CS 5264/4224; ECE 5414/4414 (Advanced) Linux Kernel Programming Lecture 9

Process Scheduling III

February 25, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

EEVDF: Earliest Eligible Virtual Deadline First

- **Lag:** difference between the ideal runtime and the actual runtime of a task
- **Eligibility:** a task is eligible to run if its lag >= 0
- **Virtual deadline:** vruntime + requested vruntime

$$lag_T(t_1) = V_{avg}(t_1) - V_T(t_1) \geq 0$$

$$D_T(t_1) = V_T(t_1) + \Delta t_T \cdot \frac{w_{base}}{w_i}$$

EEVDF paper (1995): here

Scheduling related system calls

- `sched_getscheduler`, `sched_setscheduler`
- `nice`
- `sched_getparam`, `sched_setparam`
- `sched_get_priority_max`, `sched_get_priority_min`
- `sched_getaffinity`, `sched_setaffinity`
- `sched yield`

Linux Scheduler: Not One Size Fits All ...

- 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
  - For endusers: gaming, for hyperscalers, run a few internal workloads
- 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

The Need for More Scheduling Policies

- In-kernel scheduler design targets generality
  - Balance specific performance requirements of many applications
- Tailoring scheduling policies can substantially improve performance for specific workloads
  - (tail) latency, throughput, energy efficiency, security, etc.
  - e.g., workloads with a mix of short and long requests
  - multi-tenant setups
  - resource interferences (low-latency apps + background best-effort apps)
  - cache side channel attacks → mitigation: core isolation policies

Developing New Scheduling Policies in Linux Kernel is Hard

- Need to deal with ever-changing hardware landscape
  - increasing core counts
  - multi-core, NUMA
  - heterogeneous systems: big.LITTLE ARM, Intel SRF performance/efficiency cores
  - support for emerging compute devices: DPUs, domain-specific accelerators (GPUs, TPUs)
- Hard to develop, maintain, deploy, and test new implementations
  - Focusing on large-scale deployment, e.g., datacenter scale (millions of servers in the fleet)
  - Comply with complex kernel architecture
  - Requires extensive testing to avoid crashing the entire system
  - Disruptive upgrade require reboots, leading to downtime
  - C and assembly, hard to debug, complex synchronization, preemption, interrupts, etc.
  - Linux rarely adopts new scheduling policies → O(months) requirements from IT companies

Scheduler Design, Implementation, and Deployment

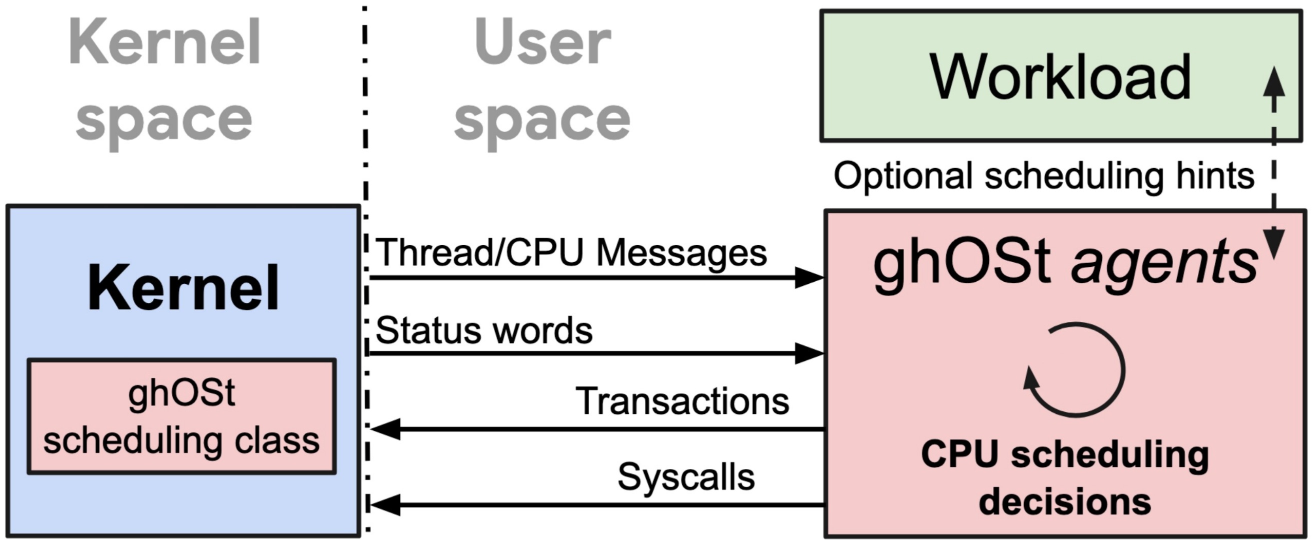
- agile userspace development
- ease of deployment
- while still enabling fast scheduling for performance
- flexibility: per-CPU or centralized scheduling model
- Principles
  - scheduling mechanism remains in the kernel
  - while policy resides in userspace
  - *Q: what abstractions and interfaces should we use?*
    - *Ideally, no applications code changes …*
    - *Compatible API/ABI*
    - *Can co-exist with existing in-kernel schedulers, e.g., CFS / EEVDF*
    - *Fast event communication between user/kernel space*
    - *…*

ghOSt: Fast & Flexbile User-Space Delegation of Linux Scheduling

- SOSP 2021 research paper from Google, here
- Design goals
  - Policies should be easy to implement and test
  - scheduling expressiveness and efficiency
  - enabling scheduling decisions beyond the per-CPU model
  - supporting multiple concurrent policies
  - non-disruptive updates and fault isolation

ghOSt Design

- Kernel side is implemented as a scheduling class (e.g., similar to SCHED_NORM)
- The scheduling class provide a rich API to define arbitrary scheduling policies
- The agents are the in-userspace scheduling policies
- The kernel shares thread status information via messages and status words
- The agents informs the kernel to make scheduling decisions via transactions/syscalls



- Agents can be implemented in any programming languages, debugging via standard tools
- For fault tolerance and isolation, if agents crash, the systems will fall back to the default scheduler, e.g., EEVDF
- No reboots
- Flexible scheduling policy model choices, per-CPU or global

- Kernel-to-agent communication: Messages
  - Why not mmap task_struct to userspace?
  - Why not expose thread state va sysfs/proc files, e.g., /proc/pid/...?
- Agent-to-kernel communication:
  - Agents send scheduling decisions to the kernel by committing transactions
  - syscall-based scheduling decision overhead: a few microseconds

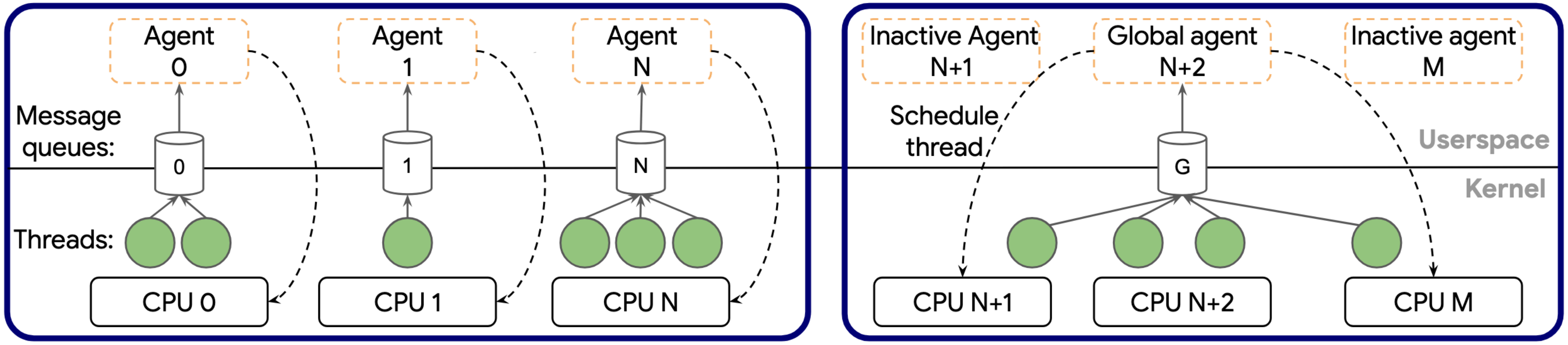
Messages
THREAD_CREATED
THREAD_BLOCKED
THREAD_PREEMPTED
THREAD_YIELD
THREAD_DEAD
THREAD_WAKEUP
THREAD_AFFINITY
TIMER_TICK

Syscalls
AGENT_INIT()
START_GHOST()
TXN_CREATE()
TXNS_COMMIT()
TXNS_RECALL()
CREATE_QUEUE()
DESTROY_QUEUE()
ASSOCIATE_QUEUE()
CONFIG_QUEUE_WAKEUP()

```
void Agent::PerCpuSchedule() {
    DrainMessageQueue(); // Read messages from queue
    Thread *next = runqueue_.Dequeue();
    if (next == nullptr) return; // Runqueue empty.
    // Schedule thread:
    Transaction *txn = TXN_CREATE(next->tid, my_cpu);
    TXNS_COMMIT({txn});
    if (txn->status != TXN_COMMITTED) {
        // Txn failed. Move thread to end of runqueue.
        runqueue_.Enqueue(next);
        return;
    }
    // The schedule has succeeded for `next`.
}
```

Per-CPU and Centralized Scheduler

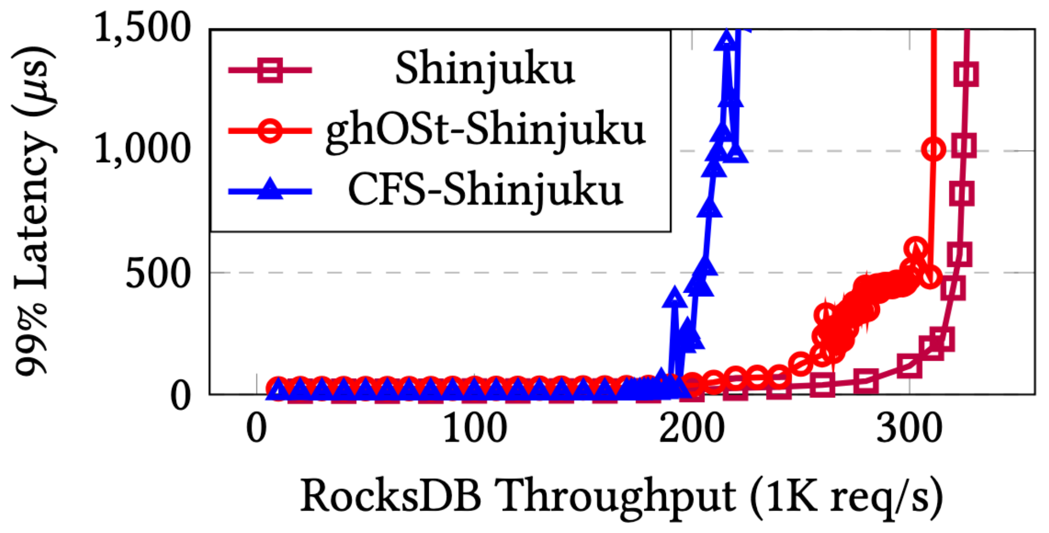
- Fine-grained policy management: per-cpu or centralized
- Centralized scheduler
  – one global agent with a single queue



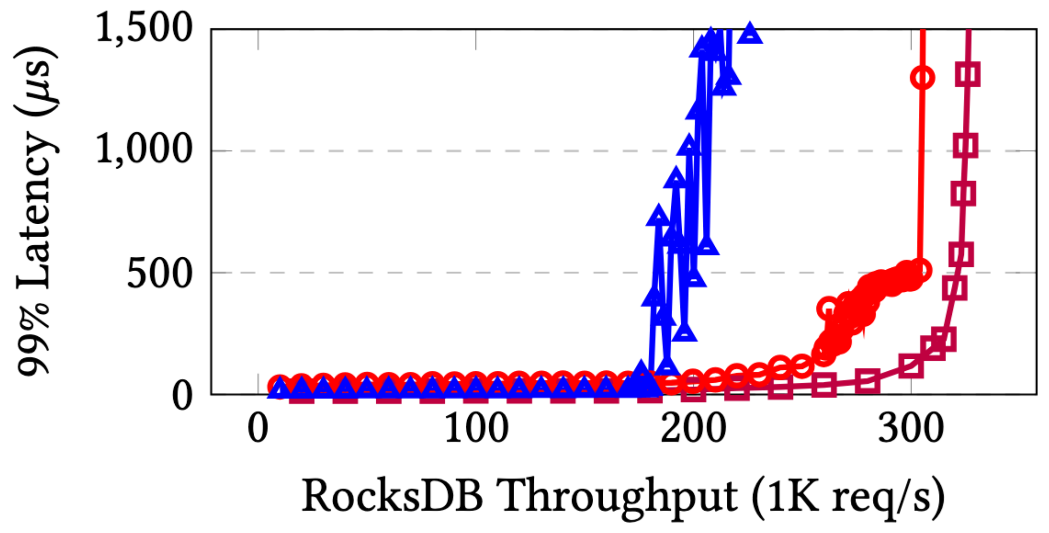
Enclave 0 running per-CPU scheduling on CPUs 0 ⟶ N

Enclave 1 running centralized scheduling on CPUs N+1 ⟶ M

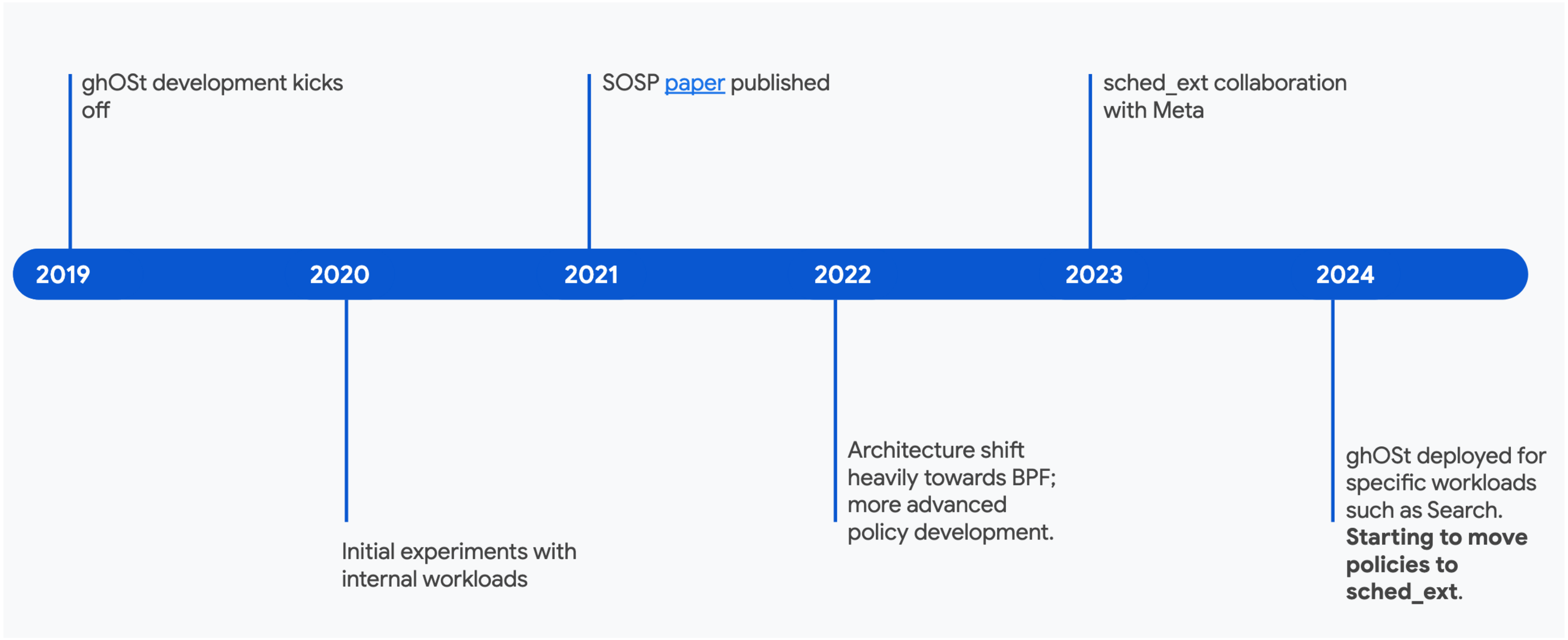
ghOSt Evaluation



(a) Tail latency for dispersive loads.



(b) RocksDB co-located with a batch app.


ghOSt development kicks off
SOSP paper published
sched_ext collaboration with Meta
2019
2020
2021
2022
2023
2024
Initial experiments with internal workloads
Architecture shift heavily towards BPF; more advanced policy development.
ghOSt deployed for specific workloads such as Search. **Starting to move policies to sched_ext.**

Source: here

Linux sched_ext: BPF-extensible Scheduler Class

- Berkeley Packet Filter (BPF)
  - A recently revived techniques that has attracted a lot of attention
  - Extensively used for system observability by offloading userspace code to run safely in kernel space
  - eBPF , the BPF verifier ensures that your custom scheduler has neither a memory bug nor an infinite loop
  - Safe fall back to default CFS/EEVDF schedulers
- Using BPF for pluggable scheduling
  - A new extensible scheduling class, SCHED_EXT (>SCHED_IDLE, <SCHED_NORMAL)
  - Allows you to write and run customized schedulers optimized for target workloads

[1] Extensible Scheduler Class, https://docs.kernel.org/scheduler/sched-ext.html
[2] The extensible scheduler class, https://lwn.net/Articles/922405/
[3] sched_ext schedulers and tools, https://github.com/sched-ext/scx

BPF: A Safe Way to Run Code in Kernel

- Kernel feature that allows custom code to run safely in the kernel
- Started in the early days for custom packet filtering
- Now much much larger and richer ecosystems
- Write C code, compile it to BPF bytecode, userspace can load it into the kernel

https://en.wikipedia.org/wiki/EBPF

sched_ext

- Write schedulers in BPF
  - implement a set of callbacks for handling: task wakeup, enqueue/dequeue, state change, load balancing, cgroup integrations, etc.
- Compile it
- Load it onto the system, letting BPF and core sched_ext infrastructure do all of the heavy lifting to enable it

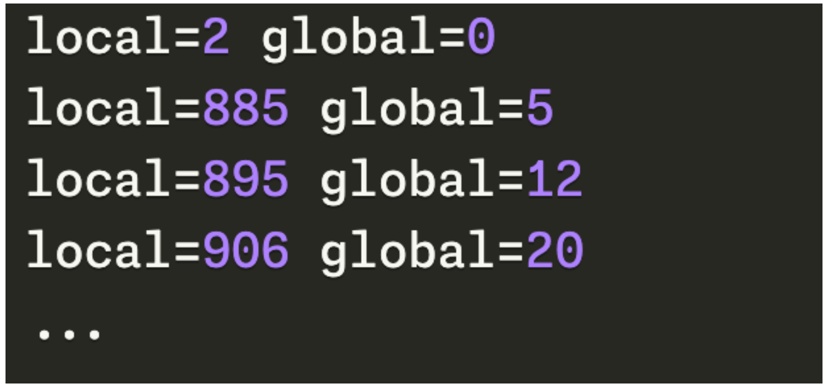
- Offload complicated logic to user space.
- Use of floating points
- Use standard debugging tools
- BPF makes it easy to share data between the kernel and user space

How to Use sched_ext

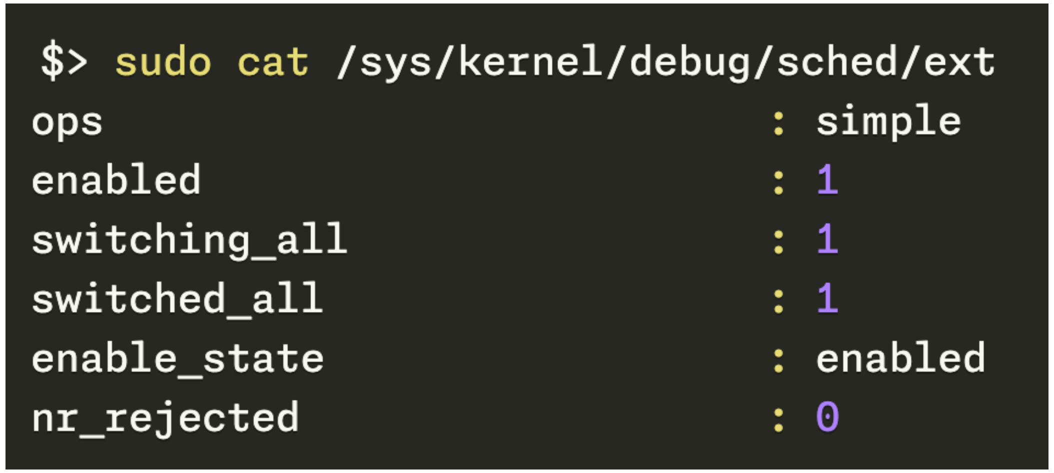
- Kernel needs to be compiled to support sched_exit
  - Enable the following configuration options in .config
  - Disable CONFIG_DEBUG_INFO_REDUCED and CONFIG_DEBUG_INFO_SPLIT first
  - Compile and boot the sched_ext enabled kernel
  - sched_ext is used only when the BPF scheduler is loaded and running.
    - » If a task explicitly sets its scheduling policy to SCHED_EXT, it will be treated as SCHED_NORMAL and scheduled by CFS until the BPF scheduler is loaded.

```
CONFIG_BPF=y
CONFIG_SCHED_CLASS_EXT=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_DEBUG_INFO_BTF=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BPF_JIT_DEFAULT_ON=y
CONFIG_PAHOLE_HAS_SPLIT_BTF=y
CONFIG_PAHOLE_HAS_BTF_TAG=y
```

- Tools under tools/sched_ext in Linux kernel source
  - Where the example userspace schedulers reside
  - "make CC=clang LLVM=1 –j"
  - Run the scheduler
    - » cd tools/sched_ext/build/bin
    - » sudo ./scx_simple
  - Despite being simple, scx_simple can even outperform CFS
  - sysfs interface for sched_ext status checking
    - » sudo cat /sys/kernel/debug/sched/ext

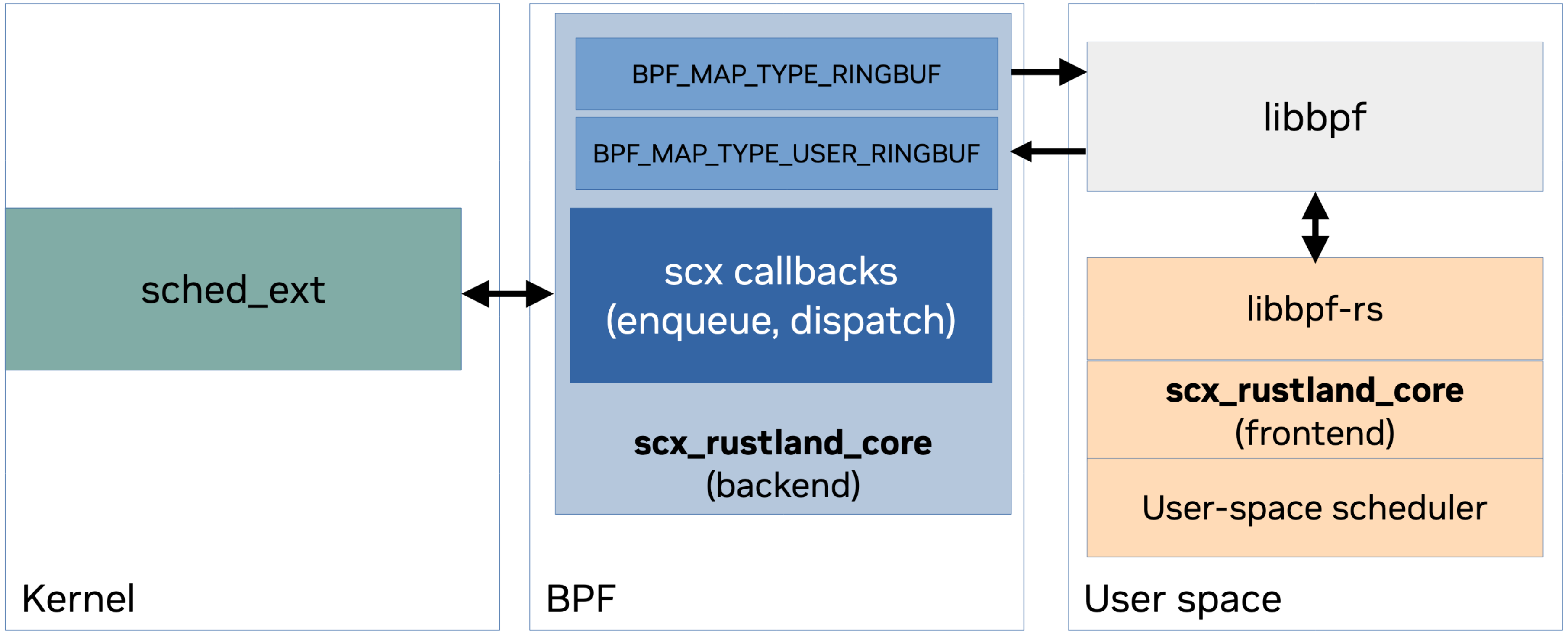


```
local=2 global=0
local=885 global=5
local=895 global=12
local=906 global=20
...
```



```
$> sudo cat /sys/kernel/debug/sched/ext
ops                             : simple
enabled                         : 1
switching_all                   : 1
switched_all                    : 1
enable_state                    : enabled
nr_rejected                     : 0
```

sched-ext Architecture and Workflow


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

1. sched_ext callback intercepts tasks that want to run
2. Tasks are added to a BPF_MAP_TYPE_RINGBUF
3. BPF component schedules a user-space task (scheduler)
4. User-space scheduler consumes tasks from the ringbuf and assigns a CPU and time slice to each one of them
5. Tasks are added to a BPF_MAP_TYPE_USER_RINGBUF
6. BPF component consumes tasks from the user ringbuf and dispatches

sched_ext Architecture and Interface

```
+========================+
|| User-space part of   ||
|| your scheduler       ||
|| (e.g., main.rs)      ||
+========================+
   \\//   ^^^^
   \\//   ^^^^ <== Interface 4: BPF scheduler <=> user-space counter part
   \\//   ^^^^
+========================+
|| Your BPF scheduler   ||
|| (e.g., main.bpf.c)   ||
+========================+
   ^^^^   \\// <== Interface 3: BPF scheduler => sched_ext framework
   ^^^^   \\//
   ^^^^  <======== Interface 2: sched_ext framework => BPF scheduler
+========================+
|| Sched_ext framework  ||
||                      ||
|| (kernel/sched/ext.c) ||
+========================+
          ^^^^
          ^^^^ <== Interface 1: core kernel scheduler => scheduler class
          ^^^^

+------------------------+
| Core kernel scheduler  |
|  (kernel/sched/core.c) |
+------------------------+
```

Source: https://blogs.igalia.com/changwoo/sched-ext-scheduler-architecture-and-interfaces-part-2/

- Interface 1: core kernel scheduler ➔ scheduler class (struct sched_class)
  - The sched_ext framework provides the common implementation for BPF schedulers.
- Interface 2: sched_ext framework ➔ BPF scheduler
  - sched_ext_ops.init(), .exit()
  - .init_task(), .exit_task()
  - .runnable() , .running(), .stopping()
  - .select_cpu(), .enqueue()
  - .dispatch()
  - .tick()

```
+========================+
|| User-space part of   ||
|| your scheduler       ||
|| (e.g., main.rs)      ||
+========================+
   \\//   ^^^^
   \\//   ^^^^ <== Interface 4: BPF scheduler <=> user-space counter part
   \\//   ^^^^
+========================+
|| Your BPF scheduler   ||
|| (e.g., main.bpf.c)   ||
+========================+
   ^^^^   \\// <== Interface 3: BPF scheduler => sched_ext framework
   ^^^^   \\//
   ^^^^  <======== Interface 2: sched_ext framework => BPF scheduler
+========================+
|| Sched_ext framework  ||
||                      ||
|| (kernel/sched/ext.c) ||
+========================+
          ^^^^
          ^^^^ <== Interface 1: core kernel scheduler => scheduler class
          ^^^^

+------------------------+
| Core kernel scheduler  |
|  (kernel/sched/core.c) |
+------------------------+
```

- Interface 3: BPF scheduler → sched_ext framework
  - – BPF scheduler need to talk to sched_ext to take a certain action
  - – via BPF helper function or DSQ (dispatch queue)
  - – DSQ: core consturct between BPF scheduler and sched_ext
    - » a queue hosting runnable tasks (ordered in FIFO or virtual time, vtime)
    - » sched_ext also maintains internal DSQs: global DSQ, and per-CPU DSQ (both are FIFO)
    - » BPF scheduler can create DSQs (FIFO or vtime) to manage tasks by itself
      - scx_bpf_create_dsq(), and initialized during sched_ext_ops.init()
    - » A task can be enqueued in FIFO order (scx_bpf_dispatch()) or vtime order (scx_bpf_dispatch_vtime()), during sched_ext_ops.enqueue()
    - » Consuming tasks by moving task from a custom DSQ and move it to internal DSQ: scx_bpf_consume() or scx_bpf_consume_task() as part of sched_ext_ops.dispatch()
    - » helper utilities
      - scx_bpf_dsq_nr_queued()
      - scx_bpf_destroy_dsq()
      - scx_bpf_select_cpu(), scx_bpf_kick_cpu(): select CPU, wakeup CPU

- Interface 4: BPF scheduler and user-space counterpart
  – Any user-space program (C, Rust) + libbpf API

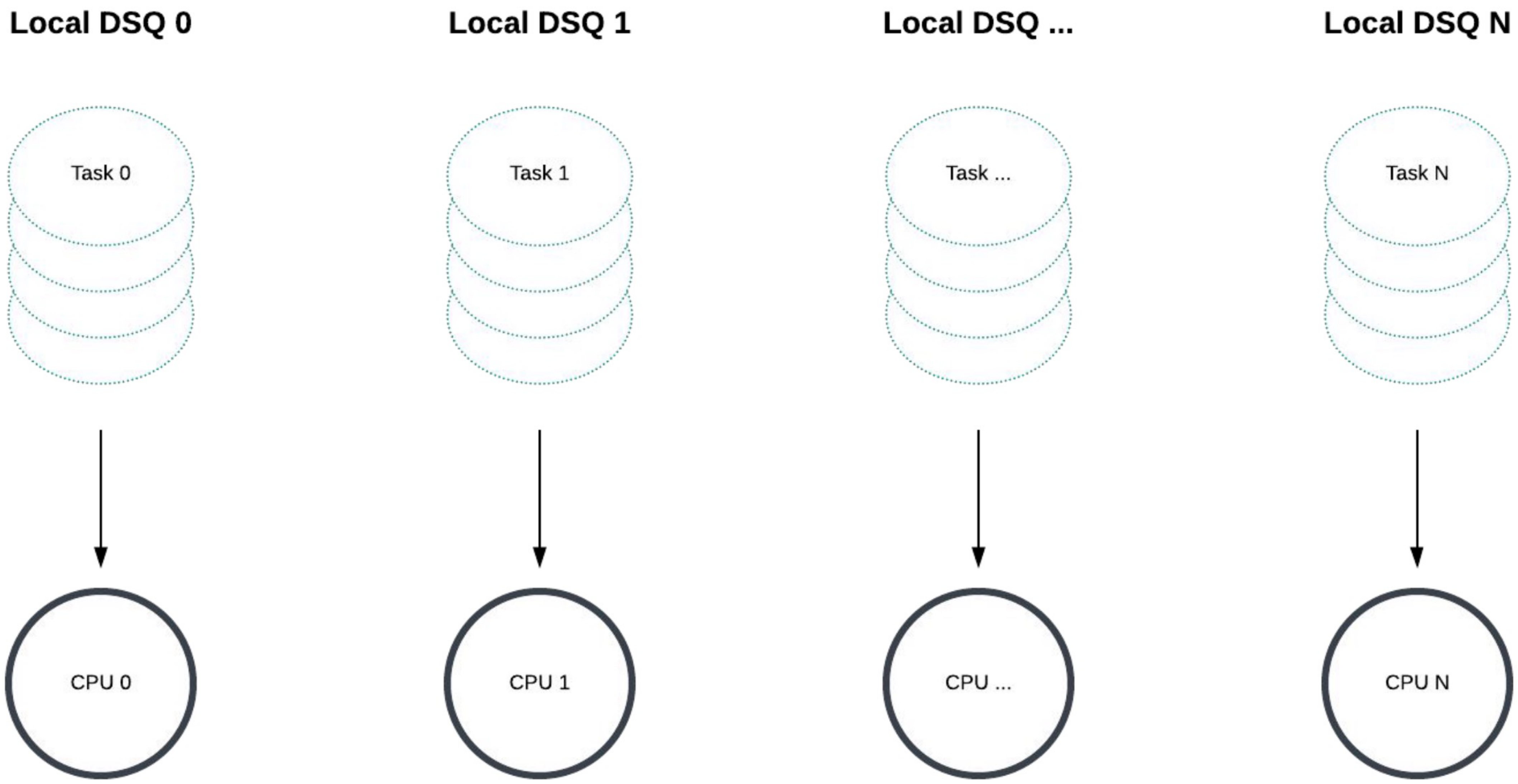
sched_ext

- sched_ext defines *struct sched_ext_ops* in kernel/sched/ext.c which specifies a list of hook functions callbacks that need to be realized by the specific scheduler instance
  - sched_ext_ops.init()
  - sched_ext_ops.exit()
  - scx_bpf_switch_all()
  - scx_bpf_create_dsq(SHARED_DSQ, -1)
  - enable()
  - task_struct->scx { .slice, .dsq_vtime }
  - enqueue()
    - » local queue (SCX_DSQ_LOCAL), staging area waiting for exec, per-CPU local DSQ
    - » scx_bpf_dispatch() // take time slice as input (e.g., 20ms)
    - » scx_bpf_dispatch_vtime()
  - stopping()

sched_ext Scheduling Policy

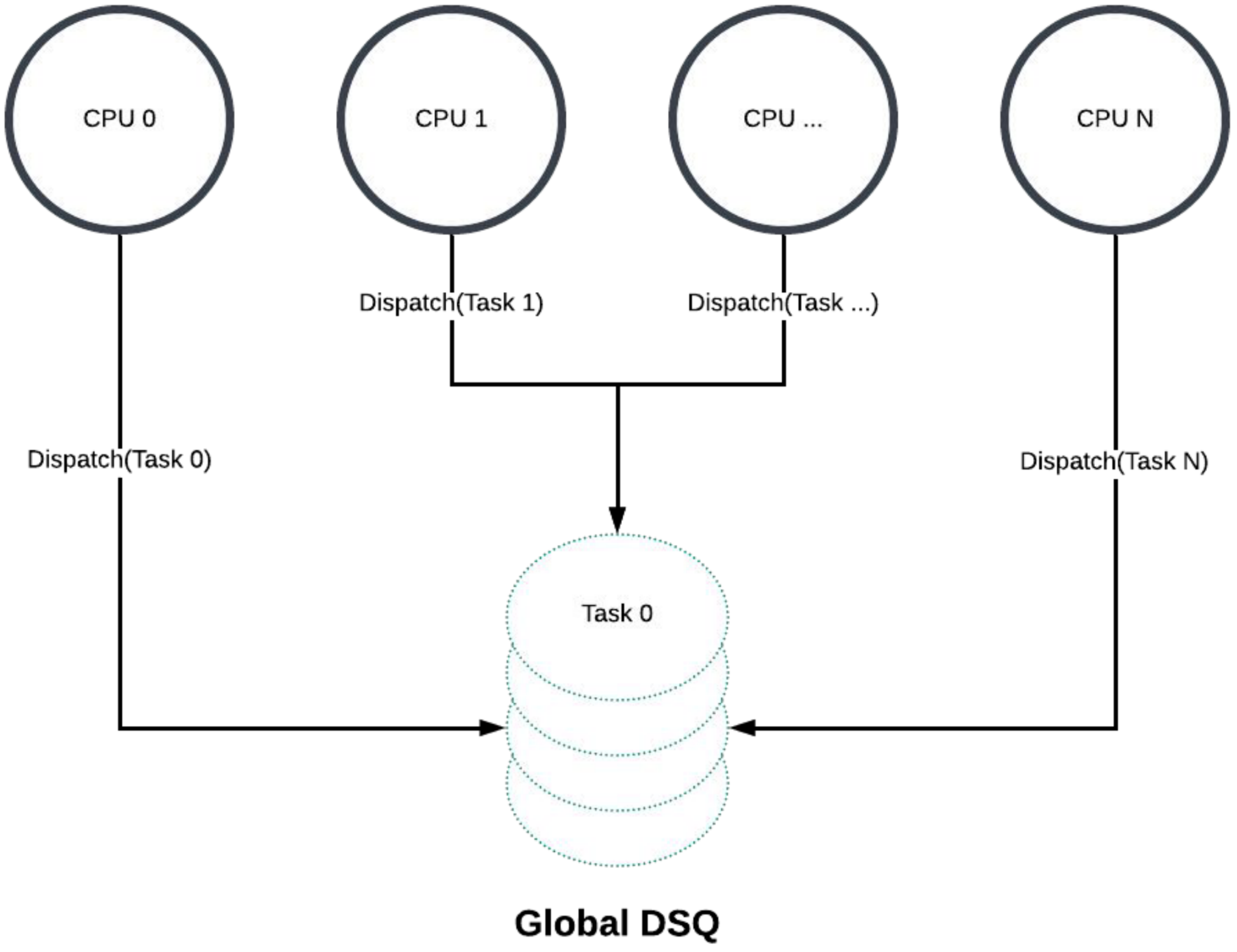
- Dispatch queue: basic building block of scheduling policies
  - A CPU always executes a task from its local DSQ
  - A task can be moved from non-local DSQ to the target CPU's local DSQ
  - When looking for next task to run, if local DSQ is not empty, pick first task there
  - Otherwise, the CPU tries to move a task from the global DSQ
  - If that doesn't yield a runnable task either, ext_ops.dispatch() is invoked() to wait for population of the local DSQ
- Scheduling cycle
  - If a task is waking up, ext_ops.select_cpu() to wake up the selected CPU
  - scx_bpf.dsq_insert()
    - » ext_ops.enqueue(): immediately insert the task into either the global or local DSQ
  - When a CPU is ready to schedule, follow the above scheduling policy

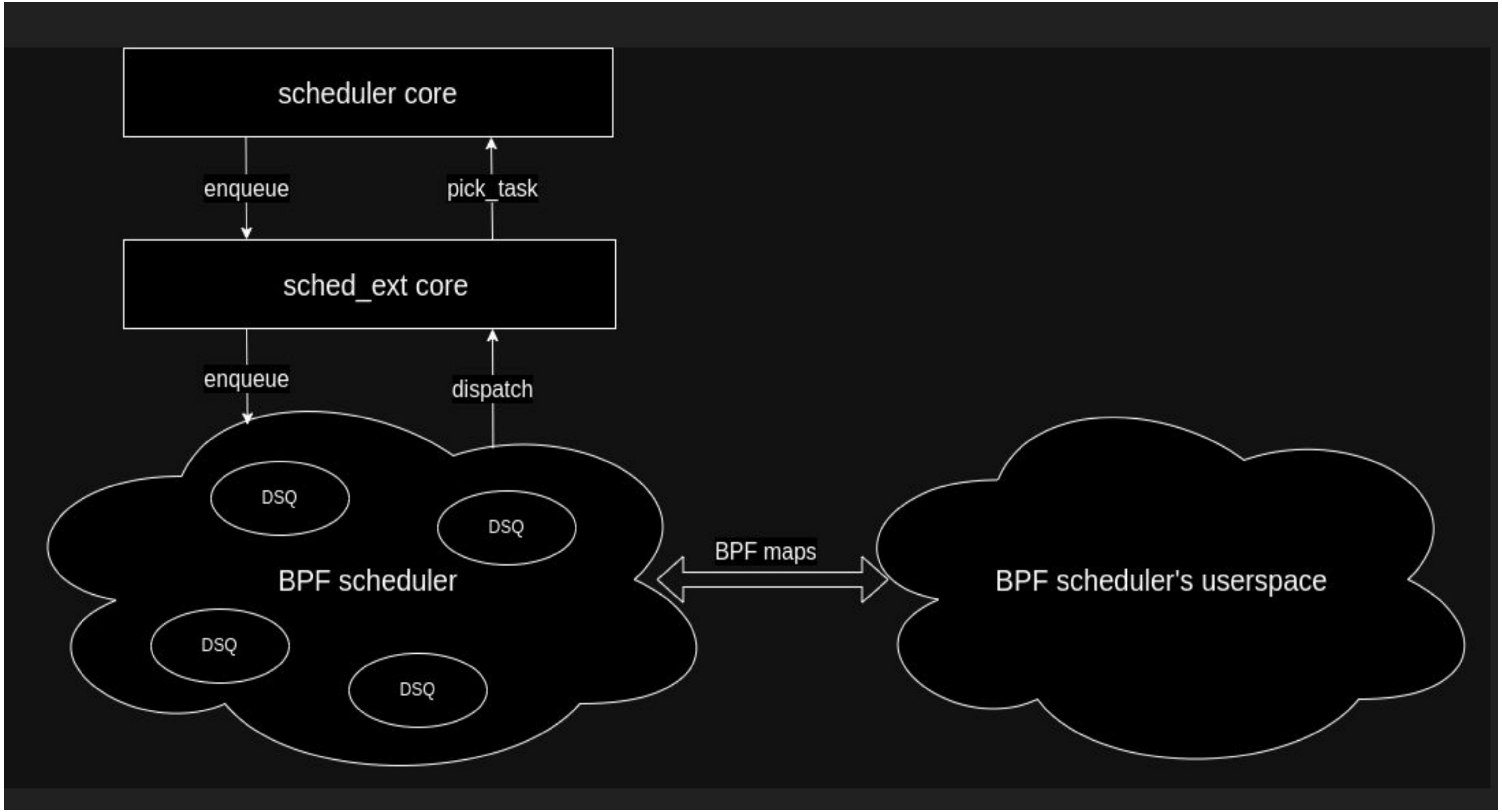
Local DSQs: per-CPU runqueue



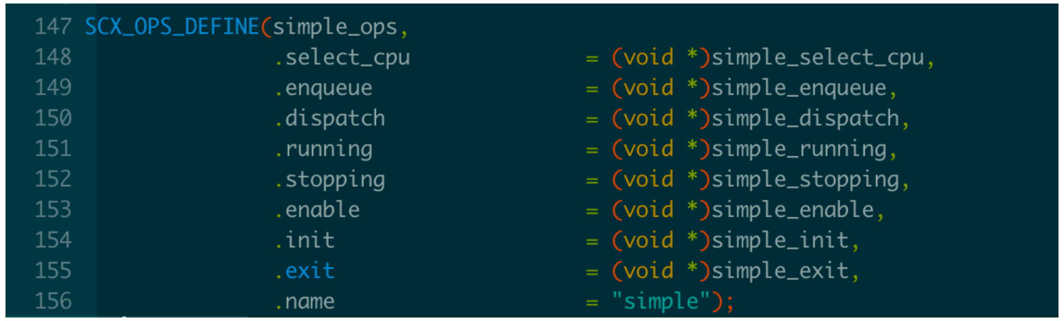
Source: https://www.socallinuxexpo.org/sites/default/files/presentations/Sched%20Ext%20-%20SCaLE%2021x.pdf

Global Queue



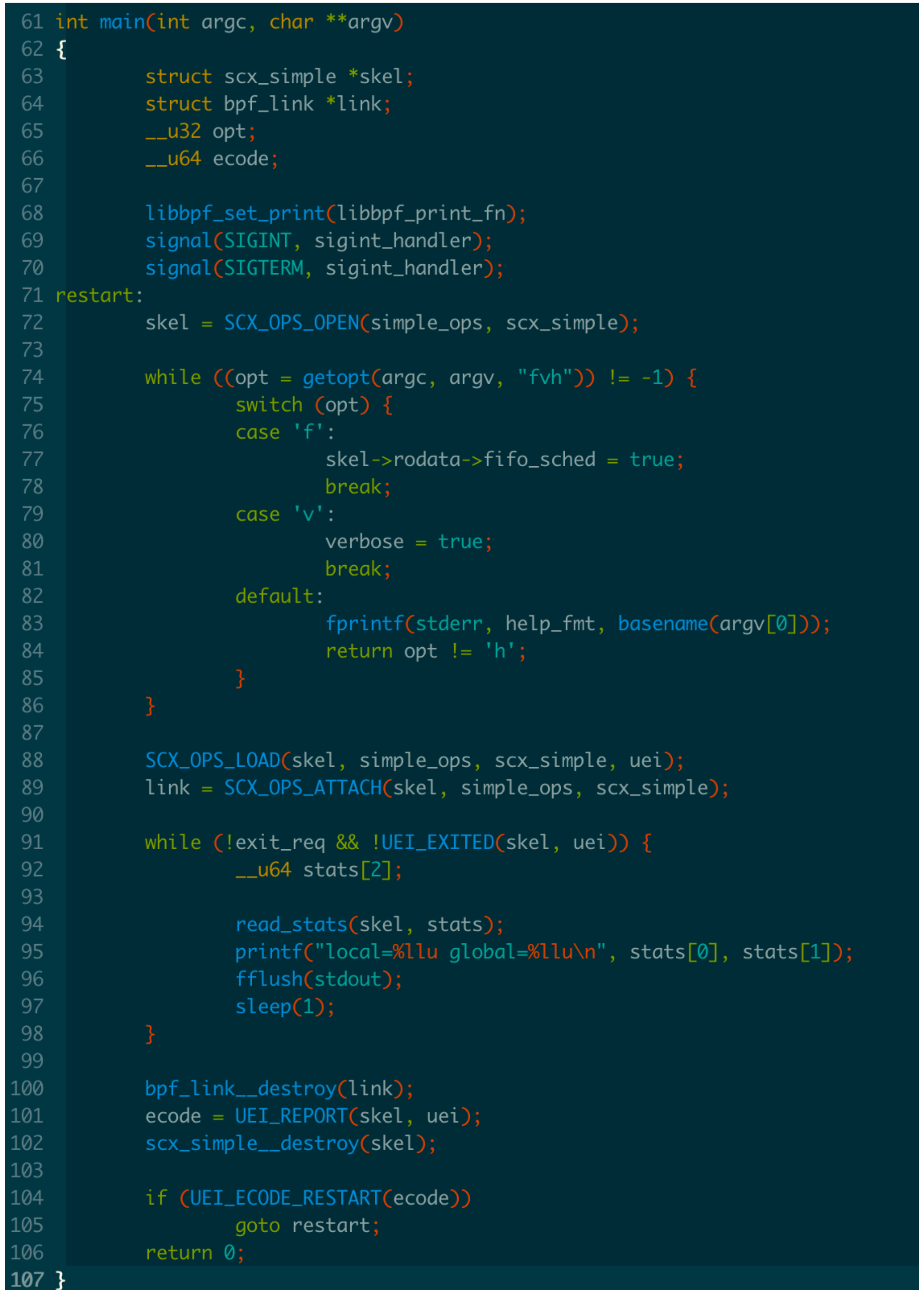

scheduler core
enqueue
pick_task
sched_ext core
enqueue
dispatch
DSQ
DSQ
BPF scheduler
DSQ
DSQ
BPF maps
BPF scheduler's userspace

scx_simple



```
SCX_OPS_DEFINE(simple_ops,
	       .select_cpu		= (void *)simple_select_cpu,
	       .enqueue			= (void *)simple_enqueue,
	       .dispatch		= (void *)simple_dispatch,
	       .running			= (void *)simple_running,
	       .stopping		= (void *)simple_stopping,
	       .enable			= (void *)simple_enable,
	       .init			= (void *)simple_init,
	       .exit			= (void *)simple_exit,
	       .name			= "simple");
```

- scx_simple.c
  - opens and load the BPF scheduler (scx_simple_open() and scx_simple_load())
  - enable BPF scheduler
- scx_simple.bpf.c



```
int main(int argc, char **argv)
{
	struct scx_simple *skel;
	struct bpf_link *link;
	__u32 opt;
	__u64 ecode;

	libbpf_set_print(libbpf_print_fn);
	signal(SIGINT, sigint_handler);
	signal(SIGTERM, sigint_handler);
restart:
	skel = SCX_OPS_OPEN(simple_ops, scx_simple);

	while ((opt = getopt(argc, argv, "fvh")) != -1) {
		switch (opt) {
		case 'f':
			skel->rodata->fifo_sched = true;
			break;
		case 'v':
			verbose = true;
			break;
		default:
			fprintf(stderr, help_fmt, basename(argv[0]));
			return opt != 'h';
		}
	}

	SCX_OPS_LOAD(skel, simple_ops, scx_simple, uei);
	link = SCX_OPS_ATTACH(skel, simple_ops, scx_simple);

	while (!exit_req && !UEI_EXITED(skel, uei)) {
		__u64 stats[2];

		read_stats(skel, stats);
		printf("local=%llu global=%llu\n", stats[0], stats[1]);
		fflush(stdout);
		sleep(1);
	}

	bpf_link__destroy(link);
	ecode = UEI_REPORT(skel, uei);
	scx_simple__destroy(skel);

	if (UEI_ECODE_RESTART(ecode))
		goto restart;
	return 0;
}
```

Example Schedulers

- Checkout:
- Simple
- scx_centrl
- scx_flatcg: a flattened cgroup hierarchy scheduler.
  - hierarchical weight-based cgroup CPU control
- scx_nest:
  - make scheduling decisions which encourage work to run on cores that are expected to have high frequency
  - optimize workloads that CPU utilization somewhat low, and which can benefit from running on a subset of cores on the host so as to keep the frequencies high on those cores
- scx_pair, scx_prev, scx_userland ...
- Rustland: https://github.com/sched-ext/scx/tree/main/scheds/rust/scx_rustland
  - prioritizes interactive workloads over CPU-intensive workloads
  - See the demo

Is sched_ext to replace CFS/EEVDF?

- Thoughts?