Local-storage user space mapping

Allowing user space to map entries in local storage (task, cgroup, etc) maps

David Vernet
Linux kernel engineer

Meta

Agenda

01 Local storage background

02 Discussing feature

03 How do we do it?

Disclaimer: This would be extremely complex, and needs a lot more thought and details

01 Local storage background

Some maps allow programs to allocate “local storage” entries for certain objects

- **BPF_MAP_TYPE_TASK_STORAGE**: Local storage entries for individual struct task_struct * objects
- **BPF_MAP_TYPE_CGRP_STORAGE**: Local storage entries for individual struct cgroup * objects

sched_ext example: task local storage

```
/* Per-task scheduling context */
struct task_ctx {
        bool    force_local;    /* Dispatch directly to local_dsq */
        struct bpf_cpumask __kptr * cpumask; /* CPUs where task may run. */
};

struct {
        __uint(type, BPF_MAP_TYPE_TASK_STORAGE);
        __uint(map_flags, BPF_F_NO_PREALLOC);
        __type(key, int);
        __type(value, struct task_ctx);
} task_ctx_stor SEC(".maps");
```

sched_ext example: Allocating task local storage

```
s32 BPF_STRUCT_OPS(qmap_prep_enable, struct task_struct *p,
                   struct scx_enable_args *args)
{
        /*
         * @p is new. Let's ensure that its task_ctx is available. We can sleep
         * in this function and the following will automatically use GFP_KERNEL.
         */
        if (bpf_task_storage_get(&task_ctx_stor, p, 0,
                                 BPF_LOCAL_STORAGE_GET_F_CREATE))
                return 0;
        else
                return -ENOMEM;
}
```

sched_ext example: Setting and querying task local storage

```
s32 BPF_STRUCT_OPS(qmap_select_cpu, struct task_struct *p,
                   s32 prev_cpu, u64 wake_flags)
{
        struct task_ctx *tctx;
        s32 cpu;

        tctx = bpf_task_storage_get(&task_ctx_stor, p, 0, 0);
        if (!tctx) {
                scx_bpf_error("task_ctx lookup failed");
                return -ESRCH;
        }

        if (p->nr_cpus_allowed == 1 ||
            scx_bpf_test_and_clear_cpu_idle(prev_cpu)) {
                tctx->force_local = true;
                return prev_cpu;
        }

        cpu = scx_bpf_pick_idle_cpu(p->cpus_ptr);
        if (cpu >= 0)
                return cpu;

        return prev_cpu;
}
```

```
void BPF_STRUCT_OPS(qmap_enqueue, struct task_struct *p, u64 enq_flags)
{
        struct task_ctx *tctx;
        u32 pid = p->pid;
        int idx = weight_to_idx(p->scx.weight);
        void *ring;

        tctx = bpf_task_storage_get(&task_ctx_stor, p, 0, 0);
        if (!tctx) {
                scx_bpf_error("task_ctx lookup failed");
                return;
        }

        /*
         * If qmap_select_cpu() is telling us to or this is the last runnable
         * task on the CPU, enqueue locally.
         */
        if (tctx->force_local || (enq_flags & SCX_ENQ_LAST)) {
                tctx->force_local = false;
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, slice_ns, enq_flags);
                return;
        }

...
```

sched_ext example: Setting and querying task local storage

```
s32 BPF_STRUCT_OPS(qmap_select_cpu, struct task_struct *p,
                   s32 prev_cpu, u64 wake_flags)
{
        struct task_ctx *tctx;
        s32 cpu;

        tctx = bpf_task_storage_get(&task_ctx_stor, p, 0, 0);
        if (!tctx) {
                scx_bpf_error("task_ctx lookup failed");
                return -ESRCH;
        }

        if (p->nr_cpus_allowed == 1 ||
            scx_bpf_test_and_clear_cpu_idle(prev_cpu)) {
                tctx->force_local = true;
                return prev_cpu;
        }

        cpu = scx_bpf_pick_idle_cpu(p->cpus_ptr);
        if (cpu >= 0)
                return cpu;

        return prev_cpu;
}
```

```
void BPF_STRUCT_OPS(qmap_enqueue, struct task_struct *p, u64 enq_flags)
{
        struct task_ctx *tctx;
        u32 pid = p->pid;
        int idx = weight_to_idx(p->scx.weight);
        void *ring;

        tctx = bpf_task_storage_get(&task_ctx_stor, p, 0, 0);
        if (!tctx) {
                scx_bpf_error("task_ctx lookup failed");
                return;
        }

        /*
         * If qmap_select_cpu() is telling us to or this is the last runnable
         * task on the CPU, enqueue locally.
         */
        if (tctx->force_local || (enq_flags & SCX_ENQ_LAST)) {
                tctx->force_local = false;
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, slice_ns, enq_flags);
                return;
        }

...
```

Local storage stored as struct bpf_local_storage __rcu object

- Fast path is O(1) lookup in a 16-element hashmap cache
  - Depends on if other programs are also using local storage, but in practice this is almost always hit
- Slow path is O(n) iteration over all local storage entries in a list
  - Added to cache after lookup, requires taking IRQ spinlock to avoid racing with deletion
- New local storage entries are allocated (when BPF_LOCAL_STORAGE_GET_F_CREATE is set). struct bpf_local_storage_elem (actual local storage) allocated `with bpf_selem_alloc()` → `bpf_mem_cache_alloc_flags()`

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        unsigned int                    __state;

...

#ifdef CONFIG_BPF_SYSCALL
        /* Used by BPF task local storage */
        struct bpf_local_storage __rcu  *bpf_storage;
        /* Used for BPF run context */
        struct bpf_run_ctx              *bpf_ctx;
#endif

...

}
```

02 Discussing feature

User space sometimes needs a shared per-thread map

- sched_ext: Used when load balancing in Atropos: https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/atropos/src/main.rs#L193
  - Can’t map local storage map, so instead they use a statically sized hashmap indexed by pid:
    - https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/atropos/src/bpf/atropos.bpf.c#L121-L128
    - https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/atropos/src/bpf/atropos.h#L34-L42
  - Also uses another hashmap to indicate which pids should be assigned to new domains:
    - https://github.com/sched-ext/sched_ext/blob/sched_ext/tools/sched_ext/atropos/src/bpf/atropos.bpf.c#L130-L140
- ghOSt apparently uses something similar, but with a BPF_MAP_TYPE_ARRAY

Statically sized maps work, but it’s wasteful and a poor UX

- Need to statically allocate all pages backing the map when prog is opened
  - Either wasteful, or insufficient
- Poor UX – every caller has to do essentially the same thing
  - Create map that’s indexed by pid (or cgroup id if doing this for cgroup local-storage map)
  - Add lookup wrappers in both kernel and user space that either maps a task pid to an index in an array, or statically sizes the array map to contain all possible pids with the pid being a direct index

03 How do we do it?

Create new local storage allocator?

- Local storage entries are put into pages that are allocated in a kfunc
- Pages are mapped contiguously in user space
  - Full region must be mmap’ed when map is created
    - Most pages will be unused – we could have unused pages in mapping not be present? Core BPF would be responsible for managing this mapping, preventing fragmentation in the allocator, publishing mapping changes to libbpf, etc.
- User space would have IDR layer, mapping some handle / fd to a local storage entry
  - APIs provided by libbpf

- Is this realistic at all? Seems very complex, but could be a really nifty feature


Meta