BPF_MAP_TYPE_CGROUP_STORAGE

The `BPF_MAP_TYPE_CGROUP_STORAGE` map type represents a local fix-sized storage. It is only available with `CONFIG_CGROUP_BPF`, and to programs that attach to cgroups; the programs are made available by the same Kconfig. The storage is identified by the cgroup the program is attached to.

The map provide a local storage at the cgroup that the BPF program is attached to. It provides a faster and simpler access than the general purpose hash table, which performs a hash table lookups, and requires user to track live cgroups on their own.

This document describes the usage and semantics of the `BPF_MAP_TYPE_CGROUP_STORAGE` map type. Some of its behaviors was changed in Linux 5.9 and this document will describe the differences.

Usage

The map uses key of type of either `__u64 cgroup_inode_id` or `struct bpf_cgroup_storage_key`, declared in `linux/bpf.h`:

```
struct bpf_cgroup_storage_key {
        __u64 cgroup_inode_id;
        __u32 attach_type;
};
```

`cgroup_inode_id` is the inode id of the cgroup directory. `attach_type` is the the program's attach type.

Linux 5.9 added support for type `__u64 cgroup_inode_id` as the key type. When this key type is used, then all attach types of the particular cgroup and map will share the same storage. Otherwise, if the type is `struct bpf_cgroup_storage_key`, then programs of different attach types be isolated and see different storages.

To access the storage in a program, use `bpf_get_local_storage`:

```
void *bpf_get_local_storage(void *map, u64 flags)
```

`flags` is reserved for future use and must be 0.

There is no implicit synchronization. Storages of `BPF_MAP_TYPE_CGROUP_STORAGE` can be accessed by multiple programs across different CPUs, and user should take care of synchronization by themselves. The bpf infrastructure provides `struct bpf_spin_lock` to synchronize the storage. See `tools/testing/selftests/bpf/progs/test_spin_lock.c`.

Examples

Usage with key type as `struct bpf_cgroup_storage_key`:

```
#include <bpf/bpf.h>

struct {
        __uint(type, BPF_MAP_TYPE_CGROUP_STORAGE);
        __type(key, struct bpf_cgroup_storage_key);
        __type(value, __u32);
} cgroup_storage SEC(".maps");

int program(struct __sk_buff *skb)
{
        __u32 *ptr = bpf_get_local_storage(&cgroup_storage, 0);
        __sync_fetch_and_add(ptr, 1);

        return 0;
}
```

Userspace accessing map declared above:

```
#include <linux/bpf.h>
#include <linux/libbpf.h>

__u32 map_lookup(struct bpf_map *map, __u64 cgrp, enum bpf_attach_type type)
{
        struct bpf_cgroup_storage_key = {
                .cgroup_inode_id = cgrp,
                .attach_type = type,
        };
        __u32 value;
        bpf_map_lookup_elem(bpf_map__fd(map), &key, &value);
        // error checking omitted
        return value;
}
```

Alternatively, using just `__u64 cgroup_inode_id` as key type:

```
#include <bpf/bpf.h>

struct {
        __uint(type, BPF_MAP_TYPE_CGROUP_STORAGE);
        __type(key, __u64);
        __type(value, __u32);
} cgroup_storage SEC(".maps");

int program(struct __sk_buff *skb)
{
        __u32 *ptr = bpf_get_local_storage(&cgroup_storage, 0);
        __sync_fetch_and_add(ptr, 1);

        return 0;
}
```

And userspace:

```
#include <linux/bpf.h>
#include <linux/libbpf.h>

__u32 map_lookup(struct bpf_map *map, __u64 cgrp, enum bpf_attach_type type)
{
        __u32 value;
        bpf_map_lookup_elem(bpf_map__fd(map), &cgrp, &value);
        // error checking omitted
        return value;
}
```

Semantics

`BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE` is a variant of this map type. This per-CPU variant will have different memory regions for each CPU for each storage. The non-per-CPU will have the same memory region for each storage.

Prior to Linux 5.9, the lifetime of a storage is precisely per-attachment, and for a single `CGROUP_STORAGE` map, there can be at most one program loaded that uses the map. A program may be attached to multiple cgroups or have multiple attach types, and each attach creates a fresh zeroed storage. The storage is freed upon detach.

There is a one-to-one association between the map of each type (per-CPU and non-per-CPU) and the BPF program during load verification time. As a result, each map can only be used by one BPF program and each BPF program can only use one storage map of each type. Because of map can only be used by one BPF program, sharing of this cgroup's storage with other BPF programs were impossible.

Since Linux 5.9, storage can be shared by multiple programs. When a program is attached to a cgroup, the kernel would create a new storage only if the map does not already contain an entry for the cgroup and attach type pair, or else the old storage is reused for the new attachment. If the map is attach type shared, then attach type is simply ignored during comparison. Storage is freed only when either the map or the cgroup attached to is being freed. Detaching will not directly free the storage, but it may cause the reference to the map to reach zero and indirectly freeing all storage in the map.

The map is not associated with any BPF program, thus making sharing possible. However, the BPF program can still only associate with one map of each type (per-CPU and non-per-CPU). A BPF program cannot use more than one `BPF_MAP_TYPE_CGROUP_STORAGE` or more than one `BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE`.

In all versions, userspace may use the the attach parameters of cgroup and attach type pair in `struct bpf_cgroup_storage_key` as the key to the BPF map APIs to read or update the storage for a given attachment. For Linux 5.9 attach type shared storages, only the first value in the struct, cgroup inode id, is used during comparison, so userspace may just specify a `__u64` directly.

The storage is bound at attach time. Even if the program is attached to parent and triggers in child, the storage still belongs to the parent.

Userspace cannot create a new entry in the map or delete an existing entry. Program test runs always use a temporary storage.