Polymorphic Kfuncs

Context-aware kfunc relocations

David Vernet
Kernel engineer

Meta

Agenda

01 Background and motivation

02 Design proposal

01 Background and motivation

BPF programs use kfuncs to call into vmlinux (or modules)

- Conceptually similar to BPF helpers (not UAPI bound)
- Provide abstractions to BPF programs to access kernel objects and logic

Some kfuncs are basic building blocks

- Not particular to any specific program type
- Have well defined, universal semantics

- `bpf_task_acquire()` / `bpf_task_release()` -> Acquire and release a struct task_struct kptr
- `bpf_rbtree_first()` / `bpf_rbtree_add_impl()`... -> Use rbtrees in BPF prog

Some kfuncs have context-specific semantics

- Only applicable to specific program types, e.g. struct_ops programs
- Semantics may depend on where a kfunc is being invoked from
  - struct_ops prog A expects different behavior than struct_ops prog B

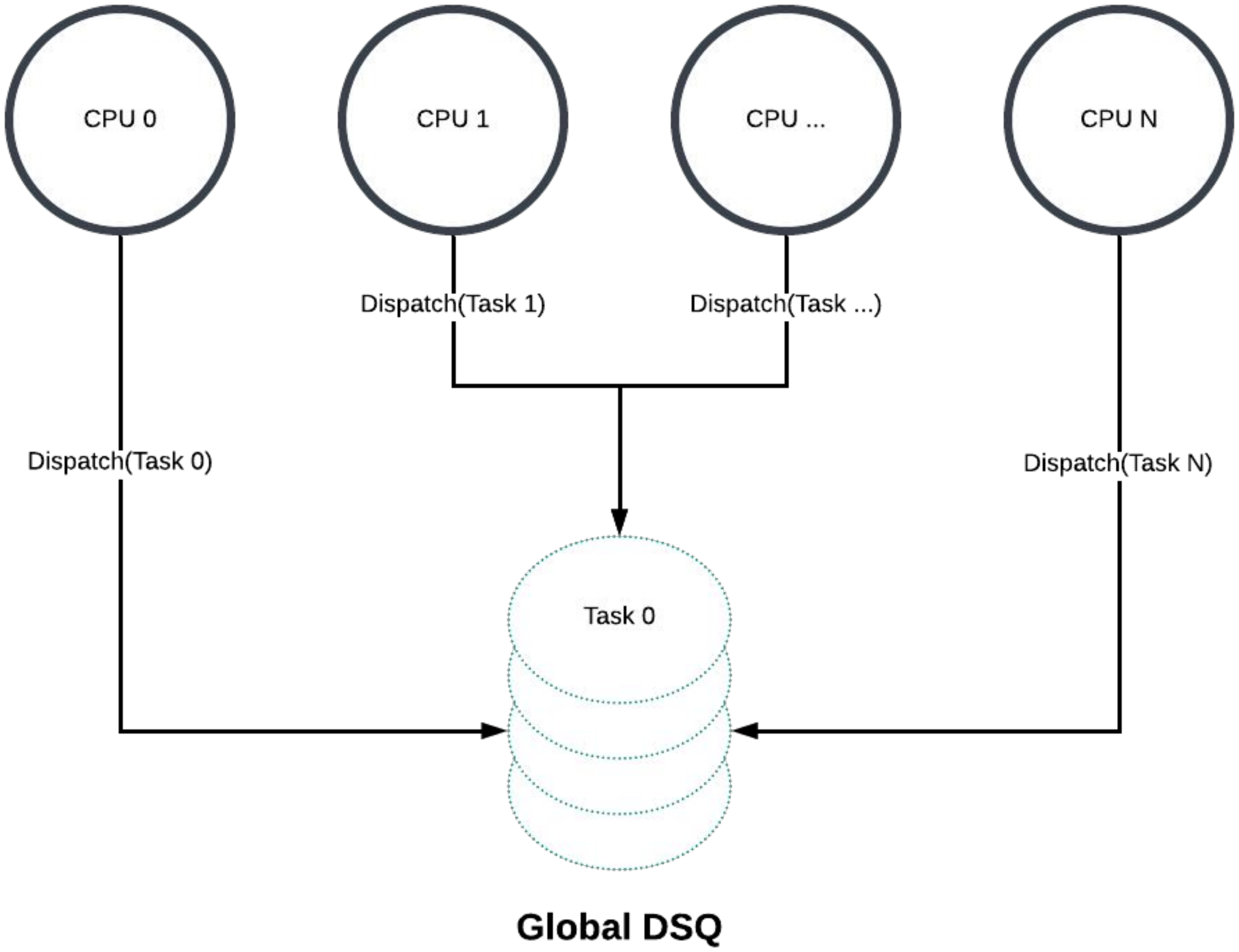
Quick aside: Dispatch Queues

Dispatch Queues (DSQs) are basic building block of scheduler policies

- Conceptually similar to runqueue
- Every core has a special “local” DSQ called SCX_DSQ_LOCAL
- Otherwise, can create as many or as few as needed
  - Gives schedulers flexibility
    - Per-domain (NUMA node, CCX, etc) DSQ?
    - Global DSQ?
    - Per-cgroup DSQ?
- The data structure / abstraction layer for managing tasks between main kernel <-> BPF scheduler (more on next slide).

Example 0: Global FIFO – enqueuing

- Scheduler "**dispatches**" tasks to global DSQ at enqueue time
  - *Not* where tasks are pulled from when being scheduled in
  - Task must be in local DSQ to be chosen to run
  - Dispatching is done with `scx_bpf_dispatch()` kfunc



scx_bpf_dispatch() has different semantics in different contexts

- sched_ext struct_ops map has many callbacks defined, including:
  - `ops.select_cpu()`: Choose a CPU to migrate a task to at wakeup or fork time
  - `ops.enqueue()`: Enqueue a task in the scheduler
  - …
  - `ops.dispatch()`: CPU out of tasks to run, choose a new one
- scx_bpf_dispatch() behaves differently in `ops.select_cpu()` and `ops.enqueue()`, compared to `ops.dispatch()`

ops.select_cpu() + ops.enqueue()

- May not drop task CPU's rq lock
  - Cannot dispatch directly to remote CPU
  - **Can** dispatch directly to local CPU
- Dispatch is "direct"
  - Task is dispatched directly from enqueue, rather than being enqueued in the BPF scheduler
  - scx_bpf_dispatch() records per-CPU variable to mark dispatch choice, consumes later on in scheduling pipeline
  - Only a single task can be dispatched from this CPU within prog scope

ops.dispatch()

- **May** drop task CPU's rq lock
  - **Can** dispatch directly to remote CPU by doing lock dropping + reacquire
  - Can also dispatch locally
- Dispatch is not direct
  - Task is dispatched directly from enqueue, rather than being enqueued in the BPF scheduler
  - Many tasks can be dispatched, one after the other

ops.select_cpu() + ops.enqueue()

- Implementation enforces only calling waking/enqueuing task can be dispatched if called from that CPU
- Uses different logic to record dispatch decision. Everything is tracked with per-CPU data structures
  - Can only dispatch at most once
  - Can only dispatch task being enqueued
  - Cannot dispatch to remote CPU local DSQ

ops.dispatch()

- Implementation allows multiple tasks to be dispatched in sequence
- Can iterate over DSQ using bounded loop iterator, select which task you want
- Can dispatch to remote CPUs' LOCAL_DSQs

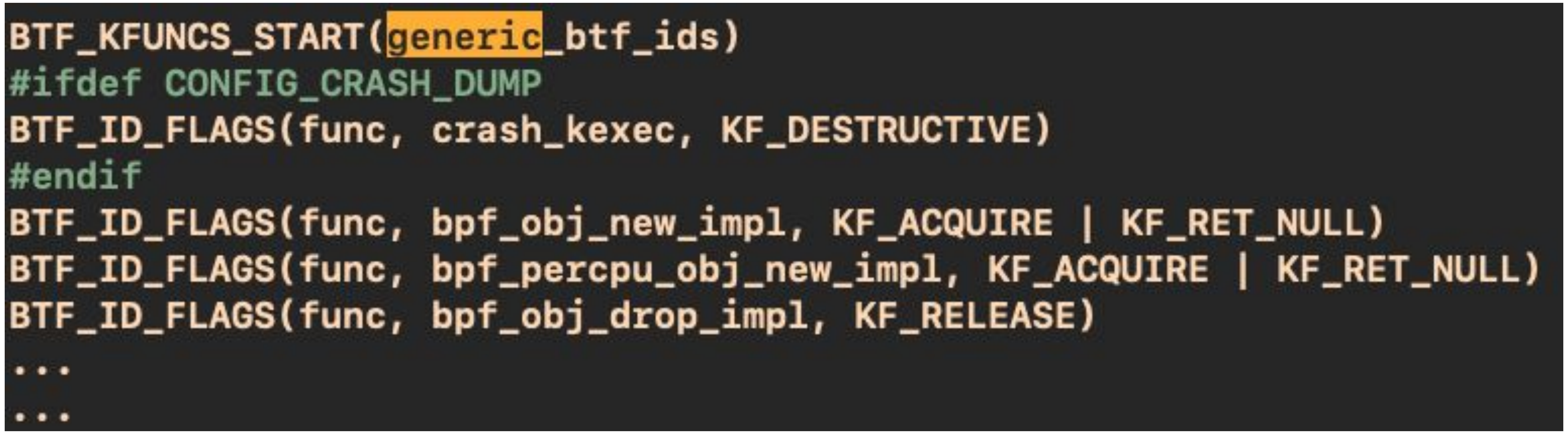
Result: Two completely different implementations, with same API

- Can we explicitly support this pattern in the BPF framework?

02 Design proposal

Currently, call BTF ID → specific kfunc

- In existing code, a BTF ID corresponds to exactly one kfunc
- libbpf does relocations, kernel sees BTF ID and patches in kfunc address



```
BTF_KFUNCS_START(generic_btf_ids)
#ifdef CONFIG_CRASH_DUMP
BTF_ID_FLAGS(func, crash_kexec, KF_DESTRUCTIVE)
#endif
BTF_ID_FLAGS(func, bpf_obj_new_impl, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_percpu_obj_new_impl, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_obj_drop_impl, KF_RELEASE)
...
...
```

Every kfunc associated with exactly 1 ID

- Problem: Every kfunc call is associated with exactly 1 BTF_ID
- Kfunc calls are static – specify BTF ID → patch kfunc

How to extend? Verifier asks subsystem for real kfunc ID

- Kfunc → kfunc mappings need to happen at prog granularity
  - `struct bpf_struct_ops` already has per-member callbacks, e.g. `init_member()`
- Must be located in the kernel (right?)
  - libbpf has no way of mapping kfunc calling context in a prog → actual kfunc symbol. Completely depends on the struct_ops implementation
- Can we add a new `.kfunc_validate_reloc()` function that lets the program map a kfunc ID passed by the verifier to the BTF ID of the kfunc they actually want to invoke?
  - Invoked for every kfunc call, for every struct_ops prog
  - Fixups happen in the kernel

```
s32 (*kfunc_validate_reloc)(const struct btf_type *t,
            const struct btf_member *member,
             struct bpf_prog *prog,
              u32 kfunc_id);
```

Proposed function signature

-
  ```
  s32 (*kfunc_validate_reloc)(const struct btf_type *t,
                               const struct btf_member *member,
                               struct bpf_prog *prog,
                               u32 kfunc_id);
  ```

- Return kfunc id of kfunc exported from struct_ops implementation, 0 if no relocation necessary, or negative error code for error

Pros

- A somewhat ergonomic API. Each kfunc handled separately, provides well-contained logic to implement on the struct_ops implementation side
- Gives struct_ops implementations a way to reject improper kfunc call at verify time instead of runtime

Cons

- Kind of a weird API to have both `.check_member()`, and another kfunc for doing validation
- More callback logic in the verifier. I know that's not always a popular design choice
- Requires runtime logic for what's really a static configuration
- Requires struct_ops implementation to do BTF resolution and track BTF IDs

Static / build-time configuration would be a nicer API

- Which kfuncs should be called from which contexts is not really dynamic
- Can we make this a build time thing?
- Would require associating struct_ops entries / progs with kfunc IDs that map to other kfunc IDs
- Probably a big pain to implement, but would end up being nicer for end users
    - Doesn’t seem like a good time investment until there are more struct_ops implementations
    - Bigger fish to fry – declaring kfuncs similar to `EXPORT_SYMBOL_GPL` would be more ideal

Meta