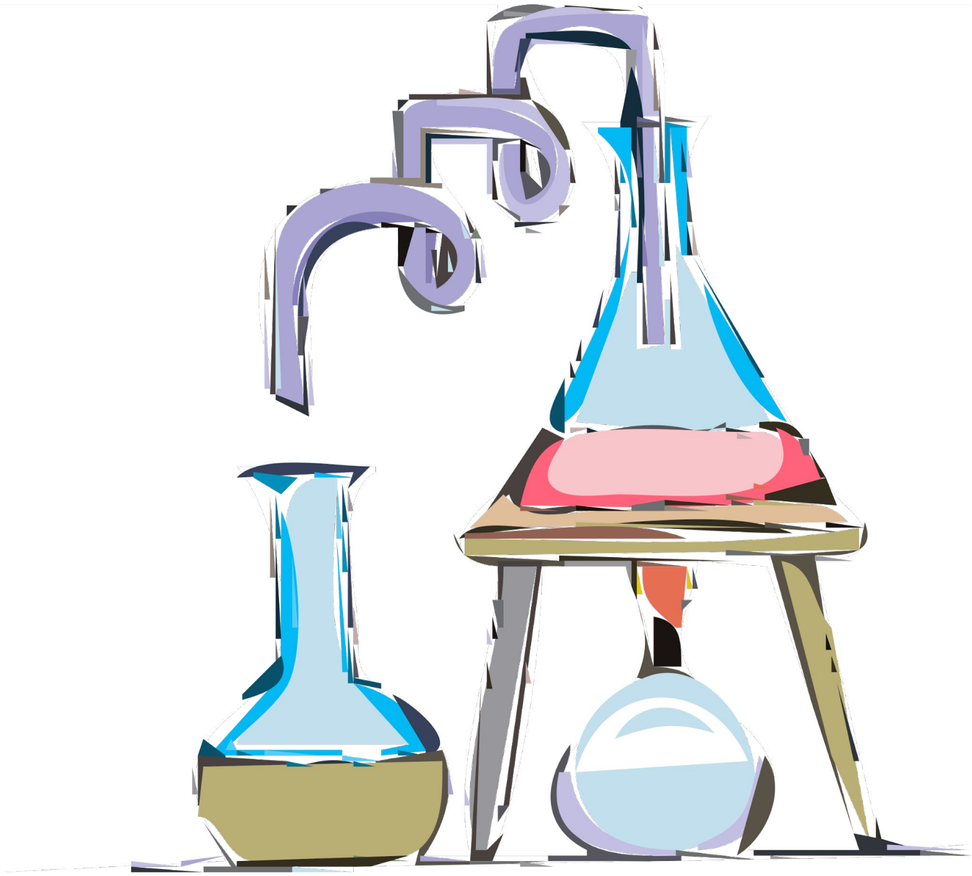
# sched_ext enables scheduling policies to be implemented in BPF programs

1. Write a scheduler policy in BPF
2. Compile it
3. Load it onto the system, letting BPF and core sched_ext infrastructure do all of the heavy lifting to enable it

- New sched_class, at a lower priority than CFS
- No ABI stability restrictions – purely a kernel <-> kernel interface
- GPLv2 only

# Rapid experimentation

- No reboot needed – just recompile BPF prog and reload
- Simple and intuitive API for scheduling policies
  - Does not require knowledge of core scheduler internals
- Safe, cannot crash the host
  - Protection afforded by BPF verifier
  - Watchdog boots sched_ext scheduler if a runnable task isn't scheduled within some timeout
  - New sysrq key for booting sched_ext scheduler through console
- See what works, then implement features in CFS



# Bespoke scheduling policies

- CFS is a general purpose scheduler. Works OK for most applications, not optimal for many
- Optimizes some major Meta services (more on this later)
  - **HHVM optimized by 2.5-3+% RPS**
  - **Looking like a 3.6 - 10+% improvement for ads ranking**
- Google has seen strong results on search, VM scheduling with ghOSt

# Moving complexity into user space

- Offload complicated logic such as load balancing to user space
- Avoids workarounds like custom threading implementations and other flavors of kernel bypass
- Use of floating point numbers
- BPF makes it easy to share data between the kernel and user space

# What is sched_ext *not*?

# sched_ext is **not** meant to replace CFS

- Virtual runtime is an elegant fairness algorithm for a general purpose scheduler
- The kernel will likely always need a general purpose scheduler
- Features discovered with and experimented on with sched_ext can be upstreamed to CFS. One of the main motivators
  - SHARED_RUNQ patchset is is a direct result of sched_ext experimentation: https://lore.kernel.org/all/20230809221218.163894-1-void@manifault.com/

# sched_ext is **not** meant to replace upstream development

- A sched_ext scheduler **must** be GPLv2 to be loaded by the verifier
  - Will fail to load at runtime otherwise
- Several schedulers included in the upstream patch set (mentioned later in the presentation)
- So much out of tree scheduler code already. The hope is that this will *improve* things.

# sched_ext is **not** meant to impose UAPIs restrictions on the kernel

- struct_ops, the main BPF feature backing sched_ext, does not have UAPI guarantees
  - Strict kernel <-> kernel interface
  - User space programs can talk to BPF programs over maps, but this is nothing new for BPF
- The core scheduler API can change, and could break out of tree schedulers
  - Not expected to happen with regularity, but it is allowed according to advertised UAPI policy for sched_ext and struct_ops BPF programs

**DISCLAIMER**: This is a somewhat subjective topic. We do our best to be explicit and both state and document our UAPI guarantees, but at the end of the day, it is up to Linus to interpret this.

# 02 Building schedulers with sched_ext

Kernel Recipes

# Implementing scheduling policies

- BPF program must implement a set of **callbacks**
  - Task wakeup (similar to select_task_rq())
  - Task enqueue/dequeue
  - Task state change (runnable, running, stopping, quiescent)
  - CPU needs task(s) (balance)
  - Cgroup integration
  - ...

- Also provides **fields** which globally configure scheduler
  - Max # of tasks that can be dispatched
  - Timeout threshold in ms (can’t exceed 30s)
  - Name of scheduler

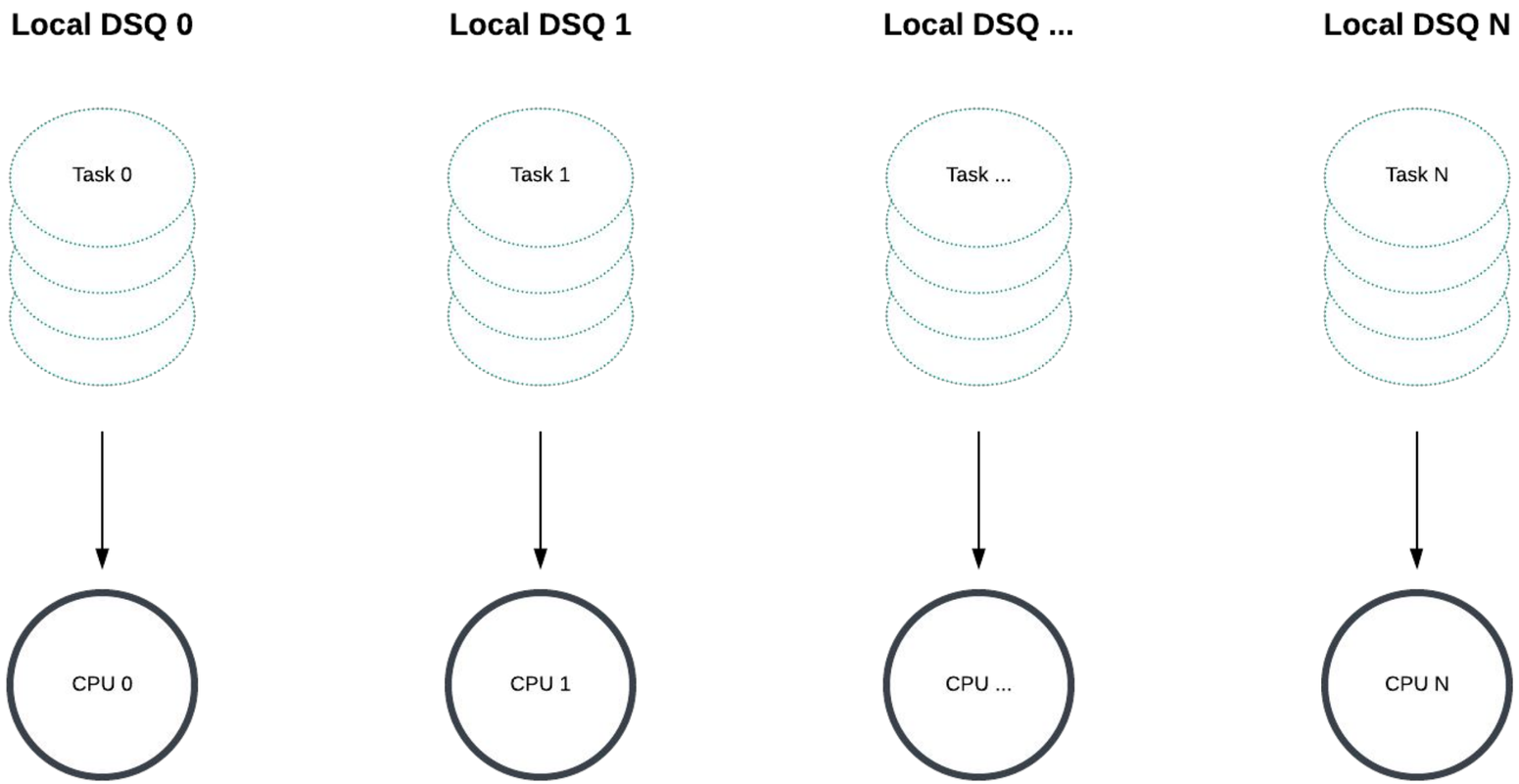
# Dispatch Queues (DSQs) are basic building block of scheduler policies

- Conceptually similar to runqueue
- Every core has a special “local” DSQ called SCX_DSQ_LOCAL
- Otherwise, can create as many or as few as needed
  - Gives schedulers flexibility
    - Per-domain (NUMA node, CCX, etc) DSQ?
    - Global DSQ?
    - Per-cgroup DSQ?
- The data structure / abstraction layer for managing tasks between main kernel <-> BPF scheduler (more on next slide).

```
/* Return CPU that task should be migrated to on wakeup path. */
s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);

/* Enqueue runnable task in the BPF scheduler. May dispatch directly to CPU. */
void (*enqueue)(struct task_struct *p, u64 enq_flags);

/* Complement to the above callback. */
void (*dequeue)(struct task_struct *p, u64 deq_flags);
...

/* Maximum time that task may be runnable before being run. Cannot exceed 30s. */
u32 timeout_ms;

/* BPF scheduler's name. Must be a valid name or the program will not load. */
char name[SCX_OPS_NAME_LEN];
```

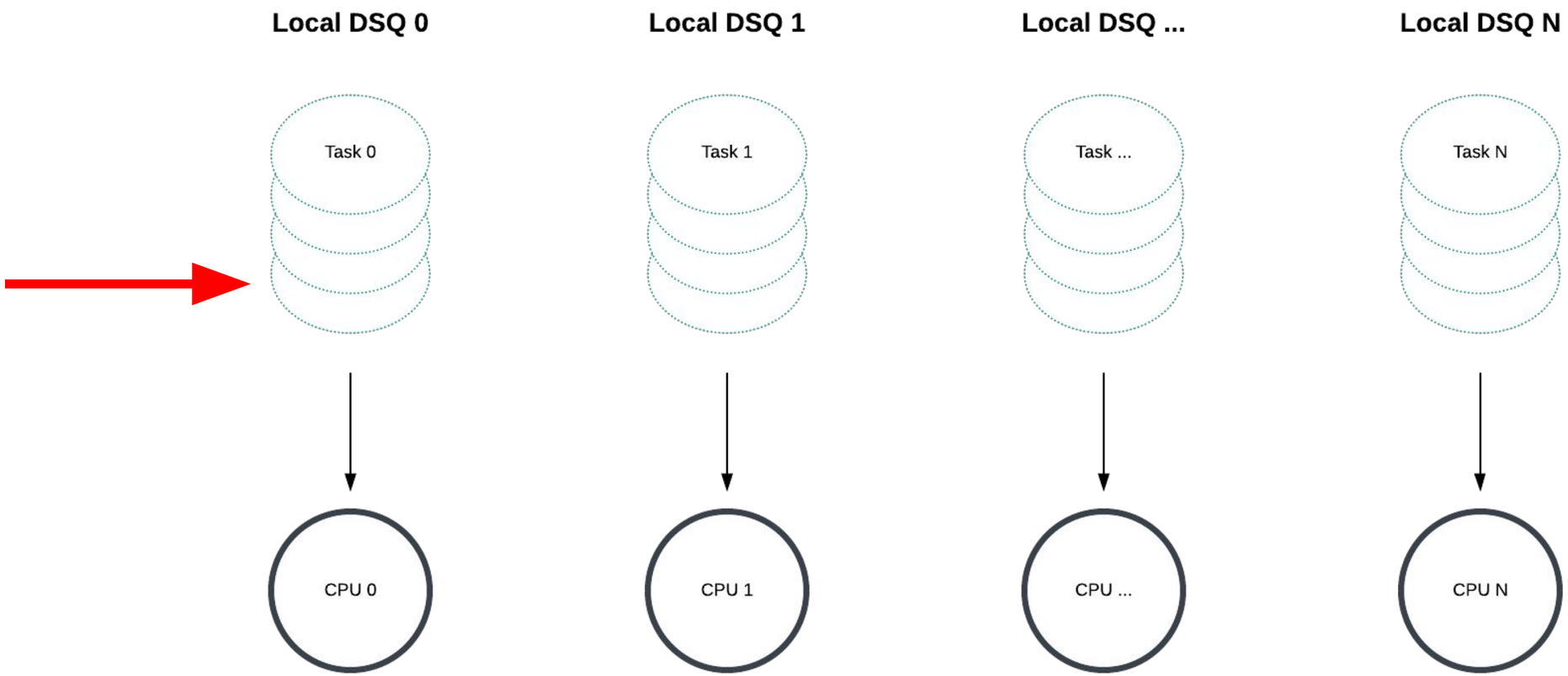
From https://github.com/sched-ext/sched_ext/blob/sched_ext/include/linux/sched/ext.h

# Local DSQs are per-CPU – the “runqueue” that the core kernel actually chooses from



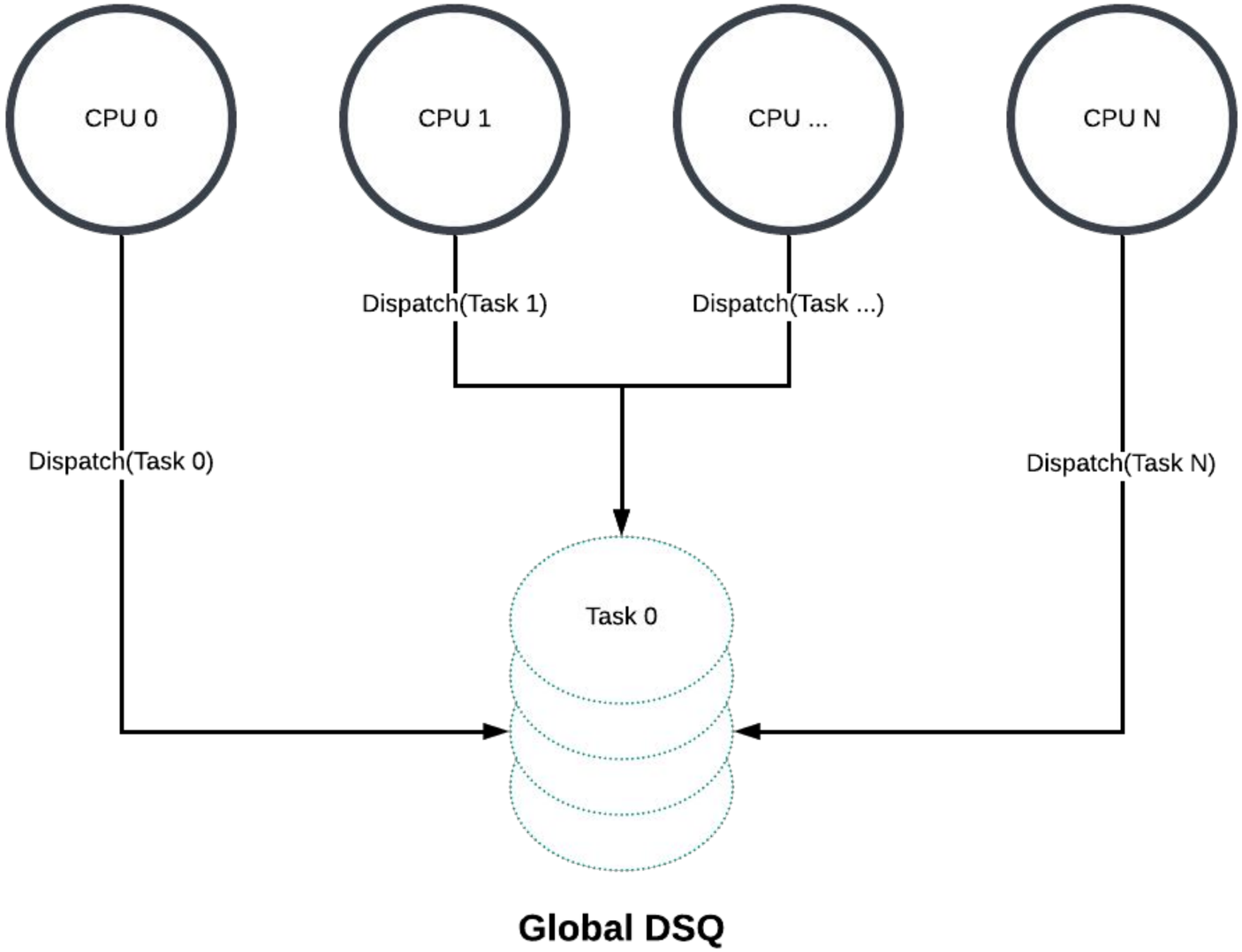
# Local DSQs are per-CPU – the “runqueue” that the core kernel actually chooses from

- FIFO or priority queue of tasks. “**dispatched**” (i.e. enqueued) from BPF
- What’s actually pulled from when a task is scheduled in



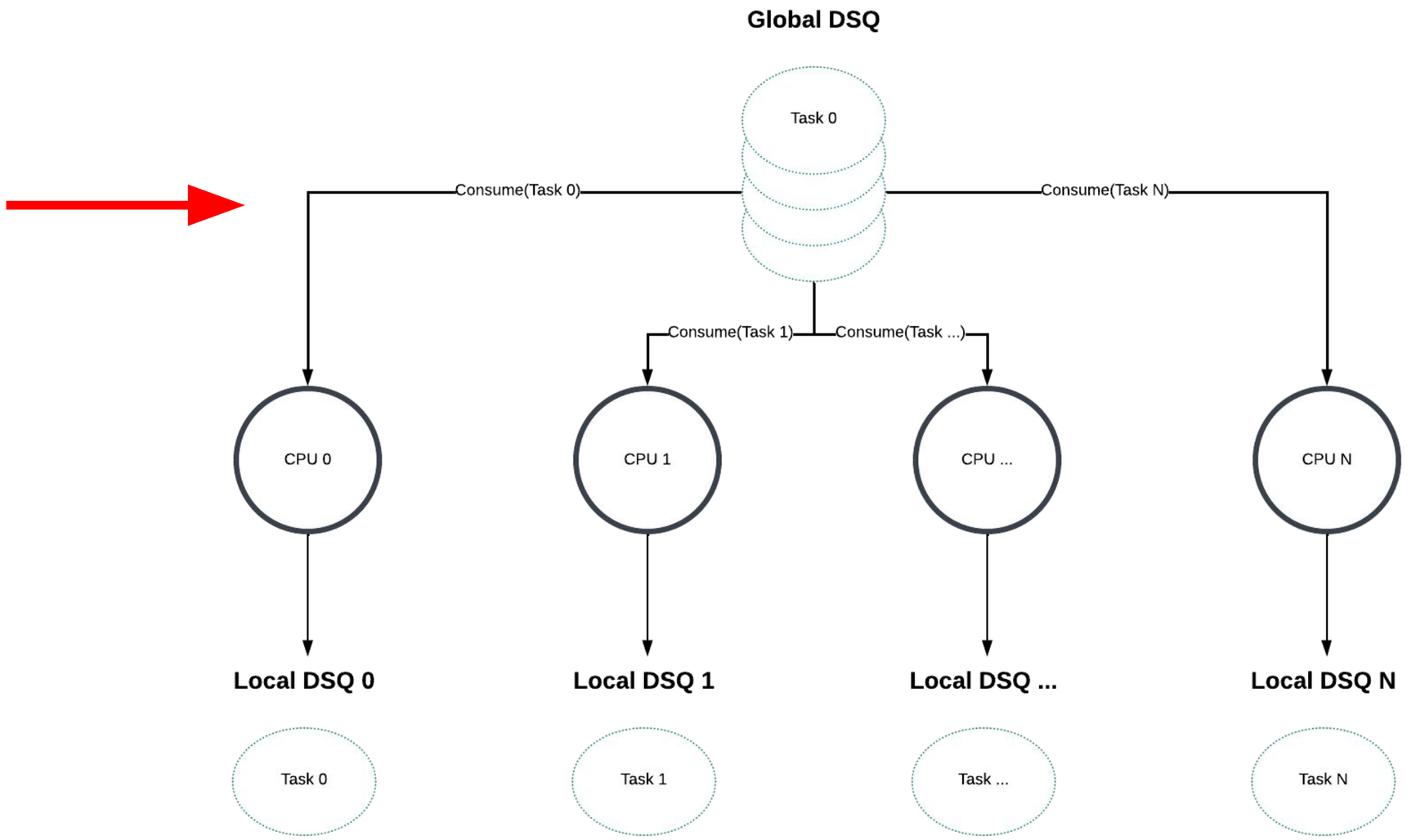
# Example 0: Global FIFO – enqueuing

- Scheduler “**dispatches**” tasks to global DSQ at enqueue time
  - *Not* where tasks are pulled from when being scheduled in
  - Task must be in local DSQ to be chosen to run



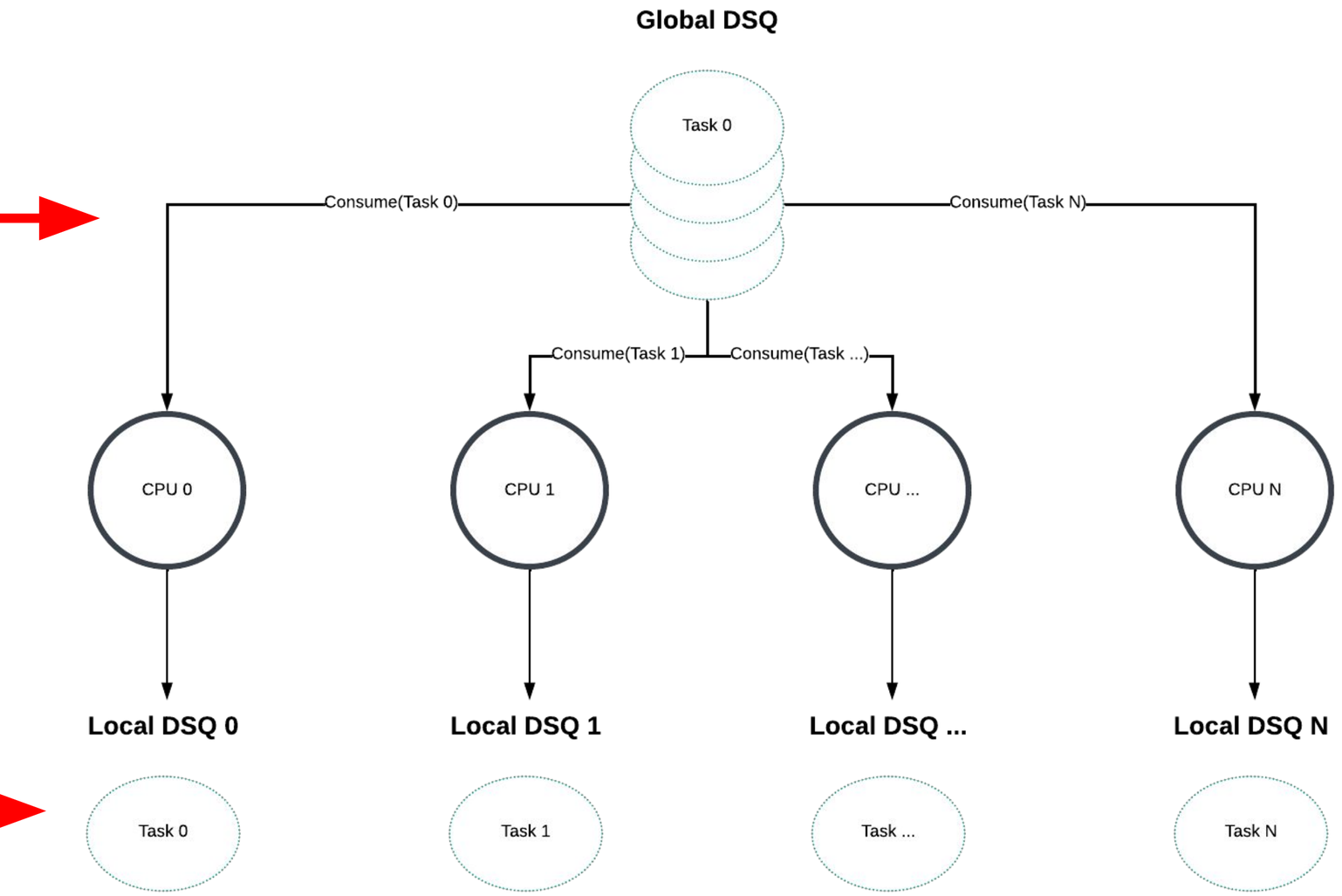
# Example 0: Global FIFO – consuming

- Cores “**consume**” tasks from the global DSQ when going idle (i.e. no tasks left in the core’s local DSQ)



# Example 0: Global FIFO – consuming

- Cores “**consume**” tasks from the global DSQ when going idle (i.e. no tasks left in the core’s local DSQ)




- And enqueue them on their local DSQ to be scheduled

# Global FIFO works surprisingly well on single socket / CCX machines

- Work conserving
- Essentially functions like the SHARED_RUNQ patchset mentioned earlier
- Very, very simple

```c
const volatile bool switch_partial; /* Can be set by user space before loading the program. */

s32 BPF_STRUCT_OPS(simple_init)
{
        if (!switch_partial) /* If set, tasks will individually be configured to use the SCHED_EXT class. */
                scx_bpf_switch_all(); /* Switch all CFS tasks to use sched_ext. */
        return 0;
}

void BPF_STRUCT_OPS(simple_enqueue, struct task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL) /* SCX_ENQ_LOCAL could be set if e.g. the current CPU has no other tasks to run. */
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, enq_flags);  /* Dispatch task to the head of the current CPU's local FIFO. */
        else
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL, enq_flags); /* Dispatch task to the global FIFO, it will be consumed
                                                                 * automatically by ext. */
}

void BPF_STRUCT_OPS(simple_exit, struct scx_exit_info *ei)
{
        bpf_printk("Exited"); /* Can do more complicated things here like setting flags in user space, etc. */
}

SEC(".struct_ops")
struct sched_ext_ops simple_ops = {
        .enqueue                = (void *)simple_enqueue,
        .init                   = (void *)simple_init,
        .exit                   = (void *)simple_exit,
        .name                   = "simple",
};
```

# Not pictured: selecting a core in ops.select_cpu() callback

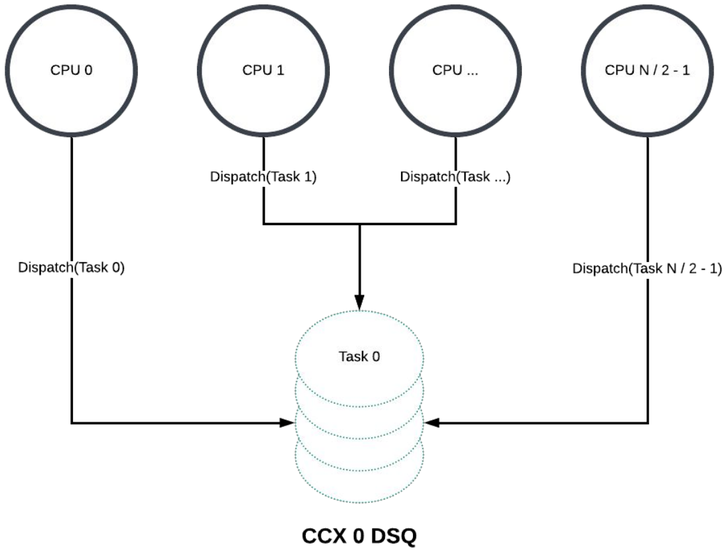
- Default implementation if not defined is to pick an idle core using the following priority order:
  - Waking core, if it would otherwise go idle
  - When SMT is enabled, a wholly idle core with no hypertwin running
  - Any idle CPU in the system

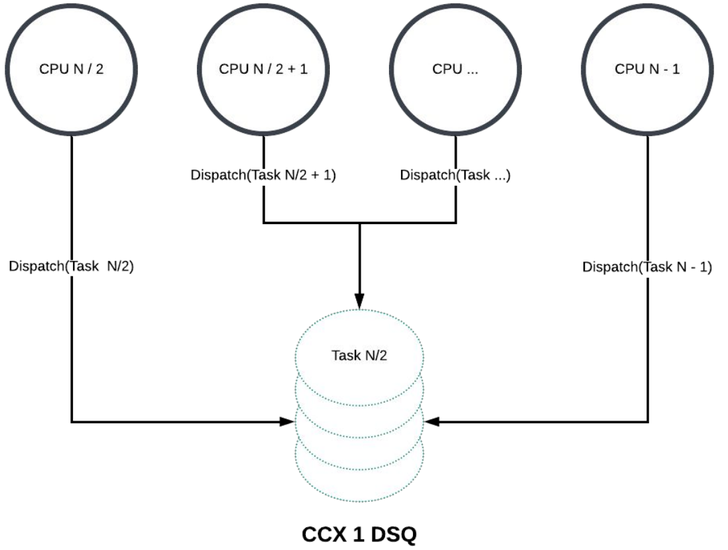
- If core is idle, an IPI is automatically sent to wake it up

- Whichever CPU is specified here is where the ops.enqueue() callback is eventually invoked

# Example 1: Per-CCX FIFO – enqueuing

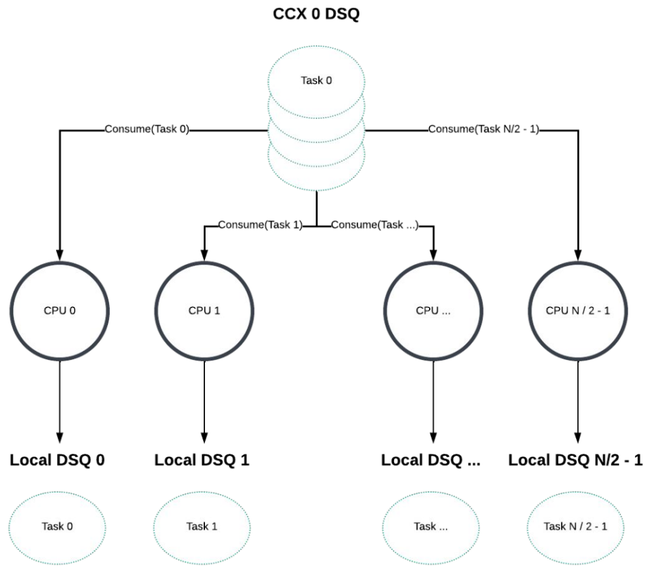
- Exact same idea as global FIFO, but on a per-CCX granularity
- A core “**dispatches**” tasks to the DSQ for its CCX / L3 cache
- And “**consumes**” from it when going to idle

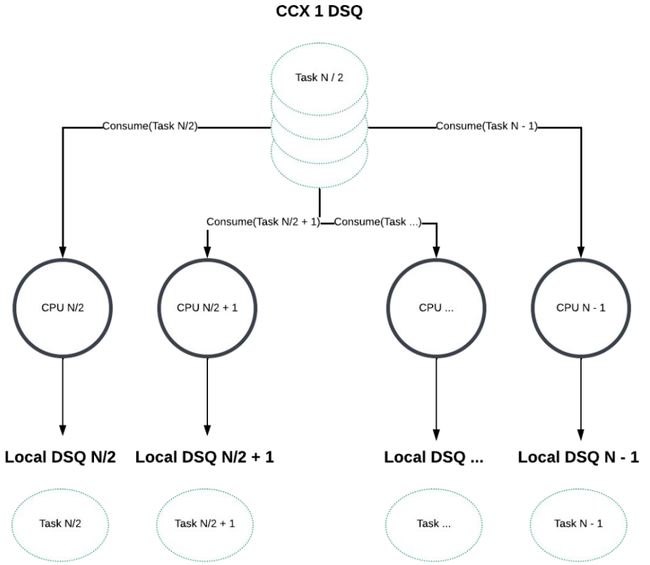


# Example 1: Per-CCX FIFO – consuming

- Cores pull from their CCX's DSQ
- Better L3 cache locality
- Unlike global FIFO, *not* work conserving
  - What if one CCX's DSQ runs out, but the other has work? Many possibilities
    - Always steal only if your CCX's DSQ is empty
    - Only steal if the other DSQ has X tasks enqueued
    - Only steal if user space marked the task as special and always steal-able?
    - …
    - Correct answer is: run experiments with sched_ext to see what works. Enable that feature as part of your scheduler (and then upstream it to CFS)



# 03 Example schedulers

**Meaning, schedulers we’re including with upstream patch set**


Meta

# First: production-ready schedulers

- These are schedulers which are usable in production environments
- Ready for prod, but they still have opportunities for improvement and more features

Rusty: https://github.com/sched-ext/sched_ext/tree/sched_ext/tools/sched_ext/scx_rusty

Simple: https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/scx_simple.bpf.c

Flatcg: https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/scx_flatcg.bpf.c

## scx_rusty

- Multi-domain BPF / user space hybrid scheduler
  - BPF portion is simple. Hot paths do round robin on each domain
  - User space portion written in rust. Contains more complex and substantial logic of load balancing, etc.
- Suitable for production workloads. Has parity with CFS on multi-domain (NUMA, CCX, etc) hosts for HHVM

## scx_simple

- Example scheduler showed earlier
- A simple weighted vtime / global FIFO
- About 200 lines total, including user space code, stats collection, etc.
- May not always be suitable for production
  - Only performant on single-socket, uniform L3 cache architectures

## scx_flatcg

- Flattened cgroup hierarchy scheduler
- Implements performant, hierarchical weight-based cgroup CPU control by flattening cgroup hierarchy
- Vulnerable to cgroup thundering herd inaccuracies
  - If many low-pri cgroups wake at the same time, they may get excess of CPU

# Next: example schedulers

- Not meant to be used in production environments (yet)
- Used to illustrate various sched_ext features
  - Can be forked to create your own, or improved upon and made production worthy

Qmap: https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/scx_qmap.bpf.c

Central:

Pair:

Userland:

## scx_qmap

- Simple five-level FIFO queue scheduler
- Slightly more complex than `scx_simple`, still very simple
- Has no practical use, just useful for demonstrating features in a simple way
- About 500 lines total, including comments, user space, stats collection, etc

## scx_central

- A “central” scheduler making (almost) all scheduling decisions from a single CPU, on a tickless system
- Possibly useful for workloads that could benefit from fewer timer interrupts or less scheduling overhead
  - VMs / cloud environment
- Not usable for production in its current form
  - Not NUMA aware, resched IPIs sent every 20ms

## scx_pair

- Demo scheduler that only schedules tasks in the same cgroup on a sibling CPU pair
- Doesn’t have any priority handling inside or across cgroups. Would need to be added to be practically useful
- Example of what could have been a stop-gap solution for L1TF before core scheduling was merged

## scx_userland

- Simple vtime scheduler that makes all scheduling decisions in user space
- Not production ready — uses an ordered list for vtime, not NUMA aware

# Minimum system requirements

- Kernel compiled from repo (https://github.com/sched-ext/sched_ext)
- .config options enabled:
  - CONFIG_SCHED_CLASS_EXT=y
  - CONFIG_DEBUG_INFO_BTF=y
  - CONFIG_BPF=y
  - CONFIG_BPF_SYSCALL=y
  - CONFIG_BPF_JIT=y
  - ...and any dependencies
- clang >= 16.0.0
  - gcc support hopefully coming soon, but it doesn't yet fully support BPF in general
- pahole >= 1.24
- rustup nightly (if you want to compile the `scx_rusty` scheduler)
- See https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/README for more information

# 04 Current status and future plans

# Upstream first philosophy

- Developers need to first merge bug fixes or features upstream before using it internally.
- General workflow is typically:

1. Debug issue and/or write patches, send upstream
2. Iterate with upstream community until patches are merged
3. Backport to Meta kernel(s)

- Allows us to follow latest upstream kernel closely (rolling out 6.4 to production now-v)

# Top priority for sched_ext is upstreaming it

- Still iterating with members of the upstream community and incorporating feedback
  - Challenging to get engagement, but we are committed to getting sched_ext upstreamed as long as it takes
- Latest v4 patch set (https://lore.kernel.org/all/20230711011412.100319-1-tj@kernel.org/):
  - New example schedulers (scx_flatcg, overhauled rusty)
  - Google committed to building ghOSt on top of sched_ext [0]
  - Manipulating and querying cpumasks directly from BPF (`struct bpf_cpumask *`)
  - Adding rbtree / priority queue semantics to DSQs
  - `ops.set_weight()` callback added to allow schedulers to lazily track weight changes
  - Using new BPF iterator feature instead of `bpf_loop()`
  - Lots of bug fixes

[0]: https://lore.kernel.org/all/CABk29Nt_iCv=2nbDUqFHnszMmDYNC7xEm1nNQXibnPKUxhsN_g@mail.gmail.com/

# New features

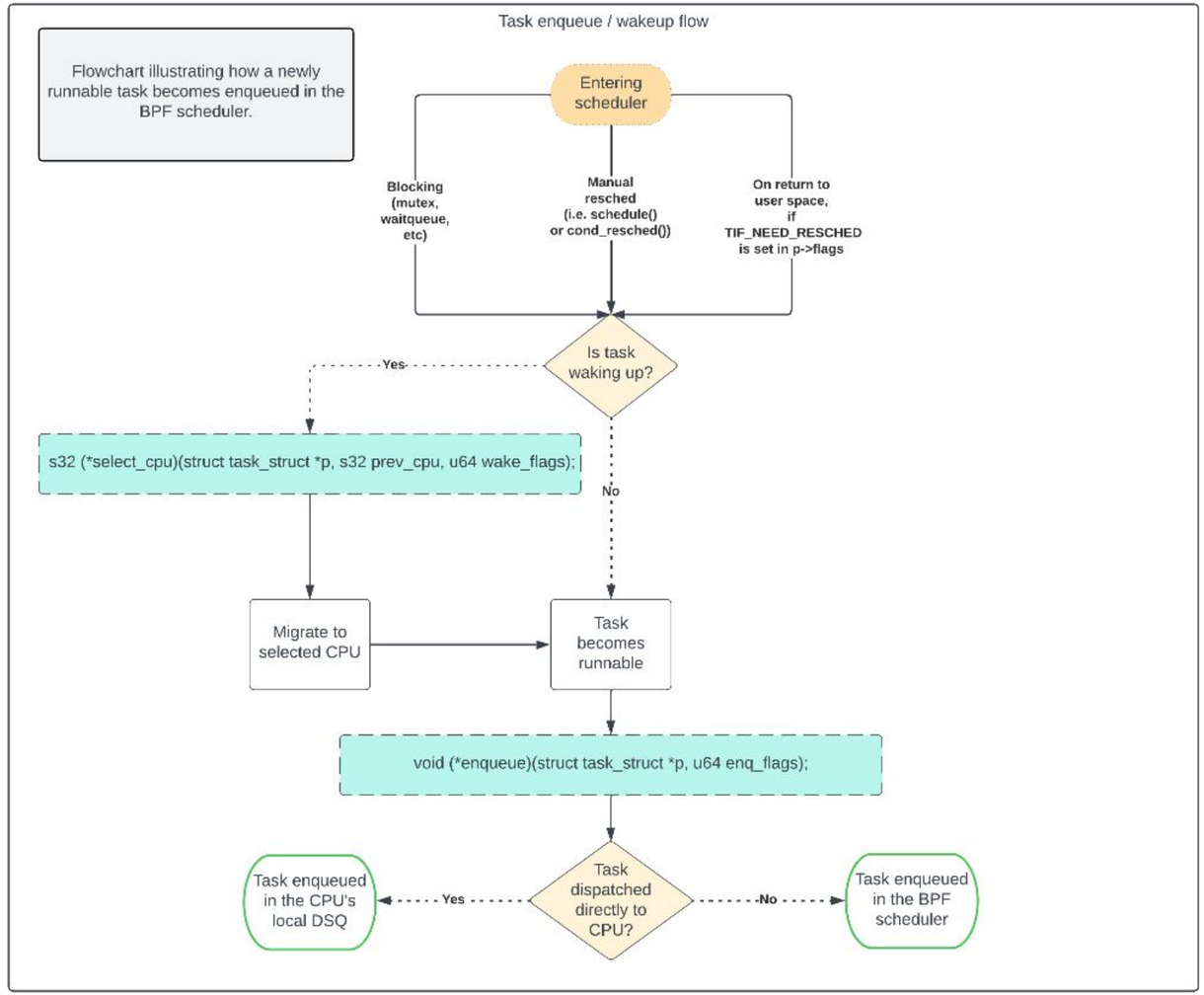
- Not much planned at the moment in terms of more sched_ext features. Mostly BPF (described below)
  - Would prefer to see what people need before adding more complexity
  - Currently rolling out to production at Meta
- More example / upstreamed schedulers
  - Power-aware
  - Latency nice
- Adding new BPF features
  - “Polymorphic” kfuncs — allowing BPF progs to call the same kfunc symbol, but have it be resolved to different implementation depending on context
  - Nested struct_ops
    - Enable different policies to be used on different partitions of a host
  - Calling into kfuncs with `struct bpf_spin_lock` held
  - Using assertions to simplify logic to appease verifier

# Links

- Main repo: https://github.com/sched-ext/sched_ext
- Latest upstream patch set (v4): https://lore.kernel.org/all/20230711011412.100319-1-tj@kernel.org/
- Example schedulers: https://github.com/sched-ext/sched_ext/tree/sched_ext/tools/sched_ext
- Example scheduler descriptions and build instructions: https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/README
- sched_ext documentation: https://github.com/sched-ext/sched_ext/blob/sched_ext/Documentation/scheduler/sched-ext.rst

# Appendix – useful supplementary info

# Appendix


Task enqueue / wakeup flow
Flowchart illustrating how a newly runnable task becomes enqueued in the BPF scheduler.
Entering scheduler
Blocking (mutex, waitqueue, etc)
Manual resched (i.e. schedule() or cond_resched())
On return to user space, if TIF_NEED_RESCHED is set in p->flags
Is task waking up?
Yes
No
s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);
Migrate to selected CPU
Task becomes runnable
void (*enqueue)(struct task_struct *p, u64 enq_flags);
Task dispatched directly to CPU?
Yes
No
Task enqueued in the CPU's local DSQ
Task enqueued in the BPF scheduler

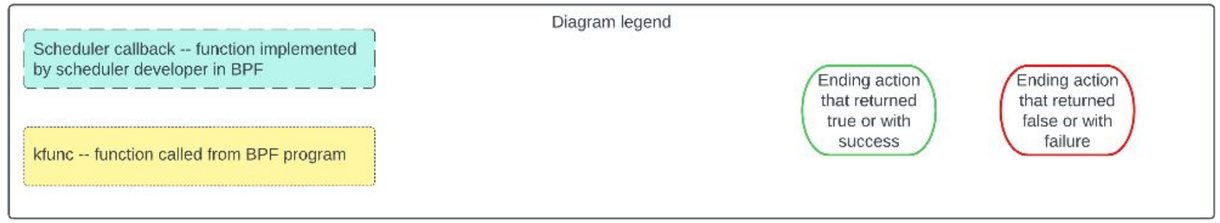
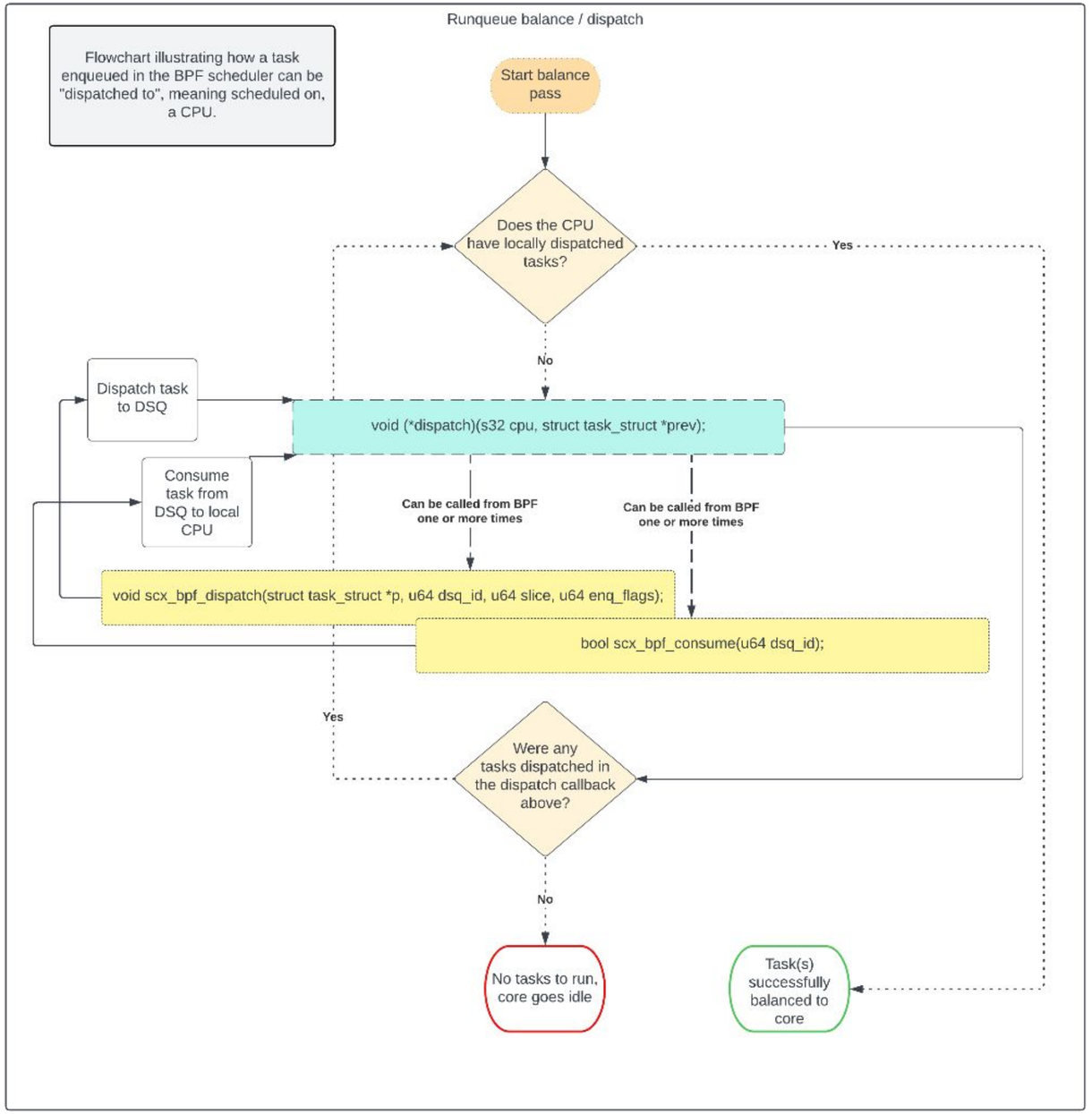


Diagram legend
Scheduler callback -- function implemented by scheduler developer in BPF
kfunc -- function called from BPF program
Ending action that returned true or with success
Ending action that returned false or with failure


Runqueue balance / dispatch
Flowchart illustrating how a task enqueued in the BPF scheduler can be "dispatched to", meaning scheduled on, a CPU.
Start balance pass
Does the CPU have locally dispatched tasks?
Yes
No
Dispatch task to DSQ
void (*dispatch)(s32 cpu, struct task_struct *prev);
Consume task from DSQ to local CPU
Can be called from BPF one or more times
Can be called from BPF one or more times
void scx_bpf_dispatch(struct task_struct *p, u64 dsq_id, u64 slice, u64 enq_flags);
bool scx_bpf_consume(u64 dsq_id);
Yes
Were any tasks dispatched in the dispatch callback above?
No
No tasks to run, core goes idle
Task(s) successfully balanced to core