Sched Ext

The extensible sched_class

Dan Schatzberg
Research Scientist

Meta
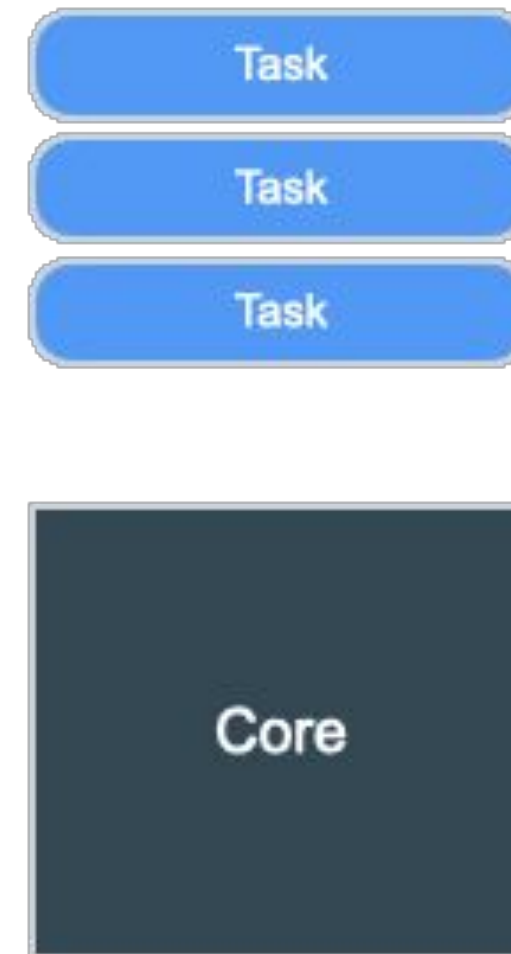
Agenda

01 Background and motivation
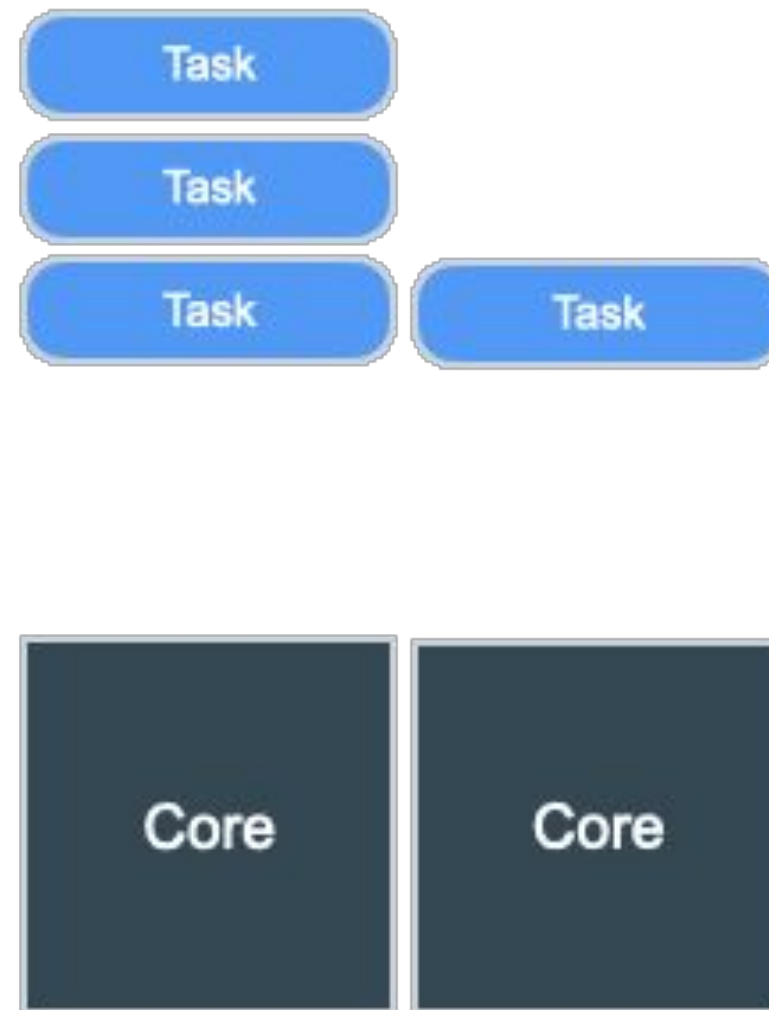
What is a CPU scheduler?

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

# 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 Completely Fair Scheduler

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
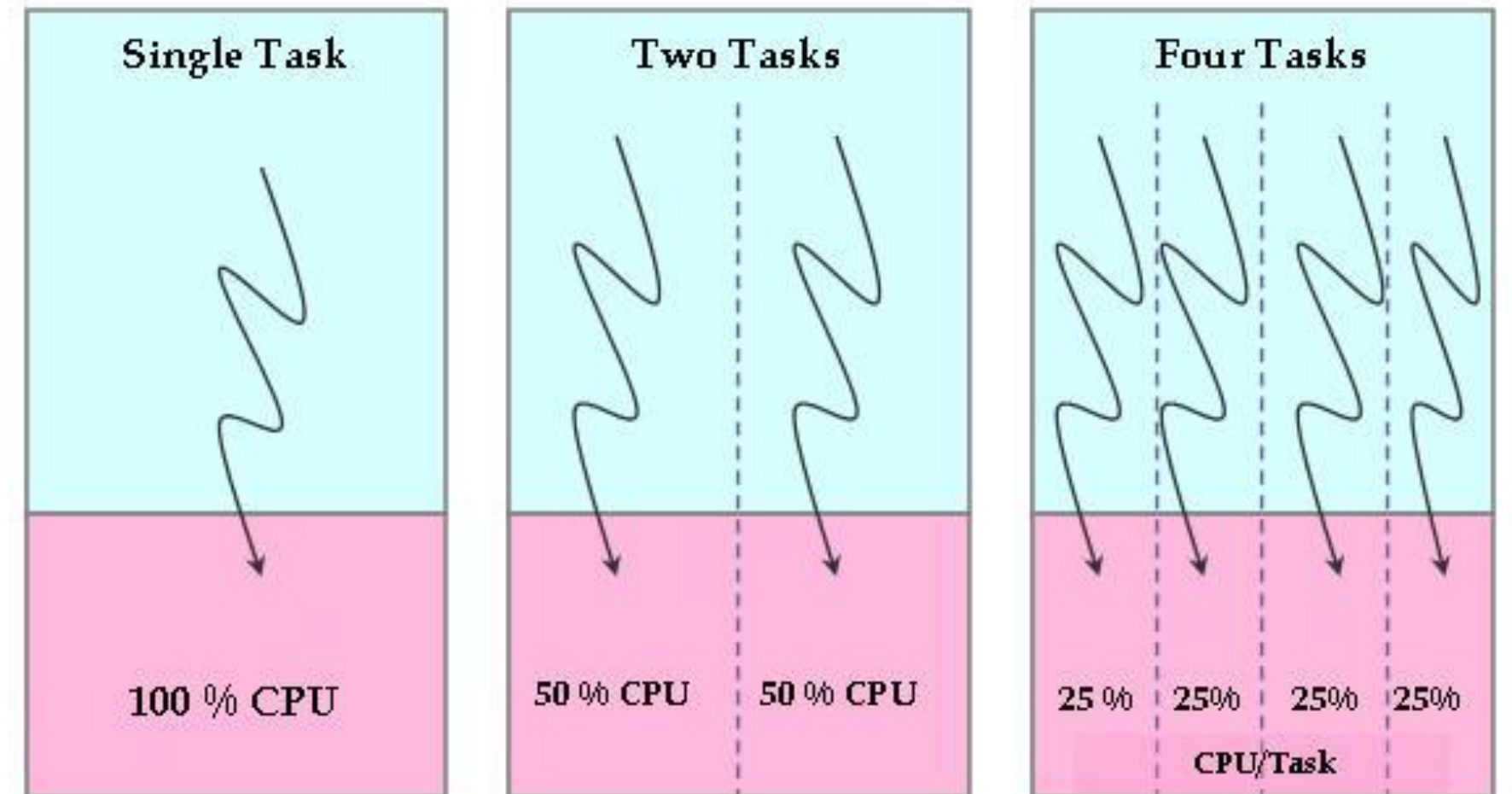
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

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
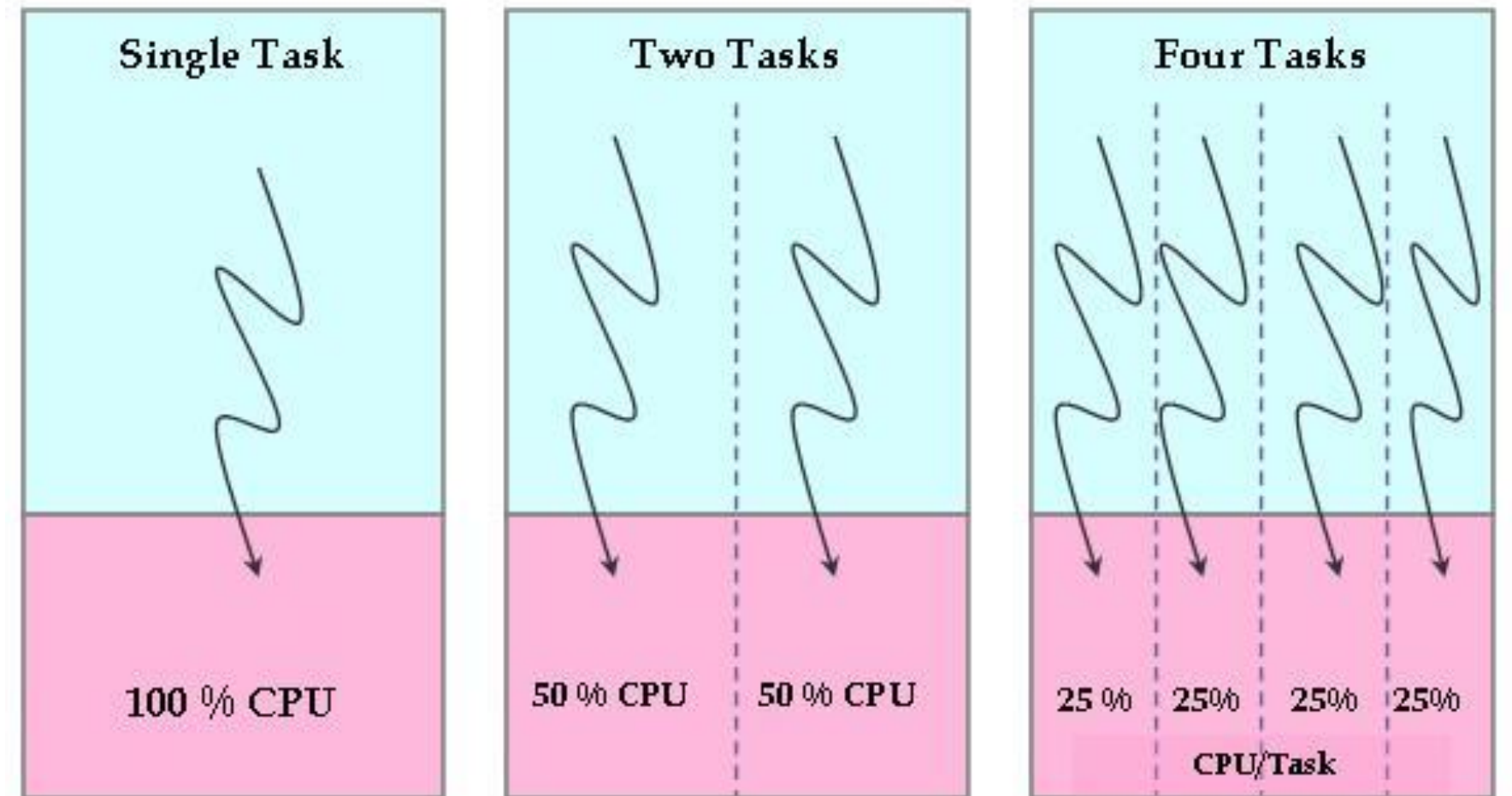 - Also has drawbacks, more on this later

Conceptual ➡️

Single Task	Two Tasks		Four Tasks			
100 % CPU	50 % CPU	50 % CPU	25 %	25%	25%	25%
				CPU/Task		

Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

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
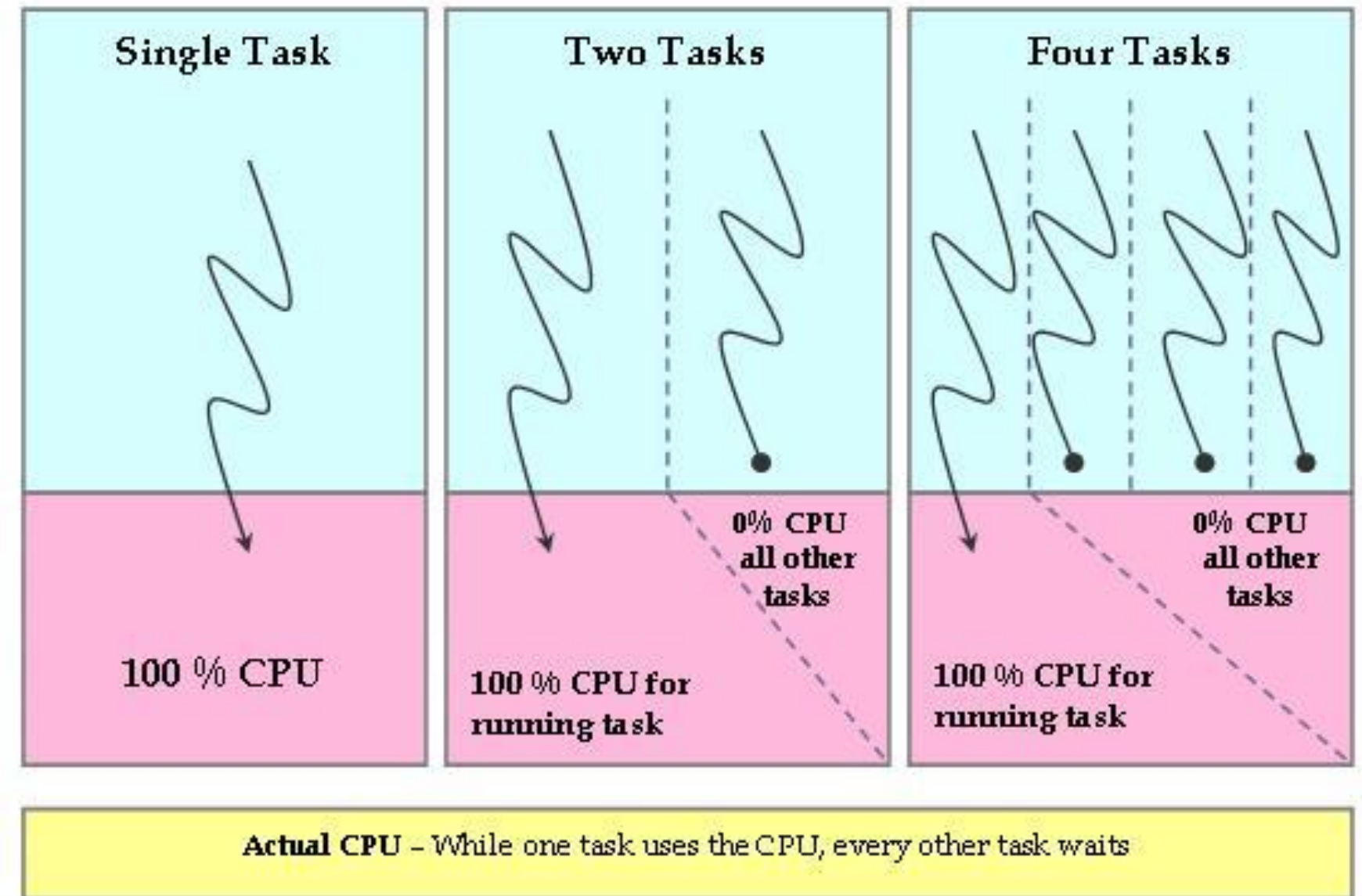 - Also has drawbacks, more on this later

Actual →

Single Task	Two Tasks	Four Tasks
	0% CPU all other tasks	0% CPU all other tasks
100 % CPU	100 % CPU for running task	100 % CPU for running task

Actual CPU – While one task uses the CPU, every other task waits

CFS has been in the kernel since 2007

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

Just one L3 cache

Just two cores
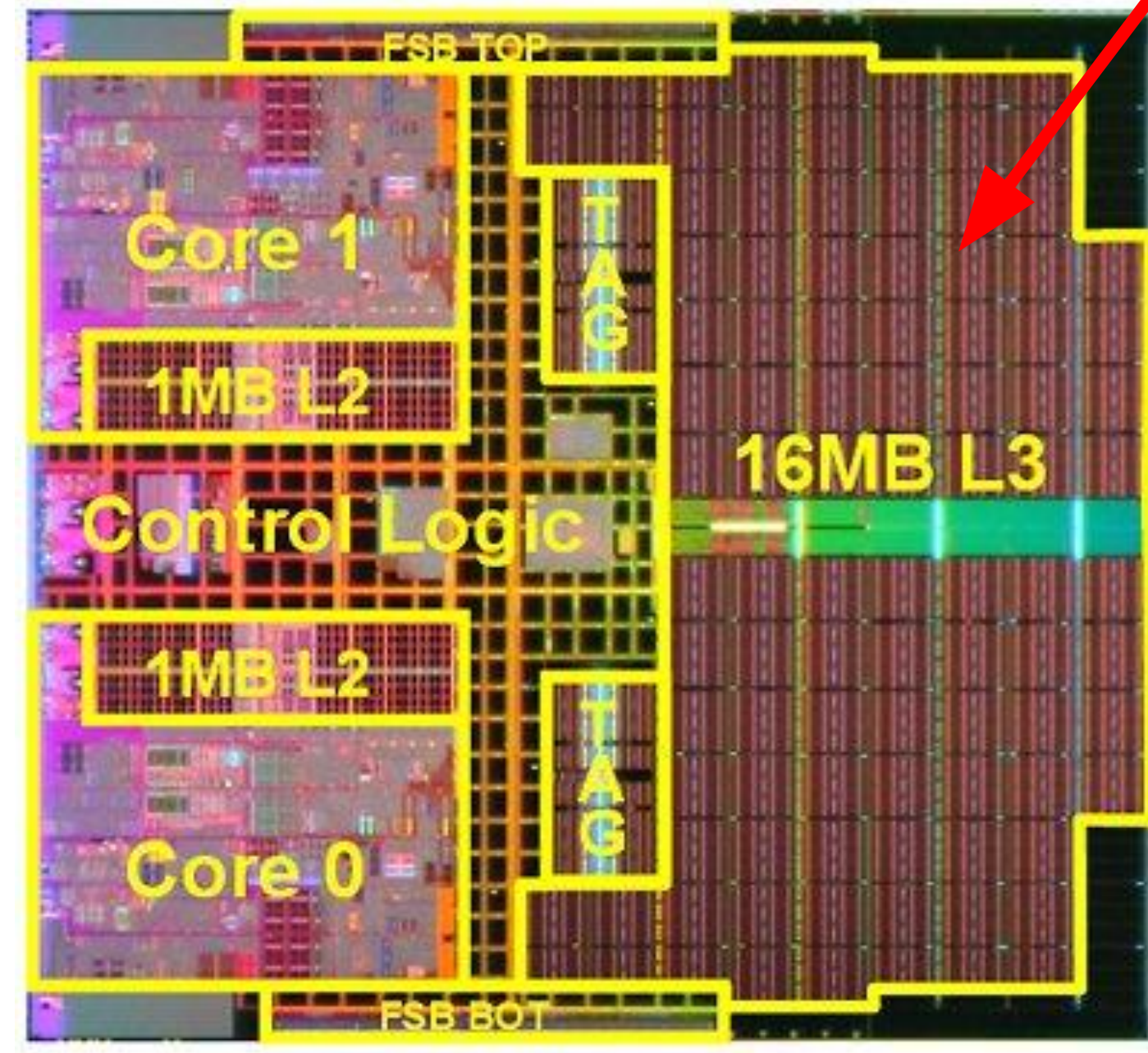
Intel Xeon MP 71xx die

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
- 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
- Write C code, compile it to BPF bytecode, userspace can load it into the kernel
- 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 disables sched_ext scheduler if a runnable task isn't scheduled within some timeout
  - New sysrq key for disabling sched_ext scheduler through console
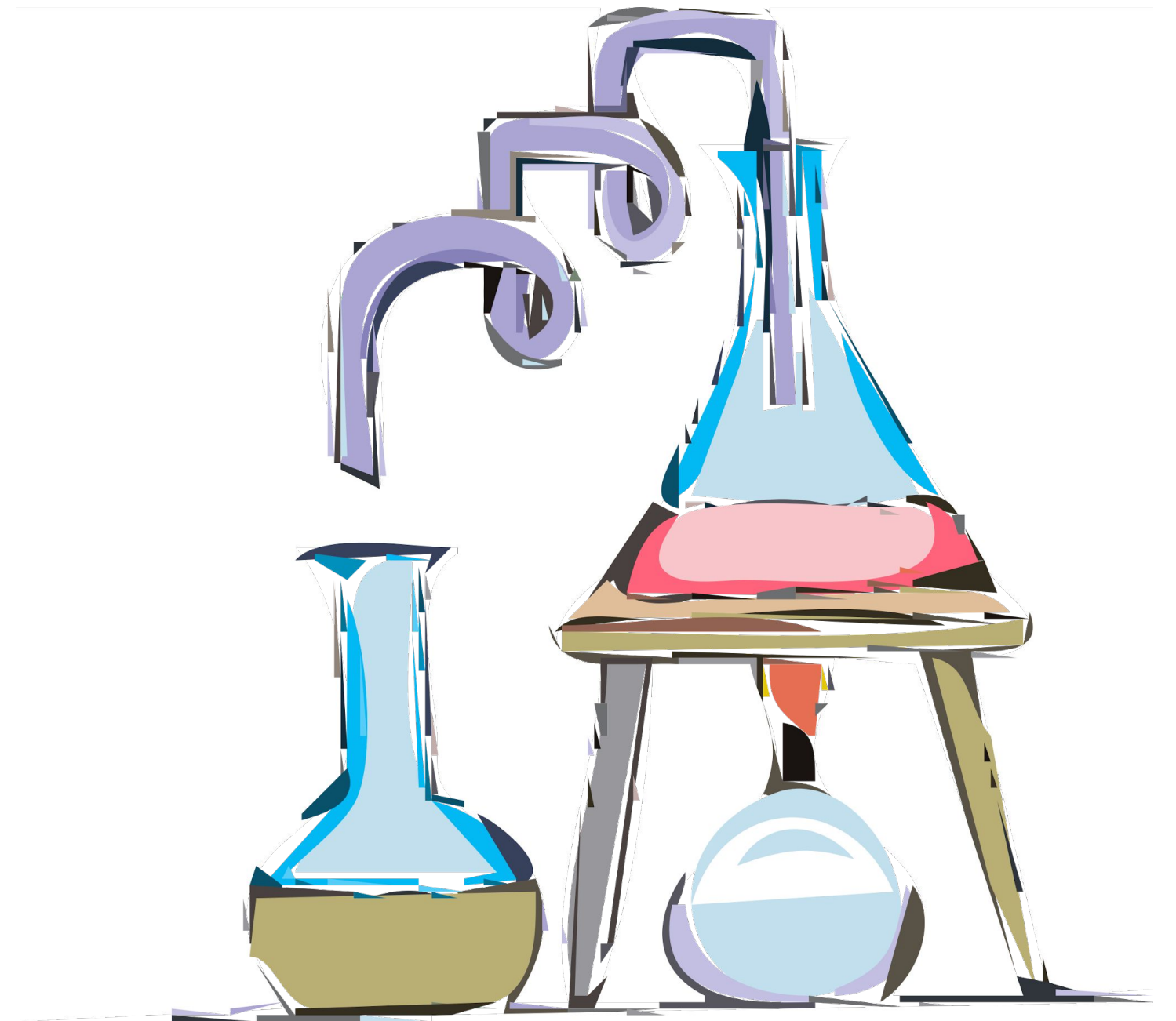
Bespoke scheduling policies

- 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

User space

- Offload complicated logic such as load balancing to user space
- Use of floating point numbers
- Use standard debugging tools
- BPF makes it easy to share data between the kernel and user space
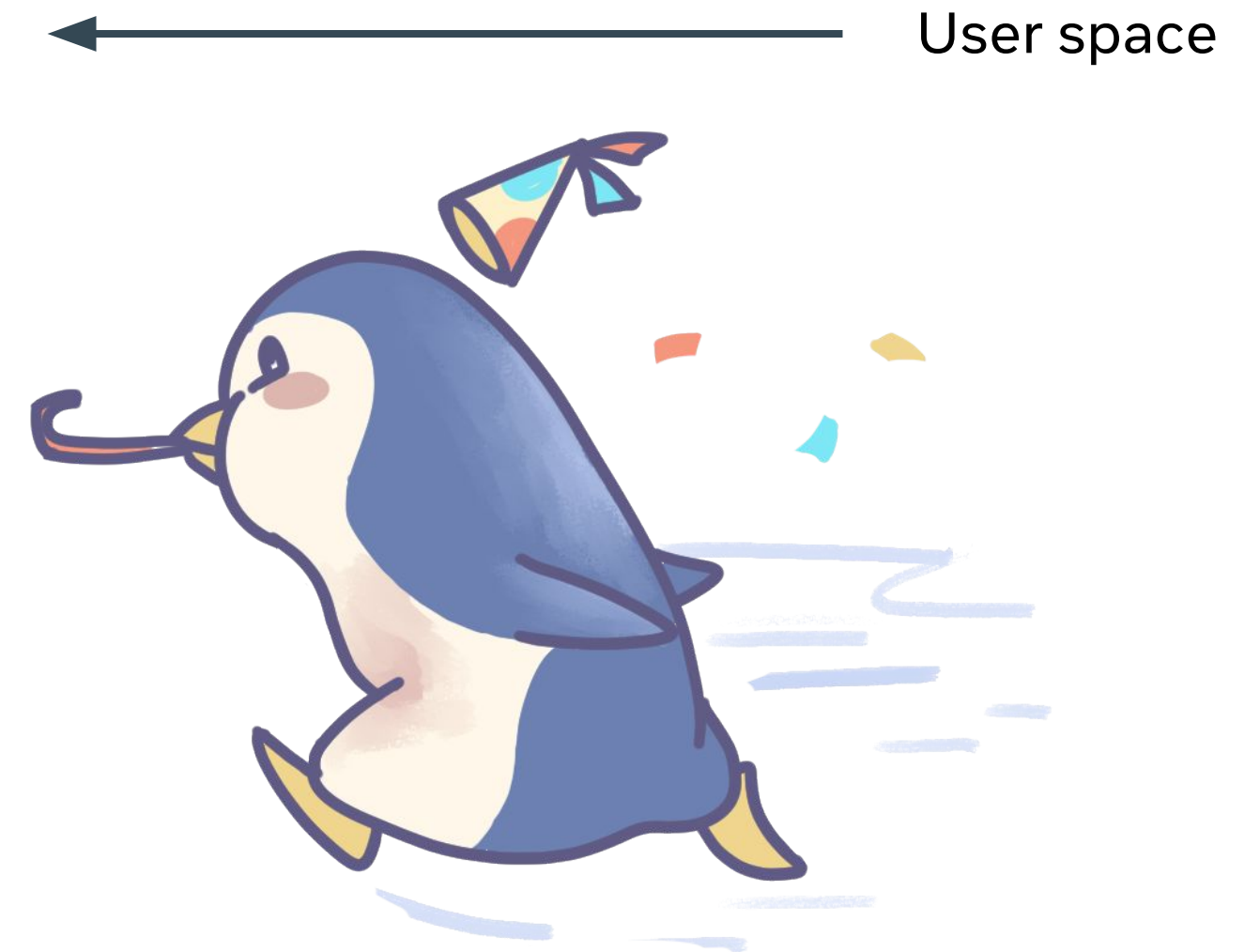
Sharing Scheduling Logic

- No need to maintain out of tree scheduler patches
    - Meta and many other companies need to do this currently
- Share BPF schedulers instead
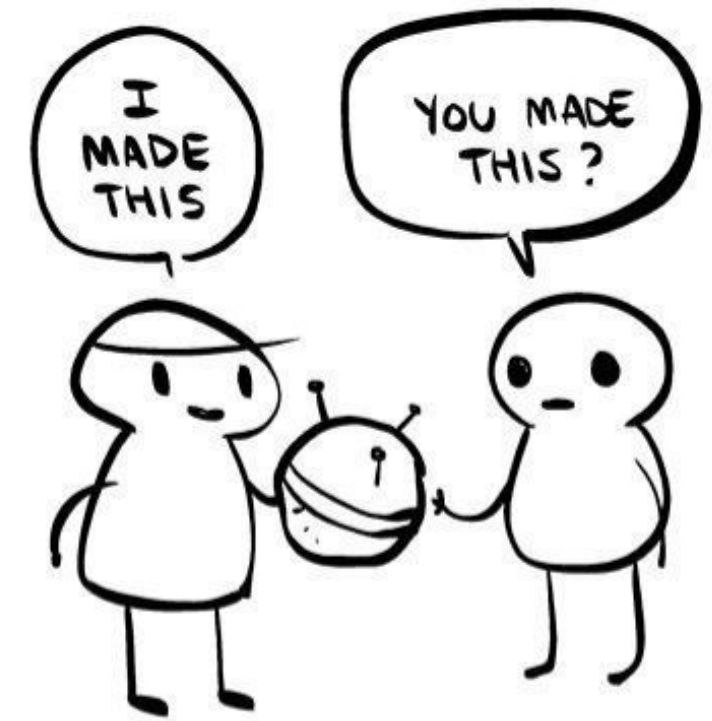
What is sched_ext *not*?

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

Implementing scheduling policies

- BPF program must implement a set of **callbacks**
  - Task wakeup
  - 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

```
/* 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);

/* Called when the CPU has no tasks to run */
void (*dispatch)(s32 cpu, struct task_struct *prev);
...

/* 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

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

- 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).

Local DSQs are per-CPU – the "runqueue" that the core kernel actually chooses from

Local DSQ 0

Task 0

↓

CPU 0

Local DSQ 1

Task 1

↓

CPU 1

Local DSQ ...

Task ...

↓

CPU ...

Local DSQ N

Task N

↓

CPU N
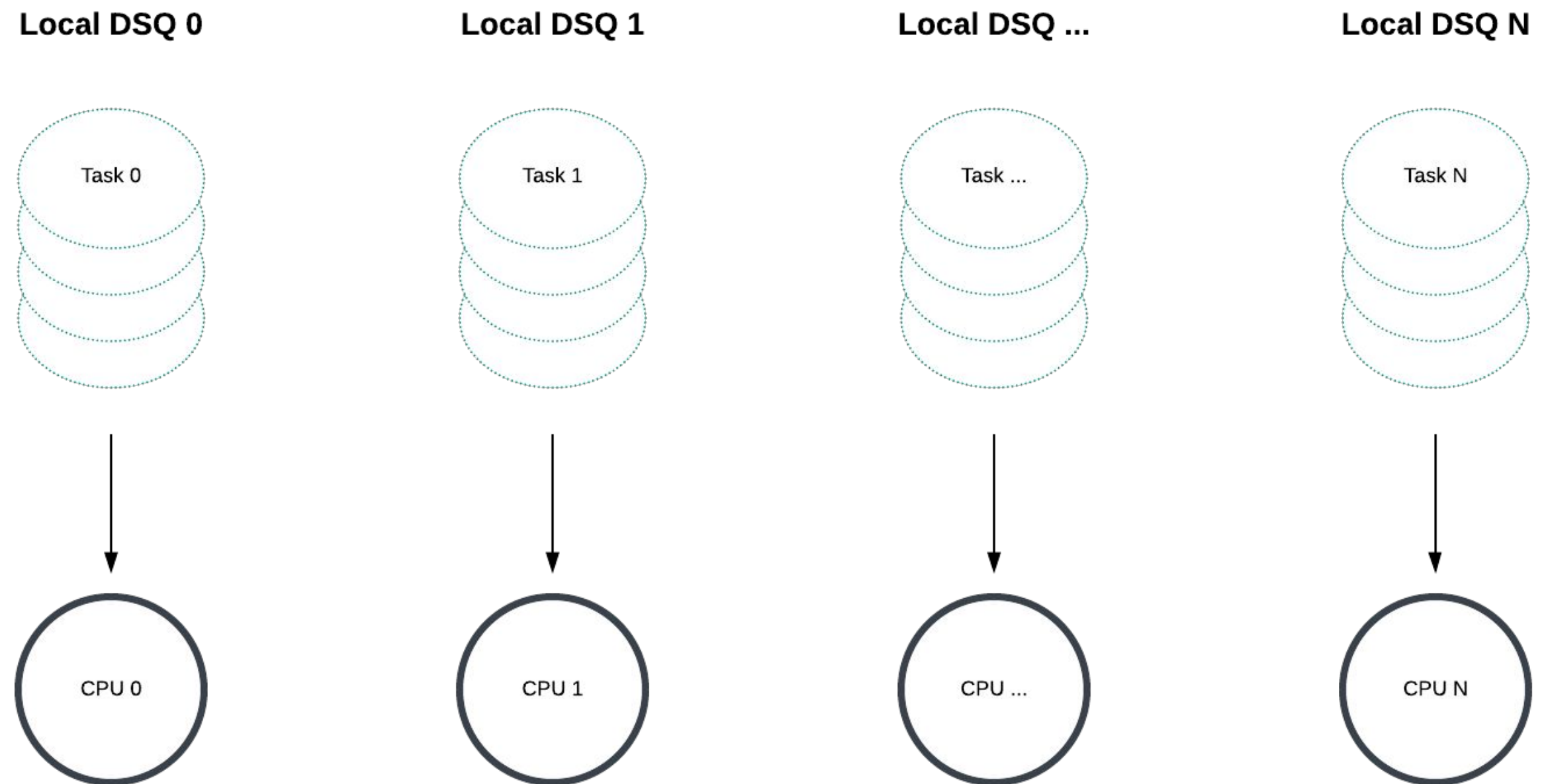
Local DSQs are per-CPU – the "runqueue" that the core kernel actually chooses from
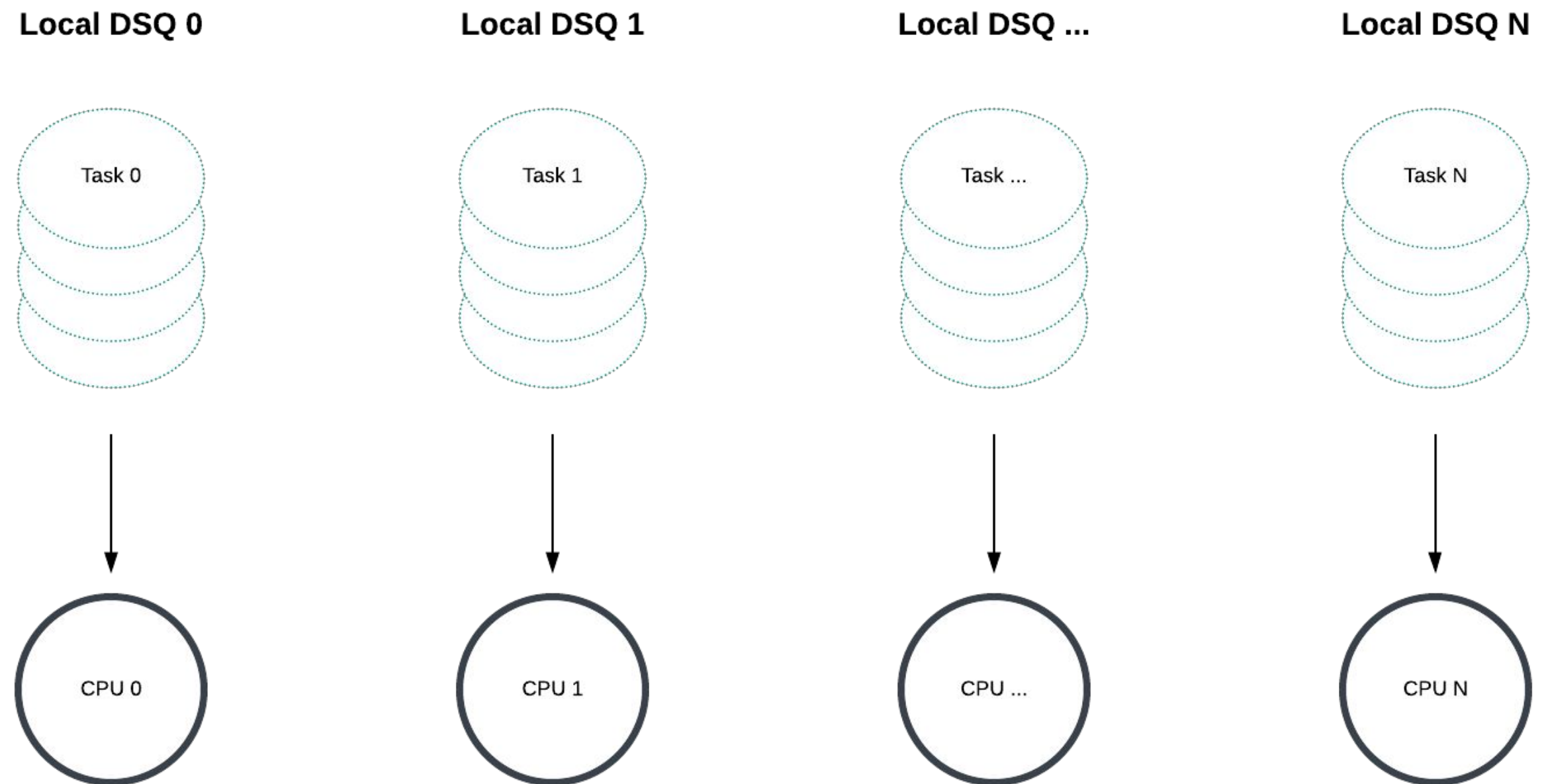
Local DSQs are per-CPU – the "runqueue" that the core kernel actually chooses from

- FIFO or priority queue of tasks. **"dispatched"** (i.e. enqueued) from BPF

- What's actually pulled from when a task is scheduled in
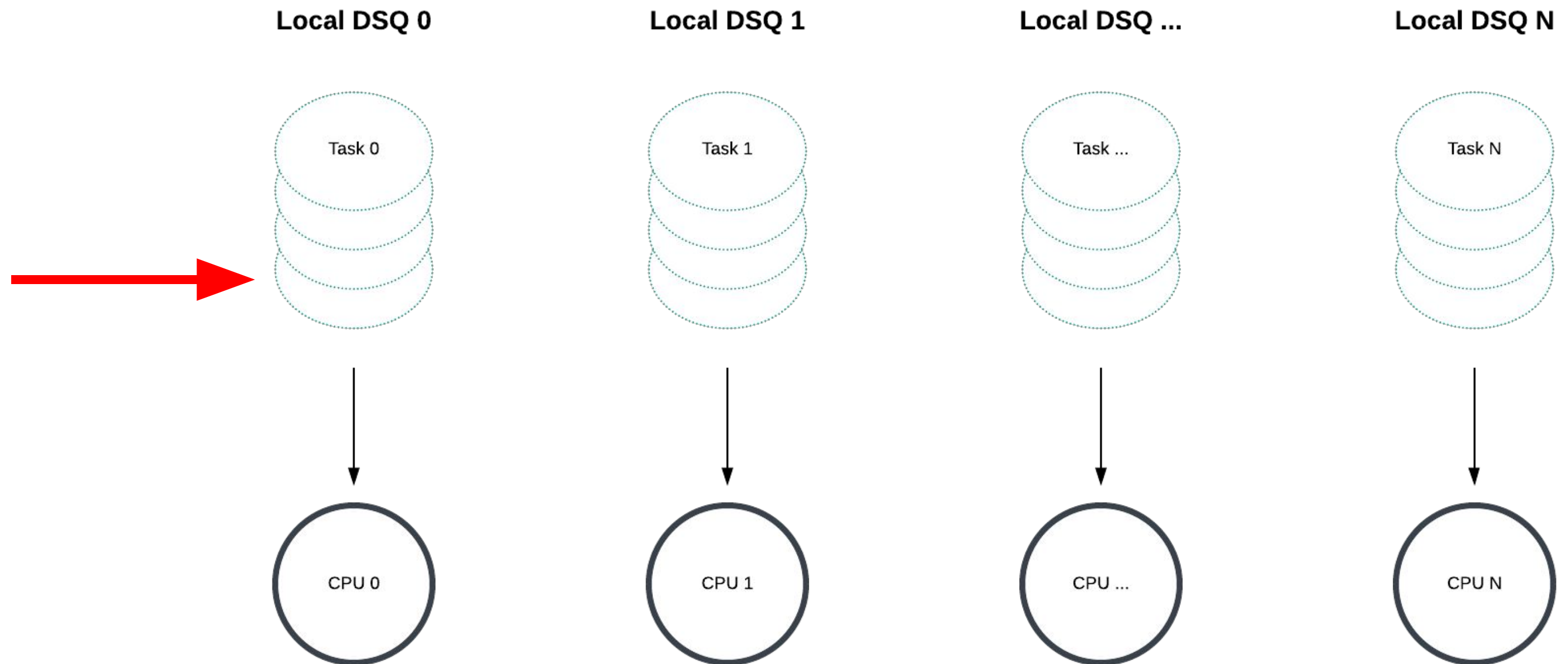
Example 0: Global FIFO – enqueuing

- 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
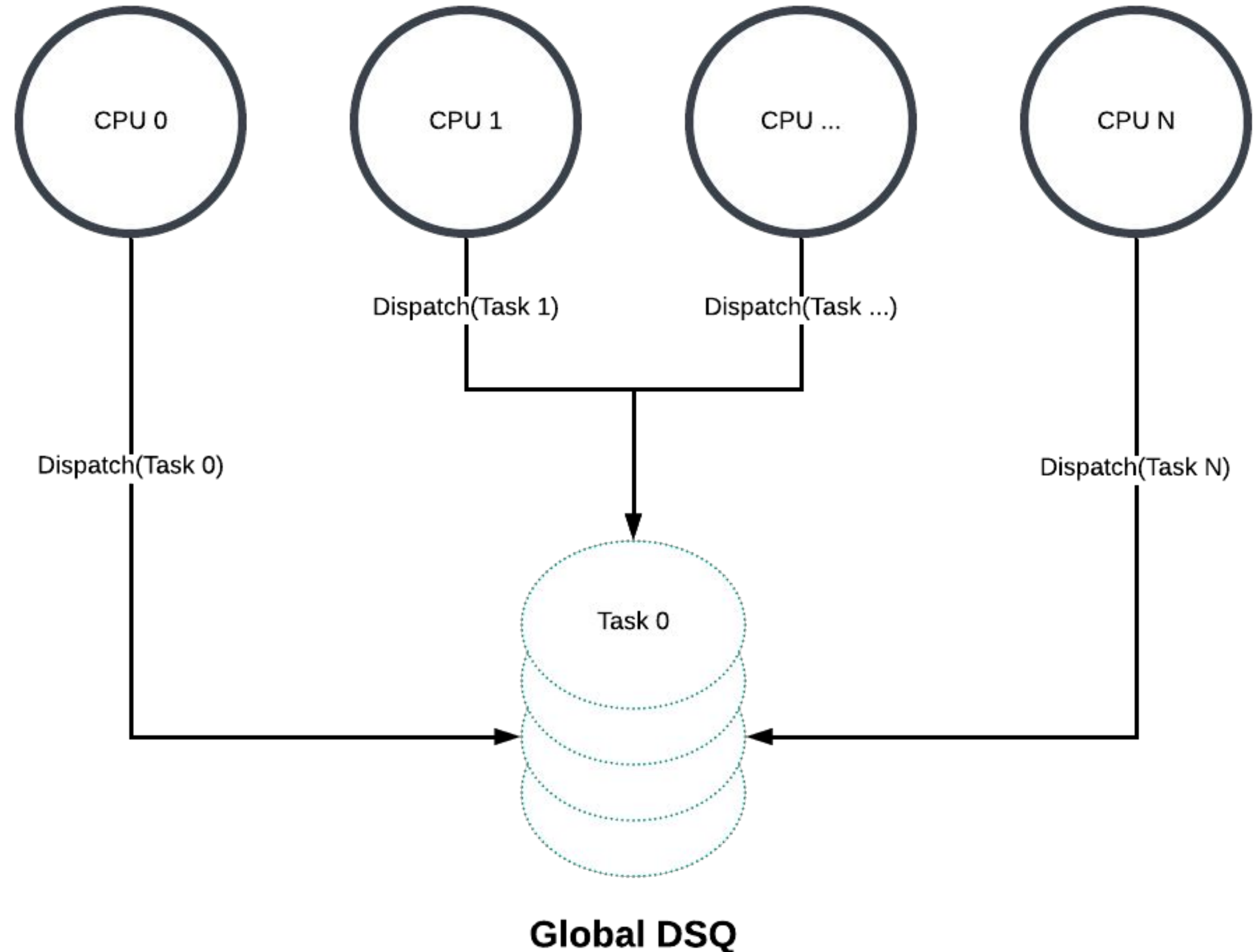
Example 0: Global FIFO – enqueuing

```
s32 BPF_STRUCT_OPS_SLEEPABLE(mysched_init)
{
    scx_bpf_create_dsq(/* queue id = */ 0, -1);
}



void BPF_STRUCT_OPS(mysched_enqueue, struct task_struct *p, u64 enq_flags)
{
    scx_bpf_dispatch(p, /* queue id = */ 0, SCX_SLICE_DFL, enq_flags);
}
```

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)
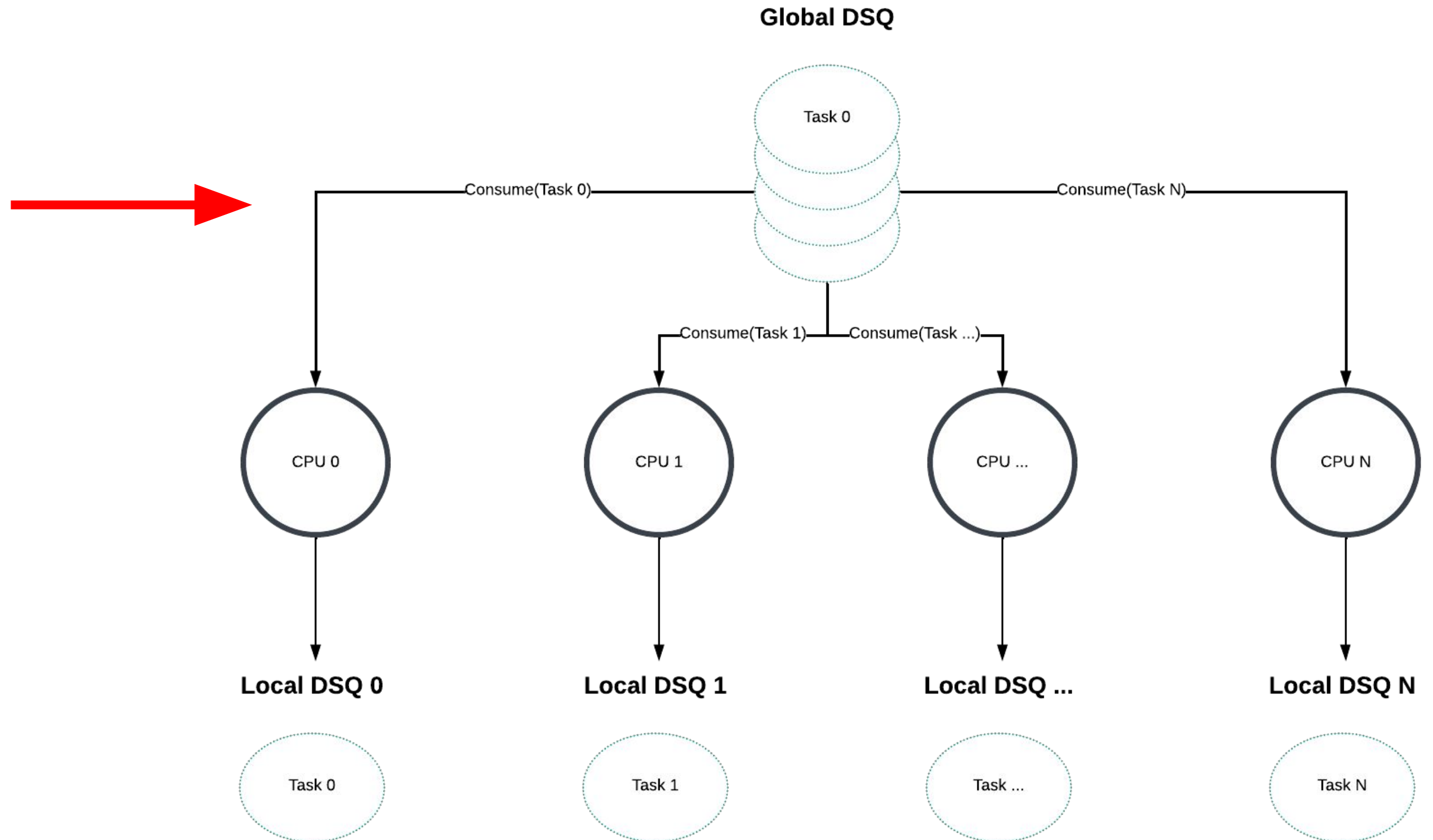
Example 0: Global FIFO – consuming

- Cores **"consume"** tasks from the global DSQ when going idle (i.e. no tasks left in the core's local DSQ)
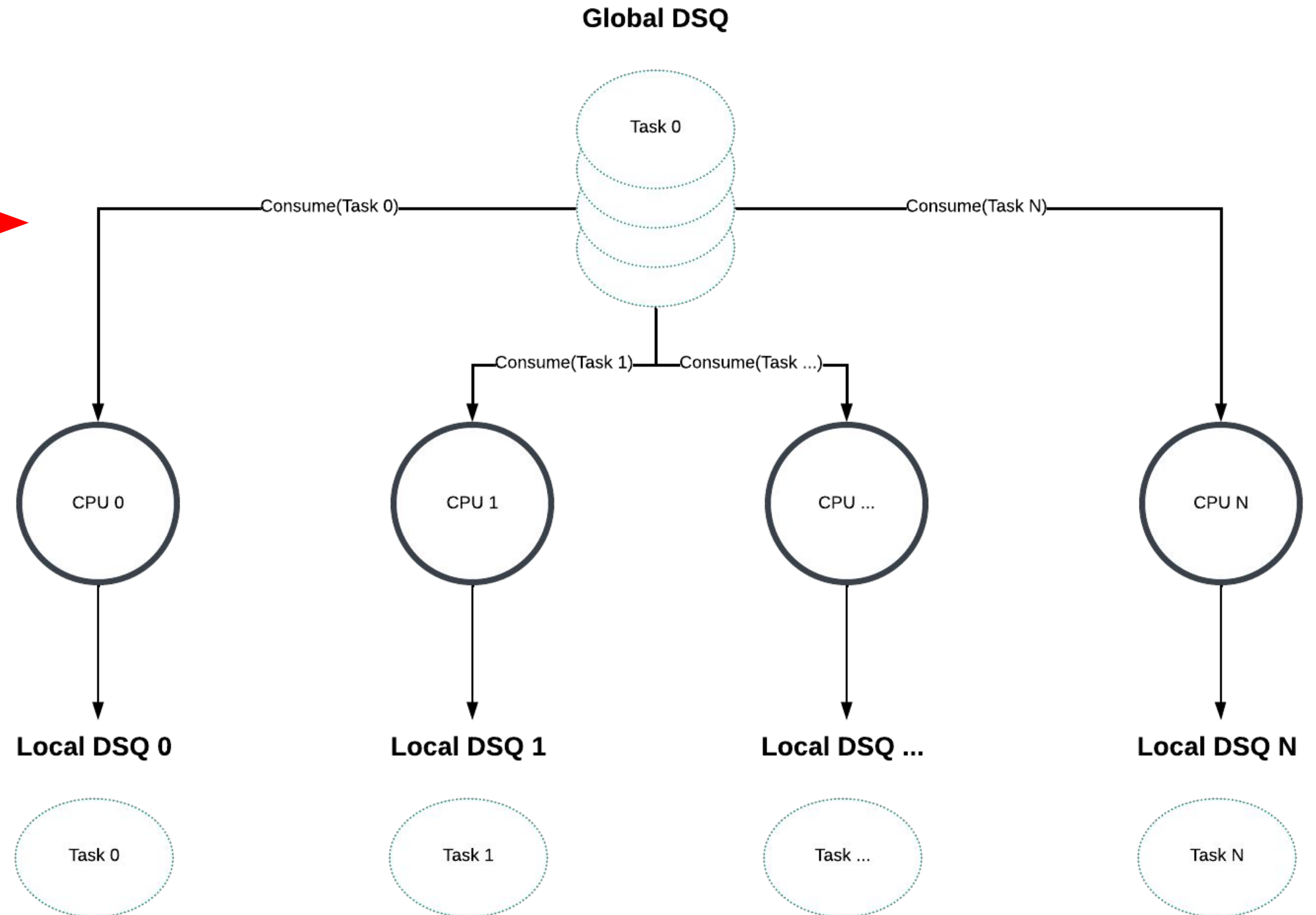
- And enqueue them on their local DSQ to be scheduled

Global DSQ

Task 0

Consume(Task 0)

Consume(Task N)

Consume(Task 1)

Consume(Task ...)

CPU 0

CPU 1

CPU ...

CPU N

Local DSQ 0

Local DSQ 1

Local DSQ ...

Local DSQ N

Task 0

Task 1

Task ...

Task N

Example 0: Global FIFO – consuming

```c
void BPF_STRUCT_OPS(mysched_dispatch, s32 cpu, struct task_struct *prev)
{
    scx_bpf_consume(/* queue id = */ 0);
}
```

Global FIFO works surprisingly well on single socket / CCX machines

- Work conserving
- Very, very simple
    - https://github.com/sched-ext/scx/blob/main/scheds/c/scx_simple.bpf.c
    - 155 lines of code

```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
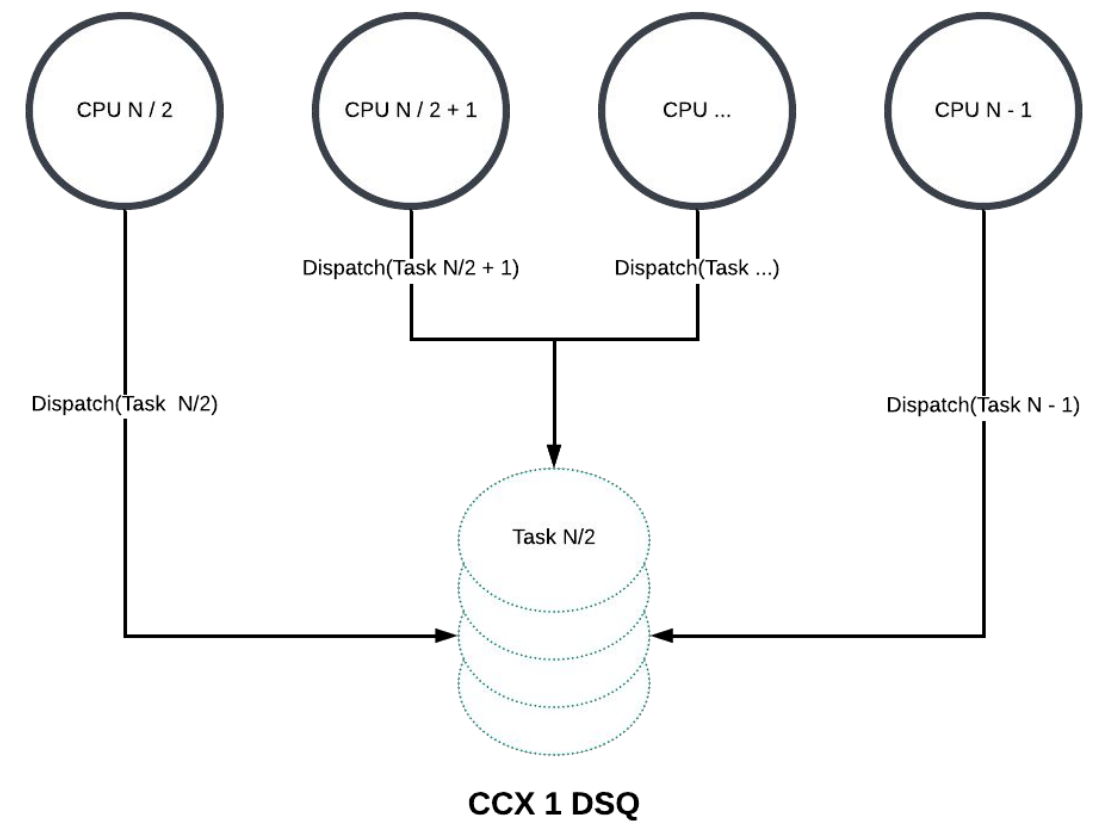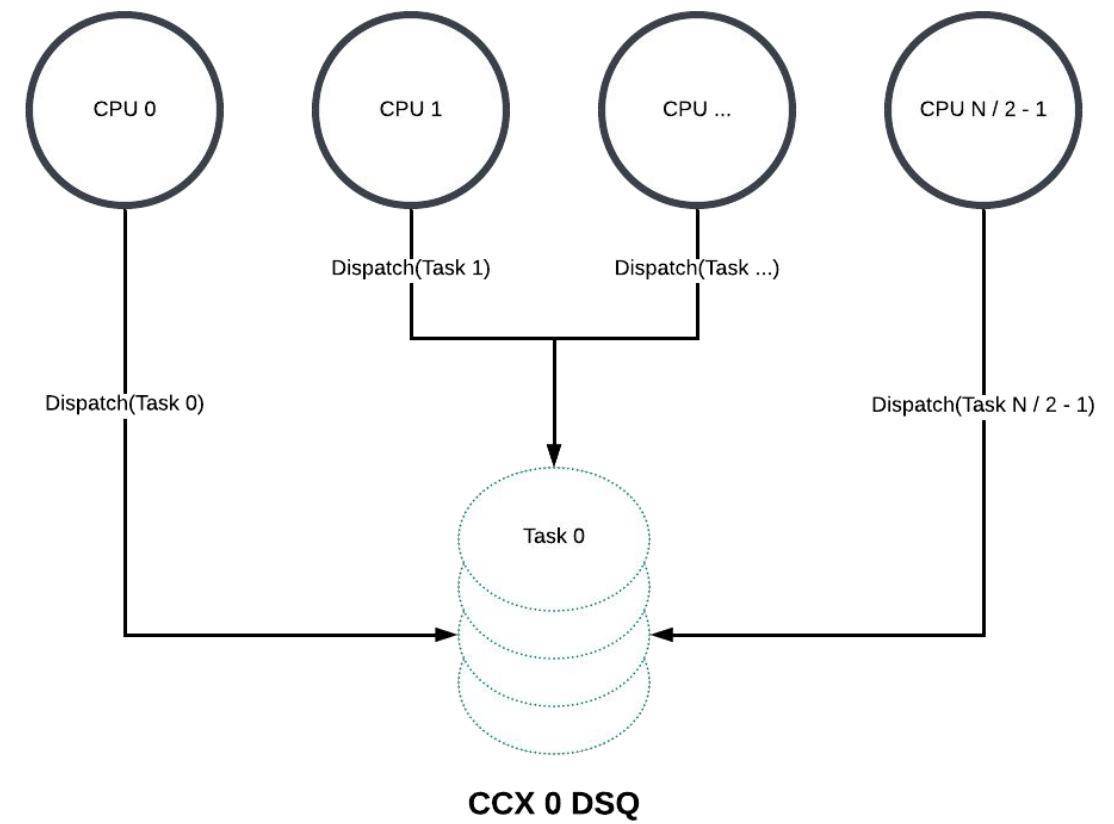
Example 1: Per-CCX FIFO – consuming

- 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.
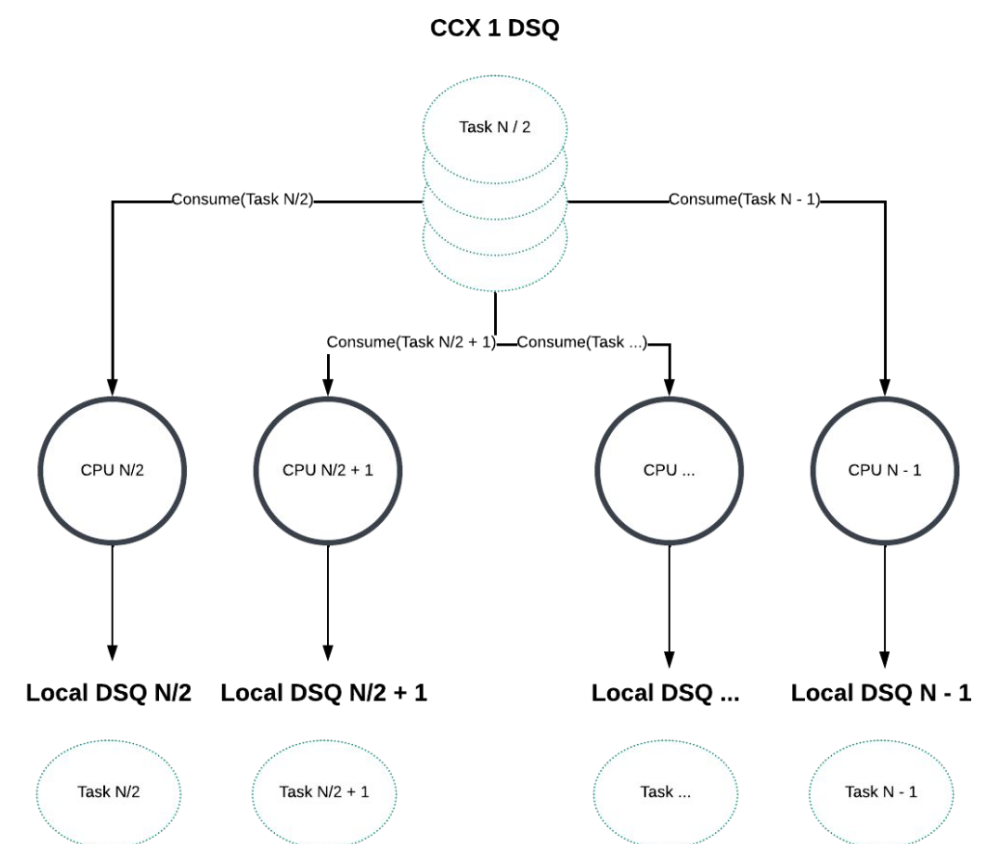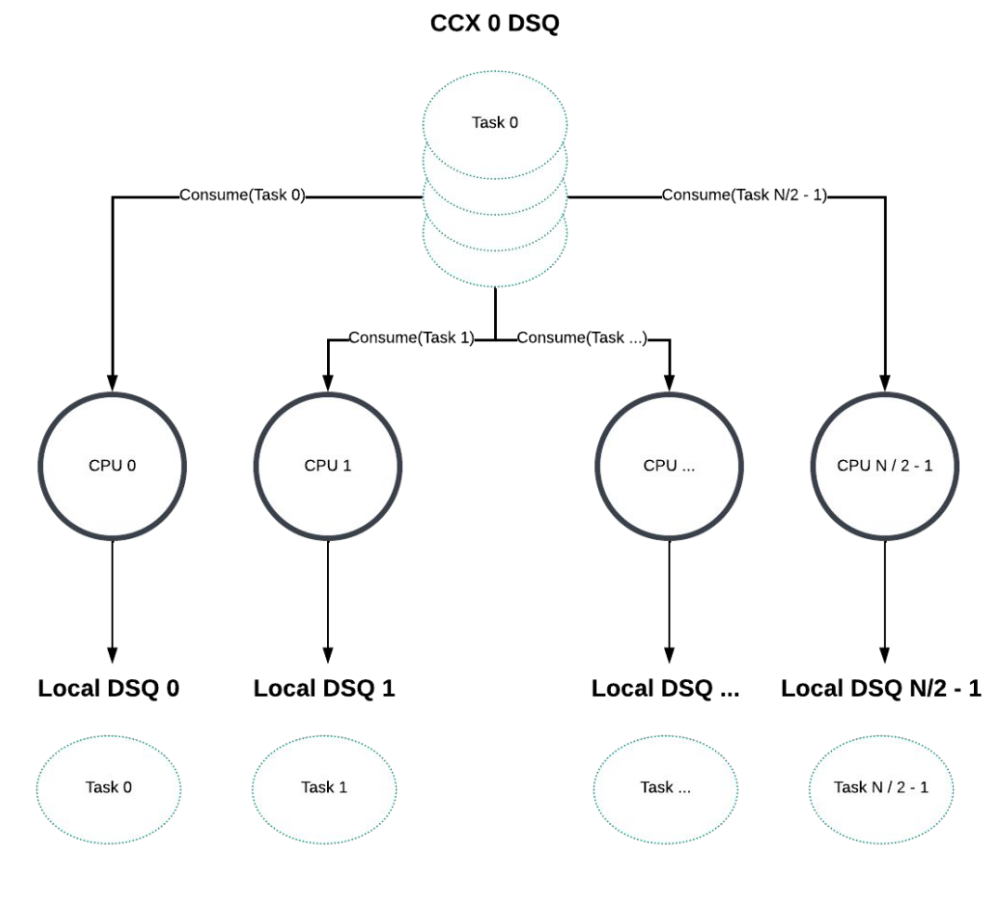
03 Example schedulers

Meta

Meta-developed Schedulers

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

Layered: https://github.com/sched-ext/scx/blob/main/scheds/rust/scx_layered/src/bpf/main.bpf.c

scx_rustland by Andrea Righi

https://github.com/sched-ext/scx/tree/main/scheds/rust/scx_rustland

Prioritizes interactive workloads over CPU-intensive workloads

Default Linux scheduler
~25-30 fps

0:09 / 0:52
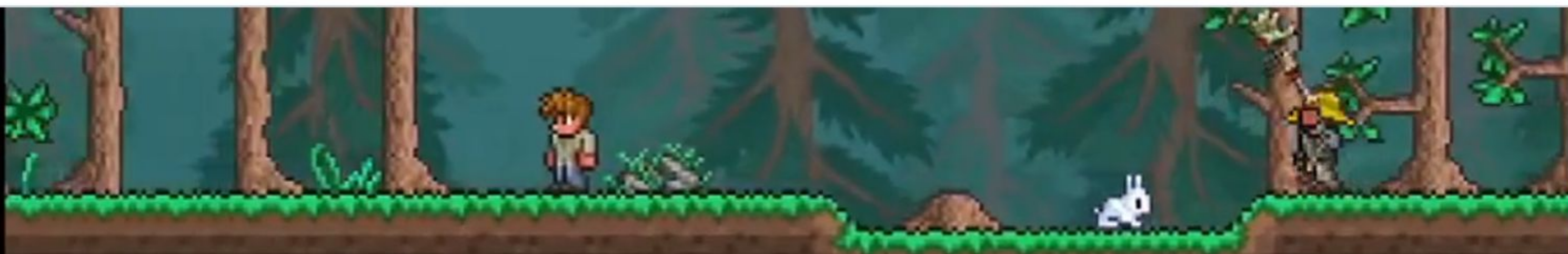
scx_rustland scheduler
~60 fps

0:33 / 0:52

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)

Top priority for sched_ext is upstreaming it

- Still iterating with members of the upstream community and incorporating feedback
    - We are committed to getting sched_ext upstreamed as long as it takes
- Latest v5 patch set (https://lore.kernel.org/all/20231111024835.2164816-1-tj@kernel.org/T/#u):
    - 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
  - Soft-affinity
- 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 (v5): https://lore.kernel.org/all/20231111024835.2164816-1-tj@kernel.org/T/#u
- Example schedulers: https://github.com/sched-ext/scx/tree/main
- 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

s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);

No

Migrate to selected CPU

Task becomes runnable

void (*enqueue)(struct task_struct *p, u64 enq_flags);

Task dispatched directly to CPU?

Yes — Task enqueued in the CPU's local DSQ

No — Task enqueued in the BPF scheduler

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

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