BPF Qdisc

Amery Hung

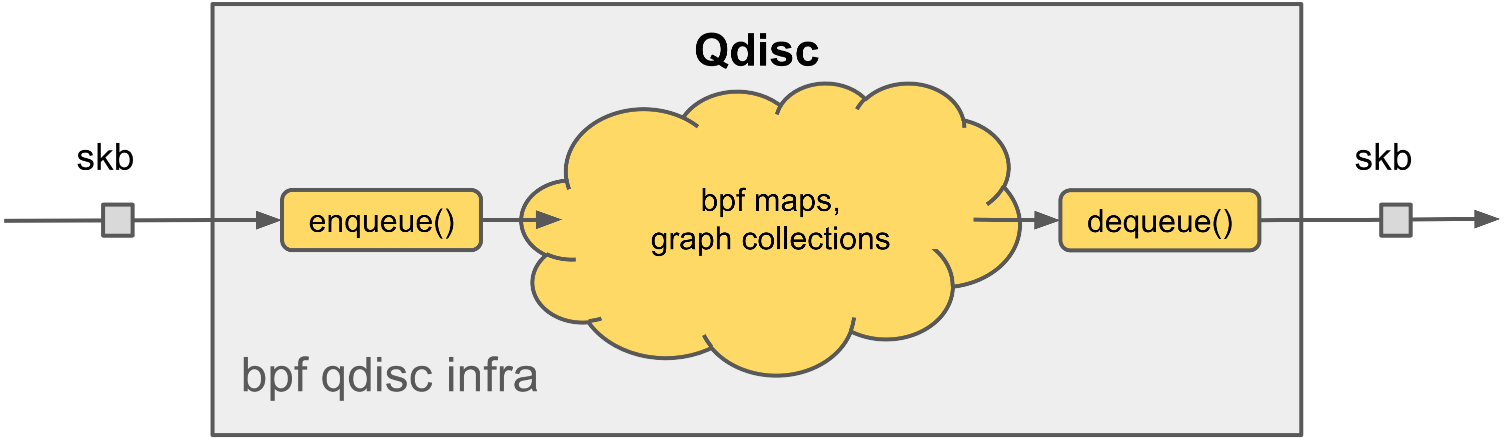
System Technologies & Engineering, ByteDance

ByteDance字节跳动

- Background
- BPF changes
- BPF Qdisc kfuncs
- Examples and evaluation

Enable users to innovate in Qdisc and beyond

- Flexibility: Allow users implement core Qdisc logic using bpf
- Ease-of-use: Implement the mundane part for the user in bpf Qdisc infra

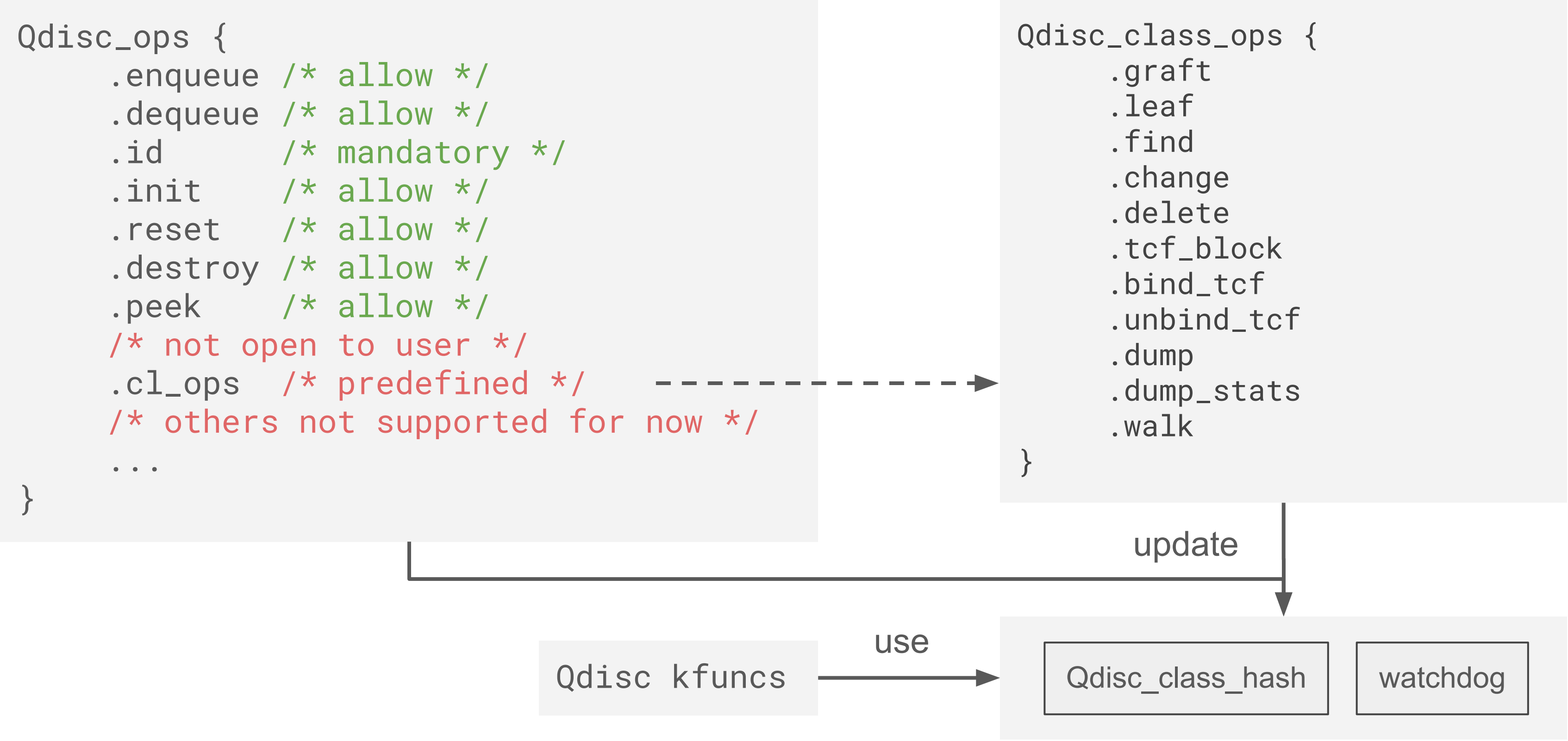


Only require the user to implement the core logic

```
Qdisc_ops {
     .enqueue /* allow */
     .dequeue /* allow */
     .id      /* mandatory */
     .init    /* allow */
     .reset   /* allow */
     .destroy /* allow */
     .peek    /* allow */
     /* not open to user */
     .cl_ops  /* predefined */
     /* others not supported for now */
     ...
}
```

Todo: look into more operators

Implement the mundane part for users

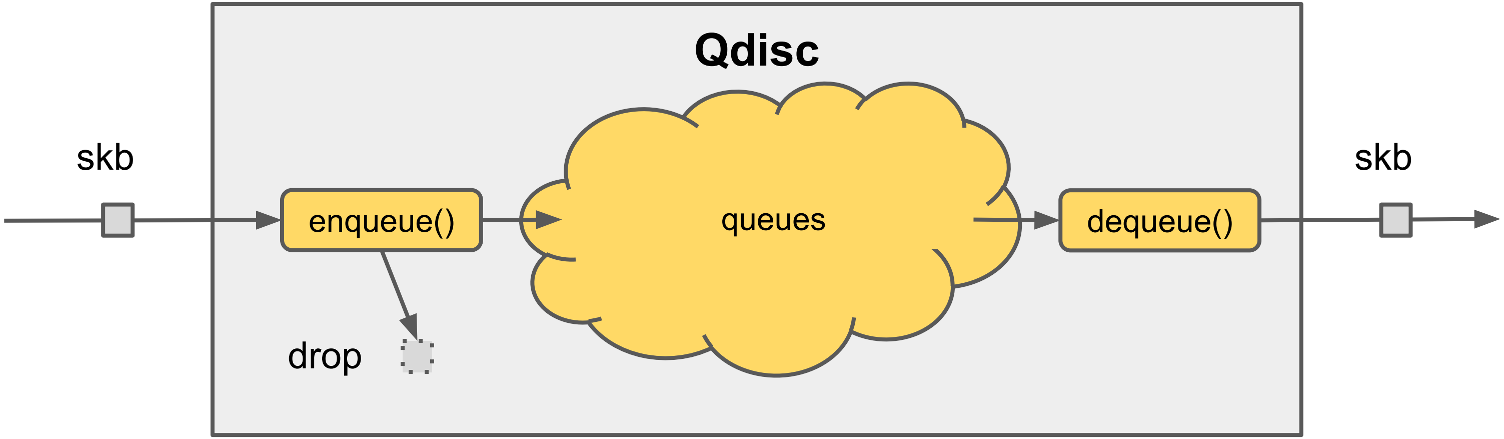


```
Qdisc_ops {
     .enqueue /* allow */
     .dequeue /* allow */
     .id      /* mandatory */
     .init    /* allow */
     .reset   /* allow */
     .destroy /* allow */
     .peek    /* allow */
     /* not open to user */
     .cl_ops  /* predefined */
     /* others not supported for now */
     ...
}
```

```
Qdisc_class_ops {
     .graft
     .leaf
     .find
     .change
     .delete
     .tcf_block
     .bind_tcf
     .unbind_tcf
     .dump
     .dump_stats
     .walk
}
```

Lifecycle of skb

- An skb passed to enqueue is either enqueued or dropped
- At dequeue, an skb is removed from queue and returned

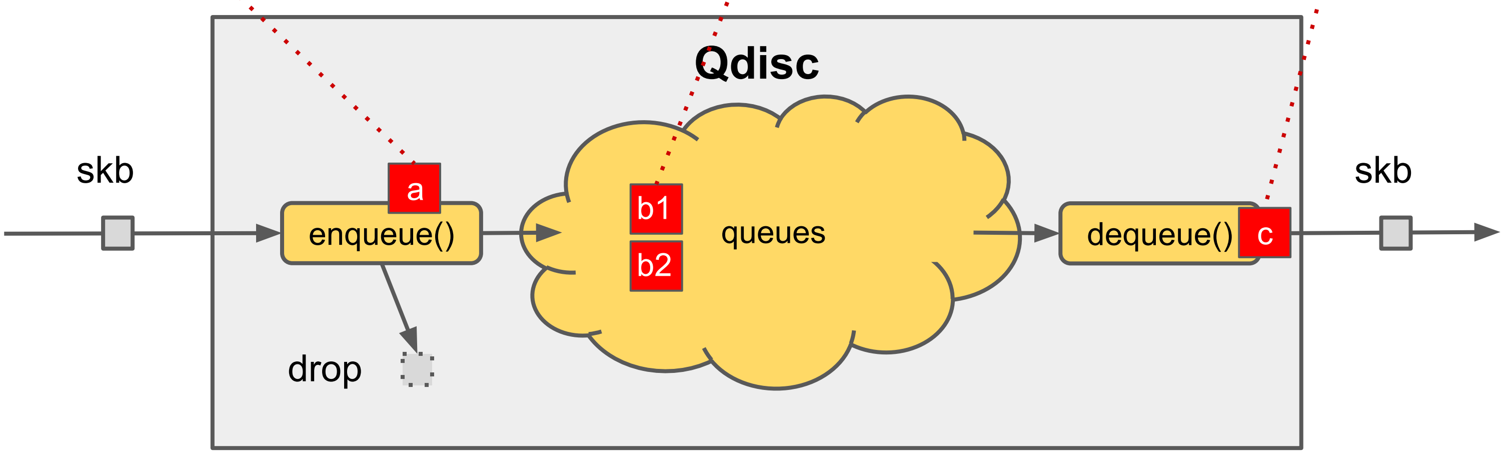


Lifecycle of skb

Duplicate reference from the same skb and dequeue one
→ dangling pointer in queue

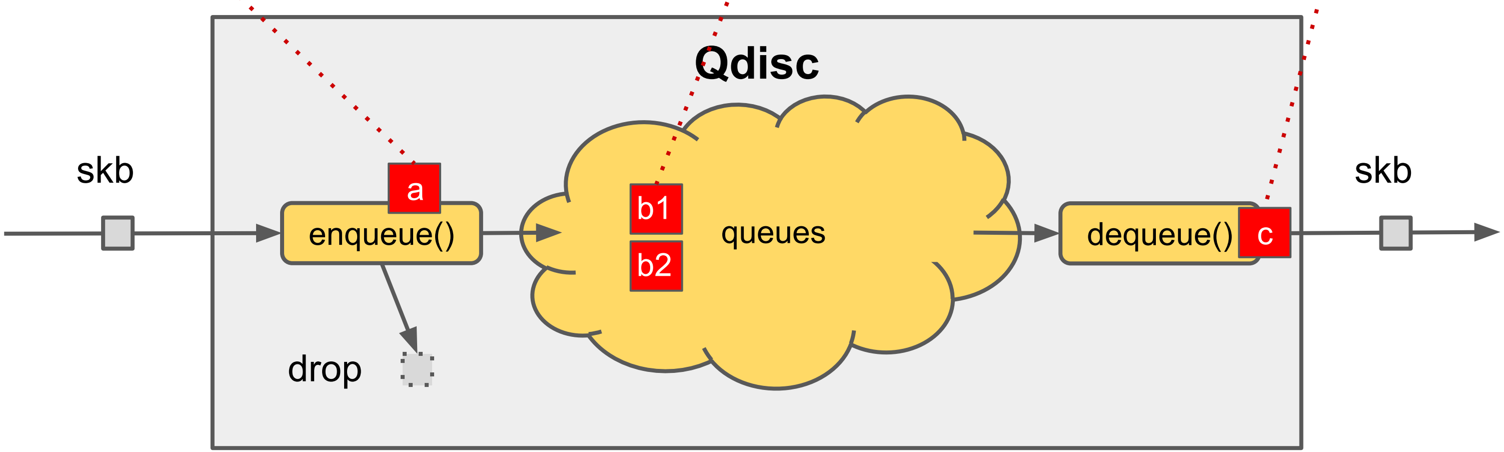
Neither enqueued nor dropped
→ reference leak

Dequeue make up a invalid skb ptr
→ kernel crash



a & c: referenced kptr, b: unique reference

Lifecycle of skb



Neither enqueued nor dropped
→ reference leak

Duplicate reference to the same skb
and dequeue one
→ dangling pointer in queue

Dequeue make up a invalid skb ptr
→ kernel crash

Status

- RFC v7
  - Make it run
  - Try to take care of the lifecycle of skb
  - Show that a sophisticated qdisc can be implemented with bpf qdisc (fq with bpf_{list, rbtree})
- RFC v8
  - Switch to struct_ops
  - Extend struct_ops to make it work better with qdisc
  - Support directly adding skb to bpf_list and bpf_rbtree
  - Add selftests: fifo, fq, netem and prio

*No production test yet; All test results are iperf client→ bpf qdisc → loopback → server

[RFC v8] https://lore.kernel.org/netdev/20240510192412.3297104-1-amery.hung@bytedance.com/

BPF changes

1/4

Allow getting a referenced kptr from a struct_ops argument

- Consider

```
int enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **to_free)
```

- skb is solely owned by the qdisc, and if is not enqueued must be dropped
  I.e., referenced kptr in BPF

- The common kfunc with KF_ACQUIRE approach does not work well with the unique reference semantic

Solution

- Annotate argument in the stub function with “ref_acquired” [2]

```
static int Qdisc_ops__enqueue(struct sk_buff *skb__ref_acquired,
                              struct Qdisc *sch, struct sk_buff **to_free)
```

- Teach verifier to acquire a reference (i.e., ref_obj_id > 0) for this argument
- Save the ref_obj_id in bpf_ctx_arg_aux and swap out in bpf_ctx_access(), and reject any subsequent accesses.

[2] [PATCH bpf-next v8 0/4] Support PTR_MAYBE_NULL for struct_ops arguments.

How about unique reference?

- Achieved through the combination of “ref_acquired” and “preventing any reference duplication mechanism for skb in qdisc“

Questions:

- Is ref_acquired a proper semantic here?
- Should we have an explicit semantic in the verifier (e.g., PTR_UNIQUE) and corresponding enforcement mechanism?

Allow returning referenced kptr from struct_ops programs

- Consider

```
struct sk_buff *dequeue(struct Qdisc *sch)
```

- If the function return type is pointer, allow return a referenced kptr or null

Solution

- First, in check_reference_leak(), allow the referenced object in the return register to leak if the BTF id matches the function return type
- Then, in check_return_code(), check:

    if reg->type == PTR_TO_BTF_ID, the ptr is in unmodified form

    if reg->type == SCALAR_VALUE, the value must be zero

Solution

- First, in check_reference_leak(), allow the referenced object in the return register to leak if the BTF id matches the function return type
- Then, in check_return_code(), check:

    if reg->type == PTR_TO_BTF_ID, the ptr is in unmodified form

    if reg->type == SCALAR_VALUE, the value must be zero

Question

- The may_be_null assumption might not work for others, and currently there is no way to annotate return in stub function

Support adding sk_buff to bpf graph collections

- Currently an skb kptr needs to be stored into a local object and then get enqueued to maps or collections
- Performance overhead and less ergonomic

.enqueue in RFC v7

```
skb_node = bpf_obj_new(typeof(*skb_node));
if (!skb_node)
      goto out;

old = bpf_kptr_xchg(&skb_node->skb, skb);
if (old)
      bpf_skb_release(old);

bpf_spin_lock(&queue_lock);
bpf_list_push_back(&queue, &skb_node->node);
bpf_spin_unlock(&queue_lock);
```

The goal

```
bpf_spin_lock(&queue_lock);
bpf_list_push_back(&queue, &skb->list_head);
bpf_spin_unlock(&queue_lock);
```

Two steps

1. Teach bpf to allow adding kernel objects to collections
2. Resolve incompatibility between sk_buff and bpf_rb_node

Teach bpf to allow adding kernel objects to collections

- Generate btf_srtuct_metas for kernel and kernel module BTF
    - Searching for special BTF fields: Allowlist for kernel; All for kernel module and program
- Teach btf and verifier to recognize graph nodes in kernel btf
    - Use contains_kptr to annotate and store btf of graph nodes in btf_field_info
- Allowing adding kernel objects to collections
    - Teach verifier that graph nodes can be PTR_TO_BTF_ID in addition to (PTR_TO_BTF_ID | MEM_ALLOC)

Resolve incompatibility between sk_buff and bpf_rb_node

- bpf_rb_node does not fit into the union at offset=0 due to the additional “owner” field [3]
- Besides, bpf_rb_node and bpf_list_node cannot coexist

```
struct sk_buff {
        union {
                ...
                struct rb_node rbnode;
                struct list_head list;
        }
        ...
};
```

```
struct bpf_rb_node_kern {
        struct rb_node rb_node;
        void *owner;
};
```

[3] [PATCH v2 bpf-next 0/6] BPF Refcount followups 2: owner field

3/4

Alternative exclusive ownership

- Introduce bpf_rb_excl_node and bpf_list_excl_node, structures wrapping around rb_node and list_head for BTF annotation
- Cannot coexist with typical bpf_rb_node, bpf_list_node or bpf_refcount in the same struct
- Allow two _excl_ field to be at the same offset
- Graph kfuncs can skip owner checks if the graph contains exclusive-ownership nodes

Discussion

- Todo: restore skb->dev
  - Doing fixup during verification that automatically inserts a call to bpf_qdisc_skb_set_dev(), and then fingers crossed that skb is the only one that need fixup
  - Seem too complicated. Any suggestions?
- Maybe argument-dependent polymorphic kfunc?
- Or, the simple but ugly(?) way: skb-flavor graph kfuncs

```
bpf_skb_list_push_back()
bpf_skb_list_push_front()
bpf_skb_list_pop_back()
bpf_skb_list_pop_front()
bpf_skb_rbtree_add()
bpf_skb_rbtree_remove()
bpf_skb_rbtree_first()
```

Customizable struct_ops entry/exit routines?

- Goal: Provide value to the user in addition to letting users implement kernel code in BPF
- BPF qdisc implement the common Qdisc_class_ops for the user, which require some work in different ops

```
    kernel code

+   do_something_before_op()

    struct_ops->op()

+   do_something_after_op()

    kernel code
```

4/4

```
int bpf_qdisc_init_pre_op(struct Qdisc *sch, struct nlattr *opt,
                          struct netlink_ext_ack *extack)
{
        struct bpf_sched_data *q = qdisc_priv(sch);
        int err;

        qdisc_watchdog_init(&q->watchdog, sch);

        err = tcf_block_get(&q->block, &q->filter_list, sch, extack);
        if (err)
                return err;

        err = qdisc_class_hash_init(&q->clhash);
        if (err < 0)
                return err;

        return 0;
}
```

net/sched/bpf_qdisc.c

```
#if defined(CONFIG_BPF_SYSCALL) && defined(CONFIG_BPF_JIT)
        if (ops->cl_ops == &sch_bpf_class_ops) {
                err = bpf_qdisc_init_pre_op(sch, tca[TCA_OPTIONS], extack);
                if (err != 0)
                        goto err_out4;
        }
#endif
        if (ops->init) {
                err = ops->init(sch, tca[TCA_OPTIONS], extack);
                if (err != 0)
                        goto err_out4;
        }
```

net/sched/sch_api.c

Qdisc kfuncs

```
/* temporary hack */
BTF_ID_FLAGS(func, bpf_skb_set_dev)

/* skb classification */
BTF_ID_FLAGS(func, bpf_skb_get_hash)
BTF_ID_FLAGS(func, bpf_skb_tc_classify)

/* releasing skb */
BTF_ID_FLAGS(func, bpf_skb_release, KF_RELEASE)
BTF_ID_FLAGS(func, bpf_qdisc_skb_drop, KF_RELEASE)

/* throttling */
BTF_ID_FLAGS(func, bpf_qdisc_watchdog_schedule)

/* classful qdisc manipulation */
BTF_ID_FLAGS(func, bpf_qdisc_create_child)
BTF_ID_FLAGS(func, bpf_qdisc_find_class)
BTF_ID_FLAGS(func, bpf_qdisc_enqueue, KF_RELEASE)
BTF_ID_FLAGS(func, bpf_qdisc_dequeue, KF_ACQUIRE | KF_RET_NULL)
```

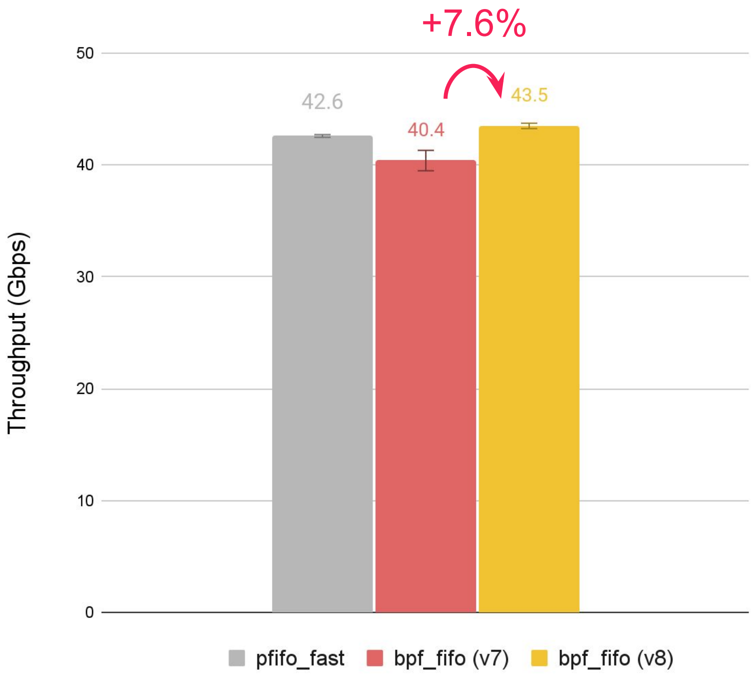
Example 1

BPF FIFO qdisc

```c
#define private(name) SEC(".data." #name) __hidden __attribute__((aligned(8)))

private(B) struct bpf_spin_lock q_fifo_lock;
private(B) struct bpf_list_head q_fifo __contains_kptr(sk_buff, bpf_list);

SEC("struct_ops/bpf_fifo_enqueue")
int BPF_PROG(bpf_fifo_enqueue, struct sk_buff *skb, struct Qdisc *sch,
             struct bpf_sk_buff_ptr *to_free)
{
        ...
}

SEC("struct_ops/bpf_fifo_dequeue")
struct sk_buff *BPF_PROG(bpf_fifo_dequeue, struct Qdisc *sch)
{
        ...
}

SEC(".struct_ops")
struct Qdisc_ops fifo = {
        .enqueue   = (void *)bpf_fifo_enqueue,
        .dequeue   = (void *)bpf_fifo_dequeue,
        .id        = "bpf_fifo",
};
```

```c
SEC("struct_ops/bpf_fifo_enqueue")
int BPF_PROG(bpf_fifo_enqueue, struct sk_buff *skb, struct Qdisc *sch,
             struct bpf_sk_buff_ptr *to_free)
{
        bpf_spin_lock(&q_fifo_lock);
        bpf_list_excl_push_back(&q_fifo, &skb->bpf_list);
        bpf_spin_unlock(&q_fifo_lock);

        return NET_XMIT_SUCCESS;
}

SEC("struct_ops/bpf_fifo_dequeue")
struct sk_buff *BPF_PROG(bpf_fifo_dequeue, struct Qdisc *sch)
{
        struct sk_buff *skb;
        struct bpf_list_so_node *node;

        bpf_spin_lock(&q_fifo_lock);
        node = bpf_list_excl_pop_front(&q_fifo);
        bpf_spin_unlock(&q_fifo_lock);
        if (!node)
                return NULL;

        skb = container_of(node, struct sk_buff, bpf_list);
        return skb;
}
```

Support adding sk_buff’s to bpf collections is a key to make bpf qdisc performant



+7.6%

Throughput of qdiscs on a loopback device

Recap

- bpf qdisc is now realized with struct_ops and some proposed changes to bpf
- Made bpf qdisc more performant
- bpf qdisc simplifies qdisc development
- Cross-components communication via bpf maps open new opportunities for new applications and optimizations

[RFC v8] https://lore.kernel.org/netdev/20240510192412.3297104-1-amery.hung@bytedance.com/

What’s next

- Production test
- kfunc availability checks
- Support qdisc statistics
- Explore support of other Qdisc_ops
- Support updating Qdisc_ops

[RFC v8] https://lore.kernel.org/netdev/20240510192412.3297104-1-amery.hung@bytedance.com/