東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

scx_chaos: Pushing the sched_ext API to schedule badly

Jake Hillion

scx_chaos

東京 2025
LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

What is scx_chaos?

- A bad scheduler
- An extended version of scx_p2dq with added features to schedule badly
- Introduces scheduling chaos to hunt bugs and race conditions
- Makes it easier to reproduce hard to find conditions

Chaos features in scx_chaos

● ***Random Delays***	introduce random latency in scheduling decisions
● ***CPU Frequency Chaos***	randomly decrease CPU frequencies
● ***Slow Kprobes***	randomly delay a task after it hits a named kprobe
● ***Futex Inversion***	invert futex priorities (WIP)

What bugs can we find/reproduce?

```
int hsmp_send_message(struct hsmp_message *msg) {
  take_per_socket_lock();
  msg_socket->send_request(msg);
  while (!timed_out) {
      resp = msg_socket->gather_response();
      if (resp.ready)
          break;
      usleep_range(100, 2000); // sleep for 100-2000 micros
  }
  release_per_socket_lock();
}
```

Extending other schedulers

Why build on top of p2dq?

- I wanted to create a scheduler that introduced delays
- I didn't want to write scheduling logic
- p2dq had a friendly maintainer

What does it look like? (p2dq at start)

```
s32 BPF_STRUCT_OPS_SLEEPABLE(chaos_init_task,
                             struct task_struct *p,
                             struct scx_init_task_args *args)
{
    s32 ret = p2dq_init_task_impl(p, args);
    if (ret)
        return ret;

    return calculate_chaos_match(p);
}
```

What does it look like? (p2dq at end)

```
void BPF_STRUCT_OPS(chaos_dispatch, s32 cpu, struct task_struct *prev)
{
    bpf_for_each(scx_dsq, p, get_cpu_delay_dsq(-1), 0) {
        if (p->scx.dsq_vtime > now) { bpf_task_release(p); break; }
        // restore vtime to p2dq's timeline
        p->scx.dsq_vtime = taskc->p2dq_vtime;
        async_p2dq_enqueue_weak(&promise, p, taskc->enq_flags);
        complete_p2dq_enqueue_move(&promise, BPF_FOR_EACH_ITER, p);
        bpf_task_release(p);
    }
    return p2dq_dispatch_impl(cpu, prev);
}
```

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

scx_chaos

What does it look like? (p2dq in middle)

```
void BPF_STRUCT_OPS(chaos_enqueue, struct task_struct *p, u64 enq_flags)
{
   struct chaos_task_ctx *taskc = lookup_create_chaos_task_ctx(p);
   // Chaos setup: capture vtime before p2dq enqueue
   taskc->p2dq_vtime = p->scx.dsq_vtime;
   async_p2dq_enqueue(&promise, p, enq_flags);
   // Check if we should apply chaos delays (skips p2dq completion)
   if (taskc->next_trait == CHAOS_TRAIT_RANDOM_DELAYS &&enqueue_chaotic(p, enq_flags, taskc))
       return;
   // Modify p2dq's enqueue promise (degrade timeslice)
   if (taskc->next_trait == CHAOS_TRAIT_DEGRADATION)
       promise.slice_ns = degradation_frac * promise.slice_ns;
   complete_p2dq_enqueue(&promise, p);
}
```

Extension problems

The one struct_ops limitation

- Fixed fairly quickly with #ifdef
- Better libraries would help

Lack of interfaces

- p2dq was not designed to be a library
- Infinite potential side effects from calling p2dq functions
- Lack of contracts on the library interface
- What state does p2dq expect? What state does it modify?
- Partly solved: monorepo lets us test all p2dq changes against chaos

Chaotic selftests

Chaotic selftests

```
[jake@fedora-32gb-sin-1 linux]$ sudo make -C tools/testing/selftests TARGETS="cgroup"
run_tests 2>&1 | tee /tmp/clean_cgroup_tests.txt
[jake@fedora-32gb-sin-1 scx]$ sudo target/release/scx_chaos \
  --random-delay-frequency 0.2 --random-delay-min-us 100000 --random-delay-max-us 200000 &
[jake@fedora-32gb-sin-1 linux]$ sudo make -C tools/testing/selftests TARGETS="cgroup"
run_tests 2>&1 | tee /tmp/chaos_cgroup
[jake@fedora-32gb-sin-1 linux]$ cat /tmp/clean_cgroup_tests.txt | grep 'ok' | wc -l
559
[jake@fedora-32gb-sin-1 linux]$ cat /tmp/chaos_cgroup | grep 'ok' | wc -l
147
```

Bonus: Futex Delays

Dual key queues

The problem

- DSQs are only ordered by one key
- For Futex delays we need to track:
  - The futex that's delaying the task
  - The eventual timeout
- Can't efficiently look up by both keys

Potential solution

- Arena-based queues should fix this, but...
- Allocation in BPF isn't perfect yet
- Patches like non-sleepable allocations making this better

Implementation constraints

Can't preempt from tracepoint/syscalls

- Must maintain complex state across functions
- Have to preempt the task from a different callback

Can't call scx_bpf_dsq_move from enqueue

- Further constrains where we can manipulate task state

Result: Complex state machine

- Operations split across 3 different functions
- State must be maintained between all 3
- Lack of easy testing in BPF makes this hard to debug

東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025