Multi-kfunc sets

Scoping kfuncs to specific BPF struct_ops operations

David Vernet
Linux kernel engineer

Meta

Agenda

01 struct_ops background

02 Discussing feature

03 How do we do it?

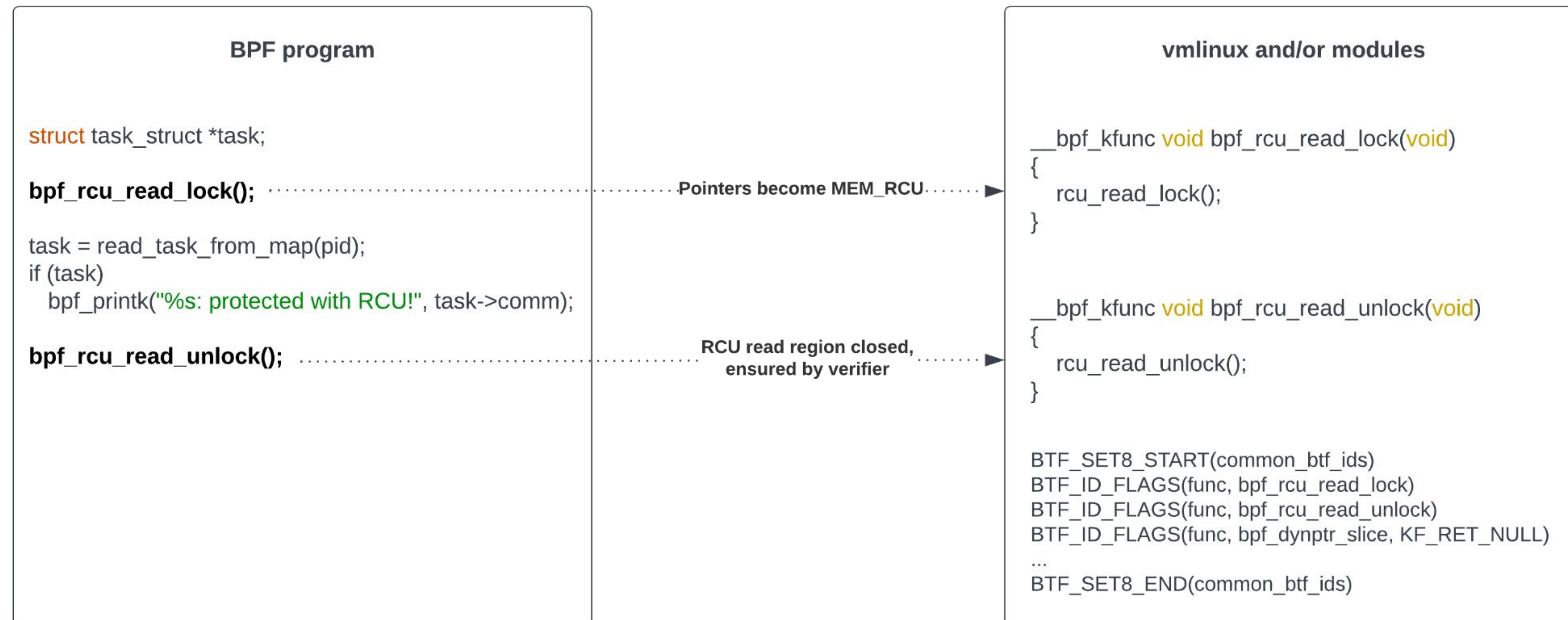
01 struct_ops background

struct_ops are structs with callbacks and flags that can be defined in BPF programs

- Allow BPF programs to implement interfaces for the kernel
- Written up in LWN in a **Kernel operations structures in BPF** article: https://lwn.net/Articles/811631/
- Currently used for TCP congestion control, and HID BPF (https://lwn.net/Articles/909109/), and will be used for sched_ext

struct_ops callbacks can invoke kfuncs

- Like any other BPF program, struct_ops callbacks can invoke kfuncs
- From example on prior slide, the `example_enqueue` BPF prog calls `scx_bpf_dispatch()` to put a task onto a sched_ext dispatch queue (DSQ)
- In example below, `bpf_rcu_read_lock()` and `bpf_rcu_read_unlock()` are also kfuncs



02 Discussing feature

kfuncs may not be safe to call in all contexts

- kfuncs may have assumptions of context: it's only valid to call me from certain places
  - E.g. the kernel may set some global state before invoking a struct_ops callback, which a kfunc relies on
- It can therefore be unsafe to call kfuncs from certain callbacks
  - In prior example, it's valid to call `scx_bpf_dispatch()` from an `ops.enqueue()` callback, when the task is being enqueued
  - It would be invalid to call `scx_bpf_dispatch()` from e.g. `ops.select_cpu()`, where a CPU is returned where a task should be migrated on the wakeup path
  - It would be invalid to call *any* sched_ext kfunc from a non-sched-ext struct_ops program
- A kfunc that can be called from a `BPF_PROG_TYPE_STRUCT_OPS` program can be called from *any* struct_ops callback
  - Verifier will only ensure that the kfunc may be called from *a* struct_ops program
  - No filtering for which struct_ops program or callback should be allowed
- kfuncs may also nest, e.g.
  - vmlinux → struct_ops callback `X` → kfunc `A` → struct_ops callback `Y` → kfunc `B`
  - It may only be valid to call kfunc `A` from callback `X`
  - It may be that no nesting is expected from kfunc `B`

Support restricting kfunc scope to specific struct_ops callbacks

- Support specifying specific struct_ops callbacks for individual kfuncs
- Possibly support statically specifying how kfuncs may be nested
  - Only for runtime safety checking
    - Specifying struct_ops callback → kfunc permissions is sufficiently expressive so as to define allowed nesting
- Possibly support invoking different kfuncs in different contexts, but with same name in BPF program
  - `scx_bpf_dispatch()` in `ops.enqueue()` vs. `ops.dispatch()` may correspond to different kfuncs in ext.c

03 How do we do it?

sched_ext uses bits in global mask to track which kfuncs may be invoked

```
/*
 * Mask bits for scx_entity.kf_mask. Not all kfuncs can be called from
 * everywhere and the following bits track which kfunc sets are currently
 * allowed for %current. This simple per-task tracking works because SCX ops
 * nest in a limited way. BPF will likely implement a way to allow and disallow
 * kfuncs depending on the calling context which will replace this manual
 * mechanism. See scx_kf_allow().
 */
enum scx_kf_mask {
	SCX_KF_UNLOCKED		= 0,	  /* not sleepable, not rq locked */
	/* all non-sleepables may be nested inside INIT and SLEEPABLE */
	SCX_KF_INIT		= 1 << 0, /* running ops.init() */
	SCX_KF_SLEEPABLE	= 1 << 1, /* other sleepable init operations */
	/* ENQUEUE and DISPATCH may be nested inside CPU_RELEASE */
	SCX_KF_CPU_RELEASE	= 1 << 2, /* ops.cpu_release() */
	/* ops.dequeue (in REST) may be nested inside DISPATCH */
	SCX_KF_DISPATCH		= 1 << 3, /* ops.dispatch() */
	SCX_KF_ENQUEUE		= 1 << 4, /* ops.enqueue() */
	SCX_KF_REST		= 1 << 5, /* other rq-locked operations */

	__SCX_KF_RQ_LOCKED	= SCX_KF_CPU_RELEASE | SCX_KF_DISPATCH |
				  SCX_KF_ENQUEUE | SCX_KF_REST,
	__SCX_KF_TERMINAL	= SCX_KF_ENQUEUE | SCX_KF_REST,
};
```

sched_ext uses bits in global mask to track which kfuncs may be invoked

```
#define SCX_CALL_OP(mask, op, args...)
do {
        if (mask) {
                scx_kf_allow(mask);
                scx_ops.op(args);
                scx_kf_disallow(mask);
        } else {
                scx_ops.op(args);
        }
} while (0)

#define SCX_CALL_OP_RET(mask, op, args...)
({
        __typeof__(scx_ops.op(args)) __ret;
        if (mask) {
                scx_kf_allow(mask);
                __ret = scx_ops.op(args);
                scx_kf_disallow(mask);
        } else {
                __ret = scx_ops.op(args);
        }
        __ret;
})
```

```
/*
 * scx_kf_mask enforcement. Some kfuncs can only be called from specific SCX
 * ops. When invoking SCX ops, SCX_CALL_OP[_RET]() should be used to indicate
 * the allowed kfuncs and those kfuncs should use scx_kf_allowed() to check
 * whether it's running from an allowed context.
 *
 * @mask is constant, always inline to cull the mask calculations.
 */
static __always_inline void scx_kf_allow(u32 mask)
{
        /* nesting is allowed only in increasing scx_kf_mask order */
        WARN_ONCE((mask | higher_bits(mask)) & current->scx.kf_mask,
                  "invalid nesting current->scx.kf_mask=0x%x mask=0x%x\n",
                  current->scx.kf_mask, mask);
        current->scx.kf_mask |= mask;
}

static void scx_kf_disallow(u32 mask)
{
        current->scx.kf_mask &= ~mask;
}
```

03 How do we do it?

```
/* @mask is constant, always inline to cull unnecessary branches */
static __always_inline bool scx_kf_allowed(u32 mask)
{
	if (unlikely(!(current->scx.kf_mask & mask))) {
		scx_ops_error("kfunc with mask 0x%x called from an operation only allowing 0x%x",
			      mask, current->scx.kf_mask);
		return false;
	}

	if (unlikely((mask & (SCX_KF_INIT | SCX_KF_SLEEPABLE)) &&
		     in_interrupt())) {
		scx_ops_error("sleepable kfunc called from non-sleepable context");
		return false;
	}

	/*
	 * Enforce nesting boundaries. e.g. A kfunc which can be called from
	 * DISPATCH must not be called if we're running DEQUEUE which is nested
	 * inside ops.dispatch(). We don't need to check the SCX_KF_SLEEPABLE
	 * boundary thanks to the above in_interrupt() check.
	 */
	if (unlikely(highest_bit(mask) == SCX_KF_CPU_RELEASE &&
		     (current->scx.kf_mask & higher_bits(SCX_KF_CPU_RELEASE)))) {
		scx_ops_error("cpu_release kfunc called from a nested operation");
		return false;
	}

	if (unlikely(highest_bit(mask) == SCX_KF_DISPATCH &&
		     (current->scx.kf_mask & higher_bits(SCX_KF_DISPATCH)))) {
		scx_ops_error("dispatch kfunc called from a nested operation");
		return false;
	}

	return true;
}
```

sched_ext uses bits in global mask to track which kfuncs may be invoked

```
/**
 * scx_bpf_dispatch_nr_slots - Return the number of remaining dispatch slots
 *
 * Can only be called from ops.dispatch().
 */
u32 scx_bpf_dispatch_nr_slots(void)
{
	if (!scx_kf_allowed(SCX_KF_DISPATCH))
		return 0;

	return scx_dsp_max_batch - __this_cpu_read(scx_dsp_ctx.buf_cursor);
}
```

Can we define flags per callback, optionally specify masks of callback flags in kfuncs?

- Could it go into another `.long` following where we store the kfunc flags in BTF?
- Verifier then ensures struct_ops ←→ kfunc loops form a DAG, and kfuncs are invoked in correct place?

Meta