The Resource Counter

The resource counter, declared at include/linux/res_counter.h,
is supposed to facilitate the resource management by controllers
by providing common stuff for accounting.

This "stuff" includes the res_counter structure and routines
to work with it.

1. Crucial parts of the res_counter structure

a. unsigned long long usage

The usage value shows the amount of a resource that is consumed by a group at a given time. The units of measurement should be determined by the controller that uses this counter. E.g. it can be bytes, items or any other unit the controller operates on.

b. unsigned long long max_usage

The maximal value of the usage over time.

This value is useful when gathering statistical information about the particular group, as it shows the actual resource requirements for a particular group, not just some usage snapshot.

c. unsigned long long limit

The maximal allowed amount of resource to consume by the group. In case the group requests for more resources, so that the usage value would exceed the limit, the resource allocation is rejected (see the next section).

d. unsigned long long failcnt

The failcnt stands for "failures counter". This is the number of resource allocation attempts that failed.

c. spinlock_t lock

Protects changes of the above values.

2. Basic accounting routines

a. void res_counter_init(struct res_counter *rc,
struct res_counter *rc_parent)

Initializes the resource counter. As usual, should be the first routine called for a new counter.

The struct res_counter *parent can be used to define a hierarchical

child -> parent relationship directly in the res_counter structure, NULL can be used to define no relationship.

c. int res_counter_charge(struct res_counter *rc, unsigned long val,
struct res_counter **limit_fail_at)

When a resource is about to be allocated it has to be accounted with the appropriate resource counter (controller should determine which one to use on its own). This operation is called "charging".

This is not very important which operation - resource allocation or charging - is performed first, but
* if the allocation is performed first, this may create a temporary resource over-usage by the time resource counter is charged;
* if the charging is performed first, then it should be uncharged on error path (if the one is called).

If the charging fails and a hierarchical dependency exists, the limit_fail_at parameter is set to the particular res_counter element where the charging failed.

d. int res_counter_charge_locked
(struct res_counter *rc, unsigned long val)

The same as res_counter_charge(), but it must not acquire/release the res_counter->lock internally (it must be called with res_counter->lock held).

e. void res_counter_uncharge[_locked]
(struct res_counter *rc, unsigned long val)

When a resource is released (freed) it should be de-accounted from the resource counter it was accounted to. This is called "uncharging".

The _locked routines imply that the res_counter->lock is taken.

2.1 Other accounting routines

There are more routines that may help you with common needs, like checking whether the limit is reached or resetting the max_usage value. They are all declared in include/linux/res_counter.h.

3. Analyzing the resource counter registrations

a. If the failcnt value constantly grows, this means that the counter's limit is too tight. Either the group is misbehaving and consumes too many resources, or the configuration is not suitable for the group and the limit should be increased.

b. The max_usage value can be used to quickly tune the group. One may set the limits to maximal values and either load the container with a common pattern or leave one for a while. After this the max_usage

value shows the amount of memory the container would require during its common activity.

Setting the limit a bit above this value gives a pretty good configuration that works in most of the cases.

c. If the max_usage is much less than the limit, but the failcnt value is growing, then the group tries to allocate a big chunk of resource at once.

d. If the max_usage is much less than the limit, but the failcnt value is 0, then this group is given too high limit, that it does not require. It is better to lower the limit a bit leaving more resource for other groups.

4. Communication with the control groups subsystem (cgroups)

All the resource controllers that are using cgroups and resource counters should provide files (in the cgroup filesystem) to work with the resource counter fields. They are recommended to adhere to the following rules:

a. File names

Field name	File name
usage	usage_in_<unit_of_measurement>
max_usage	max_usage_in_<unit_of_measurement>
limit	limit_in_<unit_of_measurement>
failcnt	failcnt
lock	no file :)

b. Reading from file should show the corresponding field value in the appropriate format.

c. Writing to file

Field	Expected behavior
usage	prohibited
max_usage	reset to usage
limit	set the limit
failcnt	reset to zero

5. Usage example

a. Declare a task group (take a look at cgroups subsystem for this) and fold a res_counter into it

```
struct my_group {
        struct res_counter res;

        <other fields>
```

```
}
```

b. Put hooks in resource allocation/release paths

```
int alloc_something(...)
{
        if (res_counter_charge(res_counter_ptr, amount) < 0)
                return -ENOMEM;

        <allocate the resource and return to the caller>
}

void release_something(...)
{
        res_counter_uncharge(res_counter_ptr, amount);

        <release the resource>
}
```

In order to keep the usage value self-consistent, both the "res_counter_ptr" and the "amount" in release_something() should be the same as they were in the alloc_something() when the releasing resource was allocated.

c. Provide the way to read res_counter values and set them (the cgroups still can help with it).

c. Compile and run :)