Next Previous Contents

13. Sysfs and kobjects

Sysfs is a virtual filesystem that describes the devices known to the system from various viewpoints. By default it is mounted on `/sys`. The basic building blocks of the hierarchy are kobjects. The entire setup is strange and messy and will cause lots of bugs. Let us start looking at the reference counting part (nothing wrong with that).

13.1 atomic_t

An atomic_t is an integer variable that can be inspected and changed atomically. It is mostly used for reference counting and semaphores. On i386 it is defined as

```
typedef struct { volatile int cnt; } atomic_t;
```

and available methods (with `int i` and `atomic_t *v`) are

```
atomic_read(v)                  v->cnt
atomic_set(v,i)                 v->cnt = i
atomic_add(i,v)                 v->cnt += i
atomic_sub(i,v)                 v->cnt -= i
atomic_sub_and_test(i,v)        (v->cnt -= i) == 0
atomic_inc(v)                   v->cnt++
atomic_dec(v)                   v->cnt--
atomic_dec_and_test(v)          --v->cnt == 0
atomic_inc_and_test(v)          ++v->cnt == 0
atomic_add_negative(i,v)        (v->cnt += i) < 0
```

Here the right hand sides define the intended effect, but the code must be written in assembler to guarantee atomicity. The definitions can be found in `<asm/atomic.h>`. E.g. for i386:

```
static __inline__ void atomic_dec(atomic_t *v) {
        __asm__ __volatile__(
                LOCK "decl %0"
                :"=m" (v->cnt)
                :"m" (v->cnt));
}
```

where the `LOCK` becomes a lock prefix `0xf0` on an SMP machine. For the i386 a simple aligned read is atomic:

```
#define atomic_read(v)          ((v)->cnt)
```

The contents used to be a 24-bit counter, because sparc used the low byte as a spinlock, e.g.,

```
static inline int atomic24_read(const atomic24_t *v) {
        int ret = v->cnt;
        while(ret & 0xff)
                ret = v->cnt;
        return ret >> 8;
}
```

but since Linux 2.6.3 also sparc has a 32-bit counter. (The reason the Sparc is troublesome is that its 32-bit memory writes are not atomic: it is possible for another CPU to come and read memory and find some old and some new bytes, a partly written result.)

13.2 struct kref

A `struct kref` is an object that handles reference counting. It is defined as

```
struct kref {
        atomic_t refcount;
};
```

and available methods are (with `struct kref *k` and `void (*release)(struct kref *k)`)

```
void kref_init(k)                k->refcount = 1;
void kref_get(k)                 k->refcount++;
void kref_put(k,release)         if (!--k->refcount) release(k);
```

Thus, today a `struct kref` is just an `atomic_t`. Still, it is a useful abstraction: it has far fewer methods, so reasoning about it is simpler.

Earlier the `struct kref` had a `release` field, but all uses turned out to be such that the caller knew the appropriate release function, so that it could be a parameter of `kref_put()`, saving memory.

Releasing just the `struct kref` would not be very meaningful. The typical use is shown by

```
struct scsi_disk {
        struct scsi_driver *driver;
        struct scsi_device *device;
        struct kref     kref;
        ...
};

#define to_scsi_disk(obj) container_of(obj,struct scsi_disk,kref)

static void scsi_disk_put(struct scsi_disk *sdkp) {
        down(&sd_ref_sem);
        ...
        kref_put(&sdkp->kref, scsi_disk_release);
        up(&sd_ref_sem);
}

static void scsi_disk_release(struct kref *kref)
{
        struct scsi_disk *sdkp = to_scsi_disk(kref);
        ...
        kfree(sdkp);
}
```

See also container_of.

13.3 struct kobject

A `struct kobject` represents a kernel object, maybe a device or so, such as the things that show up as directory in the `sysfs` filesystem.

It is defined as

```
#define KOBJ_NAME_LEN   20

struct kobject {
        char                    *k_name;
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct dentry           *dentry;
};
```

Every `struct kobject` has a name, which must not be NULL. The name is `kobj->k_name`, and this pointer points either to the internal array, if the name is short, or to an external string obtained from `kmalloc()` and to be `kfree()`d when the kobject dies. However, we are not supposed to know this - the name is returned by `kobject_name(kobj)` and set by `kobject_set_name(kobj, format, ...)`. (But that latter routine cannot fail in case the name we use has length less than 20.) The name is set independently of other initialization.

A `struct kobject` may be member of a set, given by the `kset` field. Otherwise, this field is NULL. The `kset` field must be set before calling `kobject_init()`.

The `entry` field is either empty or part of the circularly linked list containing members of the kset.

A `struct kobject` is reference counted. The routines `kobject_get()` and `kobject_put()` do get/put on the `kref` field. When the reference count drops to zero, a `kobject_cleanup()` is done.

A `struct kobject` must be initialized by `kobject_init()`. This does the `kref_init` that sets the refcount to 1, initializes the `entry` field to an empty circular list, and does

```
        kobj->kset = kset_get(kobj->kset);
```

which does not change `kobj->kset` but increments its refcount: one more element of the set. Note that most fields are not touched by `kobject_init()`. One should memset it to zero and possibly assign `kset` before calling `kobject_init()`.

One of the main purposes of a `struct kobject` is to appear in the sysfs tree. It is added in the tree by `kobject_add()` and deleted again by `kobject_del()`. The former calls `kobject_hotplug("add",kobj)`, the latter `kobject_hotplug("remove", kobj)`.

The sysfs dentry is given by the field `dentry`. The parent directory in the tree is represented by the kobject `parent`. The routine `sysfs_create_dir()` will hang a new directory directly below the root `/sys` when no parent is given.

Let us list the kobject methods (with `struct kobject *ko`)

```
int kobject_set_name(ko, char *, ...)
char *kobject_name(ko)
void kobject_init(ko)
struct kobject *kobject_get(ko)
void kobject_put(ko)
void kobject_cleanup(ko)
int kobject_add(ko)
void kobject_del(ko)
int kobject_register(ko)
void kobject_unregister(ko)
int kobject_rename(ko, char *new_name)
void kobject_hotplug(const char *action, ko)
char *kobject_get_path(struct kset *, ko, int)
```

Most of these were mentioned above. The routine `kobject_register()` does `kobject_init(); kobject_add()`, and `kobject_unregister()` does `kobject_del(); kobject_put()`. This is the preferred way of handling kobjects that are represented in the sysfs tree.

The rest will be discussed below.

13.4 struct kset

A `struct kset` represents a set of kernel objects. It is defined as

```
struct kset {
        struct subsystem        *subsys;
        struct kobj_type        *ktype;
        struct list_head        list;
        struct kobject          kobj;
        struct kset_hotplug_ops *hotplug_ops;
};
```

The field `list` provides a circularly linked list of the kobjects that are members of the kset. All kobjects on the list have a `kset` field that points back to us. In order to examine or manipulate the list, one needs to hold the `kset->subsys->rwsem` semaphore.

The set is also itself a kobject, given by the field `kobj`.

Methods are (with `struct kset *ks`)

```
void kset_init(ks)
struct kset *to_kset(ko)
struct kset *kset_get(ks)
void kset_put(ks)
int kset_add(ks)
int kset_register(ks)
void kset_unregister(ks)
struct kobject *kset_find_obj(ks, char *)
```

The routine `to_kset()` converts a pointer to the kobject field of a kset into a pointer to the kset itself.

The routine `kset_init()` initializes the fields `list` and `kobj`. The routine `kset_get()` does

```
        to_kset(kobject_get(&ks->kobj))
```

a rather complicated way of doing `kobject_get(&ks->kobj)` and returning `ks` again. The routine `kset_put()` does `kobject_put(&ks->kobj)`.

The routine `kset_add()` does a `kobject_add()` of its kobject. If that kobject did not yet have a parent or kset and we have a subsys, set the kobject's parent to the kobject of the kset of the subsys. On the other hand, if that kobject has a kset but no parent, then `kobject_add()` will set the parent to the kset itself.

Just like for kobjects, we have `kset_register()` that does `kset_init(); kset_add()`, and `kset_unregister()` that just does `kobject_unregister()`.

Finally, there is `kset_find_obj()` that given a name and a kset returns the kobject in the kset with that name.

13.5 struct kobj_type

Both a kobject and a kset have a field of type `struct kobj_type`. Such a struct represents a type of objects, and will hold the methods used to operate on them. The definition is

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops        * sysfs_ops;
        struct attribute        ** default_attrs;
};
```

Here the release function is what is called by `kobject_cleanup()`. It uses the ktype of the kset of the kobject, or, there is none, the ktype of the kobject itself - this is what is returned by the method `get_ktype()`.

Method:

```
struct kobj_type *get_ktype(ko)
```

The attributes describe the ordinary files in the sysfs tree. It is a NULL-terminated list of

```
struct attribute {
        char                    *name;
        struct module           *owner;
        mode_t                  mode;
};
```

The contents of these files is generated by `show()` and can possibly be modified by the `store()` function:

```
struct sysfs_ops {
        ssize_t (*show)(kobj, struct attribute *attr, char *buf);
        ssize_t (*store)(kobj, struct attribute *attr, const char *, size_t);
};
```

13.6 struct subsystem

Defined by

```
struct subsystem {
        struct kset             kset;
        struct rw_semaphore     rwsem;
};
```

A kset belongs to a subsystem, and the rwsem of the subsystem is used to protect its list.

Subsystems tend to correspond to toplevel directories in the sysfs hierarchy. Their names in the source tend to end in `_subsys` (produced by the macro `decl_subsys()`). I see

```
% ls /sys
block  bus  class  devices  firmware  module
%
```

and find in the source `system_subsys`, `block_subsys`, `bus_subsys`, `class_subsys`, `devices_subsys`, `firmware_subsys`, `class_obj_subsys`, `acpi_subsys`, `edd_subsys`, `vars_subsys`, `efi_subsys`, `cdev_subsys`, `module_subsys`, `power_subsys`, `pci_hotplug_slots_subsys`.

Here `edd` lives below `firmware`, and `system` below `devices`.

Some are not visible in sysfs. We'll look at some of these below.

13.7 struct kobj_map

Device number handling is done using the `struct kobj_map`. See `drivers/base/map.c`.

```
typedef struct kobject *kobj_probe_t(dev_t, int *, void *);

struct kobj_map {
        struct probe {
                struct probe *next;
                dev_t dev;
                unsigned long range;
                struct module *owner;
                kobj_probe_t *get;
                int (*lock)(dev_t, void *);
                void *data;
        } *probes[255];
        struct rw_semaphore *sem;
};
```

with methods

```
int kobj_map(struct kobj_map *domain, dev_t dev, unsigned long range,
             struct module *owner, kobj_probe_t *get,
             int (*lock)(dev_t, void *), void *data);
void kobj_unmap(struct kobj_map *domain, dev_t dev, unsigned long range);
struct kobject *kobj_lookup(struct kobj_map *domain, dev_t dev, int *index);
struct kobj_map *kobj_map_init(kobj_probe_t *base_probe, struct subsystem *s);
```

Each `struct probe` describes a device number interval starting at `dev` and with length `range`. With lots of devices one would need some fast data structure to find stuff, but for the time being the current version suffices. It uses the major device number mod 255 as index in the array, where each entry is head in a linked list of such structs. Handling the linked lists is protected by the semaphore `*sem`, set to the subsystem semaphore at init time.

The call `kobj_lookup()` finds the given device number `dev` on the given map `domain` (there are two: device numbers for block devices and device numbers for character devices). If the `lock` function is present it will be called, and the present probe skipped if it returns an error. Then the `get` function is called to get the kobject for the given device number. If the `owner` field is set, we take a reference on the corresponding module via `try_module_get(owner)` in order to protect the `lock` and `get` calls. The resulting kobject is returned as value, and the offset in the interval of device numbers is returned via `index`.

It is possible to have several nested intervals of device numbers, and the lists are sorted such that smaller intervals come first in the linked lists, so that the most specific entry overrides other entries. The final entry is the interval `[1,~0]` covering all device numbers.

The call `kobj_map()` adds an entry with the given data. The call `kobj_unmap()` removes it again. The call `kobj_map_init()` initializes the map. The subsystem argument provides the semaphore. The `base_probe()` parameter is the `get` function for the interval covering all device numbers.

There are two calls of `kobj_map_init()`, namely

```
        bdev_map = kobj_map_init(bbase_probe, &block_subsys);
```

and

```
        cdev_map = kobj_map_init(cbase_probe, &cdev_subsys);
```

Here the `base_probe` functions are

```
struct kobject *bbase_probe(dev_t dev, int *part, void *data) {
        if (request_module("block-major-%d-%d", MAJOR(dev), MINOR(dev)) > 0)
                request_module("block-major-%d", MAJOR(dev));
        return NULL;
}
```

and

```
struct kobject *cbase_probe(dev_t dev, int *part, void *data) {
        if (request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev)) > 0)
                request_module("char-major-%d", MAJOR(dev));
        return NULL;
}
```

The existence of `bdev_map` is hidden by the helper functions `blk_register_region()`, `blk_unregister_region()`, `get_gendisk()` that do a `kobj_map()`, `kobj_unmap()` and `kobj_lookup()`, respectively.

13.8 Example: floppy

For example, the only part of `floppy.c` that refers to kobjects is the routine

```
struct kobject *floppy_find(dev_t dev, int *part, void *data) {
        int drive = (*part & 3) | ((*part & 0x80) >> 5);
        if (something wrong)
                return NULL;
        *part = 0;
        return get_disk(disks[drive]);
}
```

that is made the probe function via

```
        blk_register_region(MKDEV(FLOPPY_MAJOR, 0), 256, THIS_MODULE,
                            floppy_find, NULL, NULL);
```

Here `get_disk()` is responsible for doing `try_module_get()` and `kobject_get()`, and the kobject involved is the one embedded in the struct gendisk.

In `floppy_init()` we see

```
        disks[dr] = alloc_disk(1);
        ...
        sprintf(disks[dr]->disk_name, "fd%d", dr);
        ....
        add_disk(disks[dr]);
```

Here `disks[]` is an array of `struct gendisks`, one for each device. The `alloc_disk()` routine allocates one and initializes some things. In particular, it sets the kset of the embedded kobject to `block_subsys.kset`. Next,

```
void add_disk(struct gendisk *disk) {
        disk->flags |= GENHD_FL_UP;
        blk_register_region(MKDEV(disk->major, disk->first_minor),
                            disk->minors, NULL, exact_match, exact_lock, disk);
        register_disk(disk);
        blk_register_queue(disk);
}
```

Here `exact_lock()` takes a reference on `disk`, and `exact_match` returns the kobject embedded in `disk`. See how the `data` field is used here to store the `disk` pointer.

The `register_disk()` copies the disk name `disk->disk_name` (set above to e.g. `"fd0"`) to `disk->kobj.name`, replacing embedded slashes by exclamation marks, so that the name is suitable as a file name. Then `kobject_add()` puts the thing in the sysfs tree. Where? Below the subsys, that is, as `/sys/block/fd0`, because that is where `alloc_disk()` pointed us.

The `blk_register_queue` adds a subdirectory `queue` below `/sys/block/fd0`.

13.9 Hotplug

One of the fields of a kset is a `struct kset_hotplug_ops` defined by (in abbreviated notation)

```
struct kset_hotplug_ops {
        int (*filter)(kset, kobj);
```

```
        char *(*name)(kset, kobj);
        int (*hotplug)(kset, kobj, char **envp, int num_envp, char *buf, int bufsz);
};
```

The routine `kobject_hotplug()` is called by `kobject_add()` when the kobject is added to the hierarchy. It needs a kset in order to call one of the kset_hotplug_ops functions, and to this end walks up the hierarchy along the parent pointers until a kobj is found with a non-NULL kset. Then it calls `kset_hotplug(action, kset, orig_kobj)`.

Now `kset_hotplug()` first calls the filter function if there is one, and does not do anything if that returns false. It doesn't do anything either when `hotplug_path` is the empty string.

(You can set `hotplug_path` by echoing to `/proc/sys/kernel/hotplug`. The default value is `"/sbin/hotplug"`.)

Now the executable with pathname found in `hotplug_path` is called with a single parameter, found by calling the `name()` function, and if that doesnt yield a name, taking the name of the (kobject of the) kset. The process gets an environment with `HOME=/`, `PATH=/sbin:/bin:/usr/sbin:/usr/bin`, `ACTION=...`, `SEQNUM=...`, `DEVPATH=...`, where the path given by `kobject_get_path()` is the path in the sysfs hierarchy to the kobj. Finally, the `hotplug()` function may add some more environment parameters. Then the executable is launched.

Next Previous Contents