# Sched Ext

**The extensible sched_class**

David Vernet
Kernel engineer

Meta

# Agenda

01 What is sched_ext?

02 Common objections

03 Interesting Changes / Additions

# 01 What is sched_ext?

# sched_ext enables scheduling policies to be implemented in BPF programs

- New sched_class, at a lower priority than CFS
- Enables scheduling policies to be written in BPF programs
- No ABI stability restrictions – purely a kernel <-> kernel interface

# Why are we doing this?

- *Rapid experimentation*
  - 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 schedulers that are optimized for specific services*
  - Used in Meta production to **optimize main FB web service by 1.25 - 3% RPS, 3-6% P99 latency** compared to vanilla (but finely tuned) CFS, which results in O($millions) in capacity savings
    - Going to try and generalize, eventually add to CFS if anything is sufficiently applicable
- *Quick rollouts of new policies (e.g. for core sched)*
  - Temporary workaround for CPU bugs until full mitigation is rolled out
- *Moving policy decisions and complexity into user space*
  - Avoids workarounds like custom threading implementations and other flavors of kernel bypass

# 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

```
/* 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-v3/include/linux/sched/ext.h

```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",
};
```

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

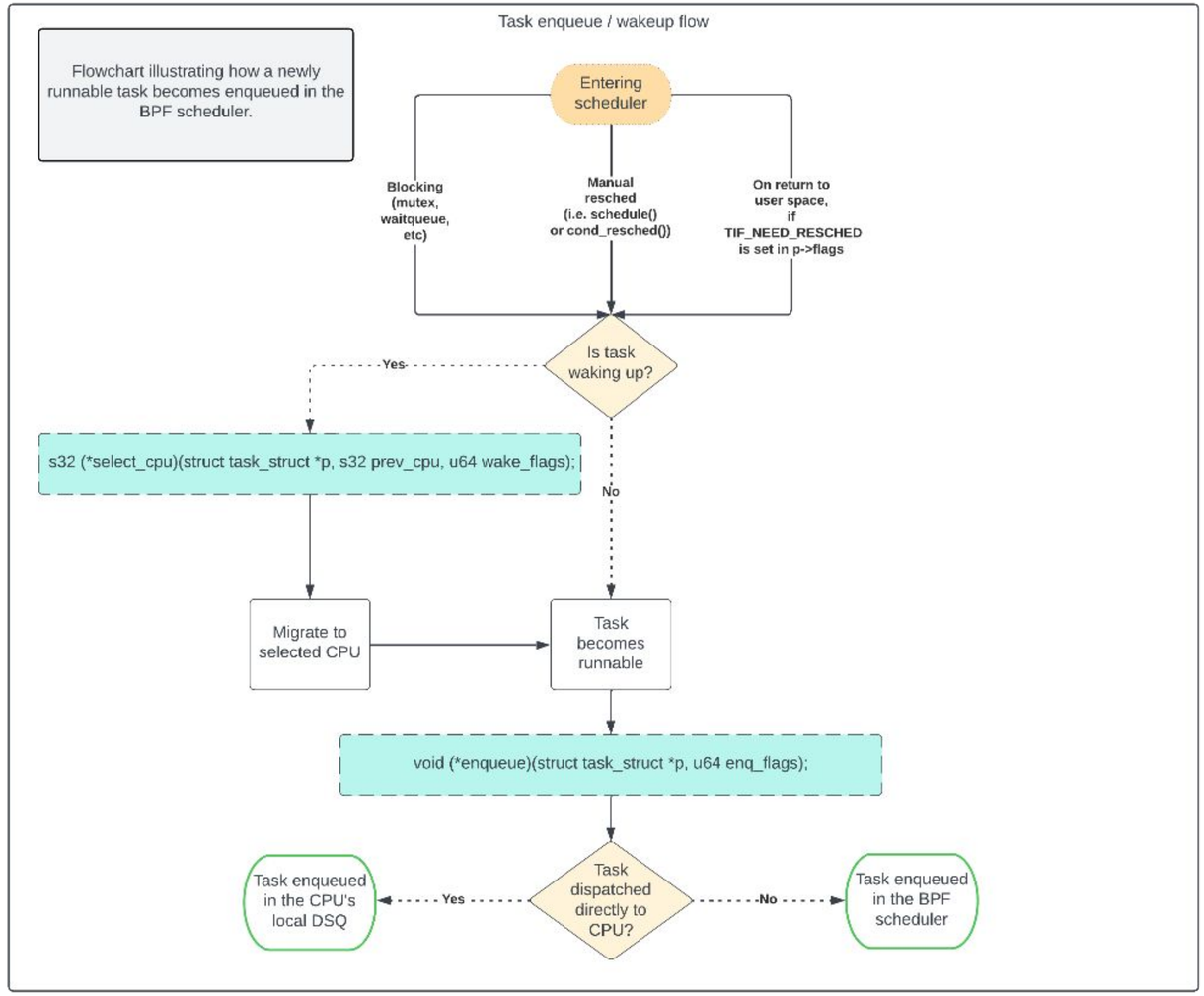
- Abstraction layer between BPF and main kernel for managing queues of tasks
- M:N relationship between DSQs and runqueues.
  - Gives schedulers flexibility
    - Per-domain (NUMA node, CCX, etc) DSQ?
    - Global DSQ?
    - Per-cgroup DSQ?
- The data structure / abstraction layer for managing tasks between main kernel <-> BPF scheduler (more on next slide). *Not* the only data structure for managing tasks in BPF
  - BPF lists
  - BPF rbtrees
  - BPF maps such as BPF_MAP_TYPE_HASHAMP, BPF_MAP_TYPE_QUEUE

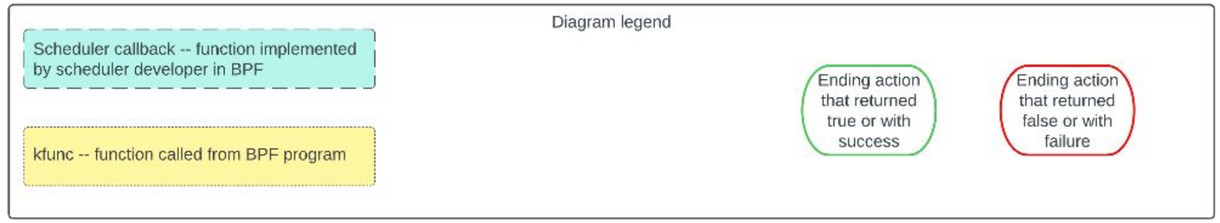
# DSQ operations: dispatch and consume

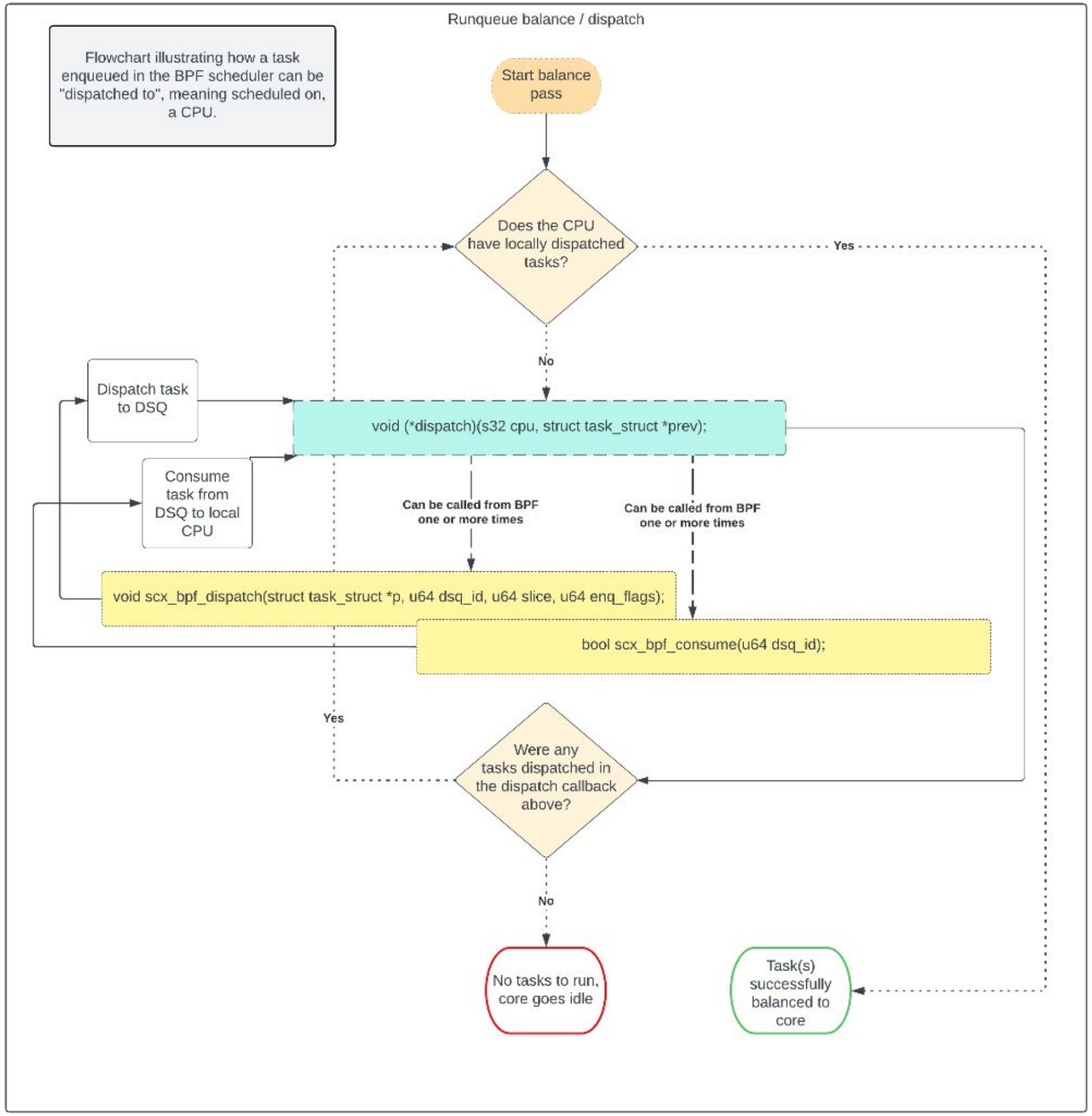
- DSQs provide two main operations (other than create / destroy):
- **Dispatch**: Placing a task into a DSQ
  - *Not* the same as picking the task to run on a CPU [0]
  - Can be done in one of two callbacks:
    - `ops.enqueue()`: Callback invoked when task is being enqueued in the BPF scheduler. If task is not dispatched to a DSQ, it is assumed to be enqueued in the BPF scheduler, and will be dispatched in the `ops.dispatch()` callback. Failing to run the task before some timeout threshold occurs will cause the scheduler to be unloaded, and revert the system back to CFS.
    - `ops.dispatch()`: Callback invoked when a core is going to go idle if a task is not found. Dispatching from `ops.dispatch()` would commonly happen if the task is enqueued in the BPF program directly (e.g. by being stored in a BPF rbtree graph-type data structure).
- **Consume**: “Consuming” a task from a DSQ to run on the calling CPU
  - More conceptually similar to pick_next_task()
  - Typically done in `ops.dispatch()` when a core is will go idle if no task is found

[0]: Tasks may be “directly dispatched” to the per-CPU local DSQ, skipping the dispatch() -> consume() flow

# 01 What is sched_ext?



# 02 Common objections

# There are a set of common objections I keep hearing

- Most people agree that the feature is useful. Objections are not about design or functionality
- I don’t agree with the rationale of many of these objections, but we need to address them in a consistent and explicit manner
- These are becoming especially pervasive as BPF transitions to “new” model described by Alexei

# “sched_ext will kill CFS contributions”

## Rationale

- Schedulers will stay out-of-tree
- No incentive for vendors, contributors, etc to contribute
- We’ll have ton of non-GPL schedulers being loaded

## How to address

- Clarify guarantees for upstreamed vs. not upstreamed schedulers
  - Guaranteed to prevent breakages from changing struct sched_ext_ops
  - Performance regressions?
- We need to be very explicit about expectations for **any** upstreamed progs, just like we are for kfuncs
  - This is not sched_ext-specific, just happens to be in the spotlight

# “sched_ext will force UAPI onto the scheduler”

## Rationale

- BPF connects user space to the scheduler
- Linus has said people haven’t *yet* complained about breakages, but they could
- BPF prog != module

## How to address

- Be very explicit about stability guarantees for all struct_ops BPF progs
  - Similar to kfuncs
- Need to develop a well-defined, well codified framework for upstreamed BPF progs

# “sched_ext will be support nightmare for distros”

## Rationale

- Distros will have to support whatever out-of-tree BPF schedulers that users / vendors implement
- Out-of-tree schedulers should taint the kernel because of how core the program is

## How to address

- Draw comparisons to other BPF features like XDP
- Write up a FAQ about this? Loop distro maintainers we have good relationships to help provide context?

# 03 Interesting Changes / Additions

# bpf_cpumask

- Goal: natural interaction w/ cpumask objects from BPF programs
  - Must be able to create, not just inspect or modify existing cpumasks

```
/* Definition and many more kfuncs in kernel/bpf/cpumask.c */
struct bpf_cpumask {
        cpumask_t cpumask;
        struct rcu_head rcu;
        refcount_t usage;
};

struct bpf_cpumask *bpf_cpumask_create(void) __ksym;
struct bpf_cpumask *bpf_cpumask_acquire(struct bpf_cpumask *cpumask) __ksym;
void bpf_cpumask_release(struct bpf_cpumask *cpumask) __ksym;
/* Can interact with plain cpumasks, not just bpf_cpumask */
bool bpf_cpumask_full(const struct cpumask *cpumask) __ksym;
u32 bpf_cpumask_any(const struct cpumask *cpumask) __ksym;
```

From kernel/bpf/cpumask.c

# Use RCU context instead of `kptr_get`

- RCU-safe types don't need explicit `kptr_get`, we're in RCU CS so can acquire and use
  - Applies to `task_struct`, `cgroup`, `cpumask`
  - Modulo sleepable progs, where explicit `bpf_rcu_read_{lock, unlock}` is needed

```
/* Before */

struct cgroup *ancestor;

kptr = bpf_cgroup_kptr_get(&v->cgrp);
if (!kptr) { /* snip */ }
ancestor = bpf_cgroup_ancestor(kptr, 1);
bpf_cgroup_release(kptr);
if (ancestor) {
    ...
}
```

```
/* After */

struct cgroup *cgrp, *ancestor;

bpf_rcu_read_lock();
cgrp = v->cgrp;
if (!cgrp) { /* snip */ }
ancestor = bpf_cgroup_ancestor(cgrp, 1);
bpf_rcu_read_unlock();
if (ancestor) {
    ...
}
```

# scx_example_flatcg: rbtree + local kptr stashing

- Implements hierarchical weight-based cgroup CPU control by flattening the cgroup hierarchy into a single layer

```
struct cgv_node {
    struct bpf_rb_node  rb_node;
    __u64               cvtime;
    __u64               cgid;
};

struct cgv_node_stash {
    struct cgv_node __kptr *node;
};

private(CGV_TREE) struct bpf_spin_lock cgv_tree_lock;
private(CGV_TREE) struct bpf_rb_root cgv_tree __contains(cgv_node, rb_node);

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 16384);
    __type(key, __u64);
    __type(value, struct cgv_node_stash);
} cgv_node_stash SEC(".maps");
```

From https://github.com/sched-ext/sched_ext/blob/sched_ext-v3/tools/sched_ext/scx_example_flatcg.bpf.c

# scx_example_flatcg: rbtree + local kptr stashing

- Stash in map on cgroup_init, unstash and add to rbtree on enqueue

```
/* fcg_cgroup_init */
struct cgv_node *cgv_node;
struct cgv_node_stash *stash;
stash = bpf_map_lookup_elem(&cgv_node_stash, &cgid);
if (!stash) { /* snip */ }

cgv_node = bpf_obj_new(struct cgv_node);
if (!cgv_node) { /* snip */ }

cgv_node->cgid = cgid;
cgv_node->cvtime = cvtime_now;

cgv_node = bpf_kptr_xchg(&stash->node, cgv_node);
```

From https://github.com/sched-ext/sched_ext/blob/sched_ext-v3/tools/sched_ext/scx_example_flatcg.bpf.c

# `scx_example_flatcg`: rbtree + local kptr stashing

- Stash in map on `cgroup_init`, unstash and add to rbtree on enqueue

```
/* cgrp_enqueued */
cgv_node = bpf_kptr_xchg(&stash->node, NULL);
if (!cgv_node) { /* snip */ }

bpf_spin_lock(&cgv_tree_lock);
cgrp_cap_budget(cgv_node, cgc);
bpf_rbtree_add(&cgv_tree, &cgv_node->rb_node, cgv_node_less);
bpf_spin_unlock(&cgv_tree_lock);
```

From https://github.com/sched-ext/sched_ext/blob/sched_ext-v3/tools/sched_ext/scx_example_flatcg.bpf.c


Meta