

[RFC] Revised CKRM API

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-01/7824.html>

From: Shailabh Nagar (nagar_at_watson.ibm.com)

Date: 01/31/04

Date: Fri, 30 Jan 2004 20:51:05 -0500

To: Rik van Riel <riel@redhat.com>

Rik van Riel wrote:

>On Wed, 28 Jan 2004, Shailabh Nagar wrote:
>
>
>>Incidentally, the website ckrm.sf.net has been updated a bit to make the
>>second version of the API/core (C01) more visible.
>>
>
>Cool.
>
>OTOH, Stephen Tweedie just convinced me to also look at another
>model of doing resource management configuration ;)
>
>Basically resource groups would be directories in a special
>filesystem. Users would be to create new hierarchical resource
>groups in directories they have write access to and call a
>system call to change into a new resource group, provided
>they have execute permission on that directory.
>
>Shell scripts can easily change themselves into a new resource
>group, while applications linked to the PAM library can be moved
>into their resource group using the pam library.
>
>I am going to look into this in more detail before I can decide
>whether or not it's completely feasible, but the fact that the
>user can subdivide their own resources easily is definately a
>big plus over the "root preconfigures all resources" model...
>
>I'll let you know where it goes. The resource usage enforcement
>will be the same in both models anyway.
>

Hi Rik,

Here's a writeup of a revised design for CKRM that we've been thinking

Linux–Kernel: [RFC] Revised CKRM API

about. It takes the state that's on the CKRM website a bit further by integrating the inbound network control (we'd described the latter at OLS'03 but not completely integrated with CKRM) and putting some more thought into the API.

As Hubertus mentioned, your ideas on hierarchical classes and a filesystem–based interface sound useful. Please review the document to see how we could come up with a uniform API and design for class–based resource management.

CKRM is not wedded to any API. We're only interested in trying to evolve a solution that

- preserves independence of the classification mechanism and the resource controllers
- integrates/allows inbound network control. This presents a bit of a challenge as inbound network classes do not fit neatly into the task group concept but it does fit in with what a user of CKRM would like to do (control inbound network connections/bandwidth by some user–defined grouping)
- implements mechanisms and leaves policy to users: This sounds like a mantra from an OS textbook but it's really important here so that resource management middleware (besides sysadmins) can make use of CKRM APIs effectively.
- causes minimal (zero would be nice) performance impact to users who don't care for such control

I'm pretty sure those are all desirable attributes for your design too so we should be able to come up with something common. As you mentioned, the control mechanisms won't change too much if the semantics of hierarchical classes are kept simple.

Looking forward to your writeup and comments.

-- Shailabh

CKRM (Class–based Kernel Resource Management)
<http://ckrm.sf.net>

v1.0

30 Jan 2004, 19:52 EST

Hubertus Franke, frankeh@watson.ibm.com
Shailabh Nagar nagar@watson.ibm.com
Vivek Kashyap vivk@us.ibm.com

Chandra Seetharaman sekharan@us.ibm.com

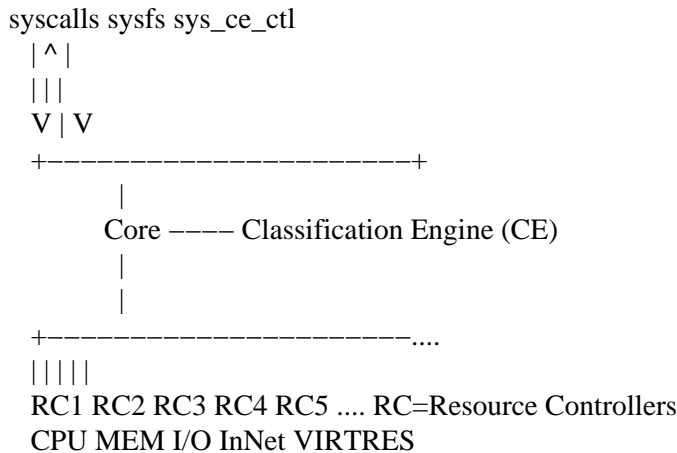
This is the third major revision of the CKRM API and framework. The first was presented at OLS'03 and the second is what's described at ckrm.sf.net as of 30 Jan 2004. The project is not committed to a particular API or architecture and welcomes discussions/comments on the proposal below. Please send feedback to any of the authors and cc: ckrm-tech@lists.sf.net.

1.0 Overview

=====

CKRM defines an infrastructure wherein the various components of class based resource management as well as their interactions are defined. This document outlines the basic notations, components and the relevant APIs (syscalls and intercomponent). Following is a schematic overview of the components involved.

Sysadmin/resource management application



The following entities are central to the proposed CKRM.

1.1 Core

Kernel patch that has three key roles. First, it defines the user API consisting of system calls and some sysfs entries. Second, it defines the APIs between itself and all other kernel components namely the resource controllers and the optional classification engine. Thus the Core acts as the switchboard for users, resource controllers and classification engines to interact. Finally, Core handles the creation and management of task classes, which are described below.

1.2 Resource controller (RC)

A kernel patch providing differentiated access to some resource. There can be multiple RCs defined simultaneously. CKRM currently provides CPU (ticks), Mem

(physical page frames), I/O (disk bandwidth) and Inbound Network (connections) controllers with extensions planned to manage virtual resources like open file descriptors, shared memory segments etc.

1.2.1 Hierarchy of resources and resource controllers

Resource controllers for traditional resources such as CPU and MEM will typically manage only one resource (cpu ticks, page frames respectively). However CKRM permits a single RC to manage a hierarchy of resources e.g. a VIRTRES resource controller can be defined which manages sub resources e.g. open file descriptors, shared memory segments etc.

The resource hierarchy can also be viewed as a resource controller hierarchy with resource controllers within the same branch sharing code, data structures etc. However, the CKRM Core only explicitly recognizes the first level of the RC hierarchy, henceforth called primary resource controllers (PRC). Each PRC registers with the Core and is thereafter responsible for handling all requests for the resource hierarchy it supports. PRCs are completely independent of each other and can be developed separately (there might be performance dependencies between the resources controlled by PRCs e.g. cpu share settings affects memory page frame usage too, but identifying and handling these dependencies is the user's responsibility while setting shares).

1.2.2 Resource type (restype)

Each resource in the hierarchy is uniquely defined by an integer valued resource type (`res_type`) which encodes the path to the resource (and its controller) in the hierarchy. The following tree shows an RC hierarchy with the `res_type` values in hexadecimal.

- (0x1) CPU
- (0x2) MEM
- (0x3) IO
- (0x4) InNET
- (0x104) AcceptQ
- (0x5) VIRTRES
- (0x105) filedes
- (0x205) shmseg

At each level of the RC/resource hierarchy, the restype is defined as a bitmap bit–shifted into the restype of its parent. The number of bits to shift for a particular level is defined by the RC at that level and is available through `sysfs` (see below). In the example above, the first 8 bits are used by the first level.

1.2.3 Resource class (resclass)

An unnamed object created by the user and owned by a resource controller at any level of the resource hierarchy. Resclasses for leaves in the RC hierarchy always have an associated share e.g. each resclass defined by the CPU, MEM and IO controllers has an associated share of ticks, page frames and disk I/O

bandwidth. Resclasses for non–leaf nodes are containers for the set of leaf resclasses in that branch e.g. a resclass defined by VIRTRES is a container for two resclasses, one each defined by 'filedes' and 'shmseg'. Each of the leaf resclasses will have a share of corresponding to max number of open files, shm segments respectively and the VIRTRES resclass effectively represents a unique set of those share values for filedes and shmsegs.

Henceforth, resclasses for the primary resource controllers will be called primary resclasses.

1.3 Task class (tskclass):

An unnamed object associating a set of tasks with a set of primary resource classes. To the user, this is the "class" in CKRM and would be defined to correspond to a distinct workload.

Tasks belonging to a tskclass implicitly belong to the set of resclasses in the tskclass and are regulated by the corresponding RCs according to the shares of the resclasses.

There is a systemwide default tskclass containing the default primary resclasses for all registered RCs.

RCs can define resclasses that cannot be part of tskclasses e.g. the inbound network controller uses such resclasses to manage a resource that is defined within a socket's context. A user can still create such resclasses, define their shares and monitor their usage using the same APIs used for resclasses that do participate in tskclasses.

Each PRC has to provide a default resclass. When a tskclass is created and does not specify the resclass for any primary res_type, it is assumed to contain the default resclass for that res_type.

Tskclasses have two states: active and inactive. Active is the normal state. Inactive tskclasses get ignored during classification (see below). Deactivating a tskclass moves its tasks to the default tskclass.

1.4. Classification engine (CE):

A module that performs two main functions. First, it classifies tasks into task classes. Typically a CE will have a set of rules that are evaluated in some order to yield the tskclass. Each rule contains attribute–value pairs with attributes from the task_struct (or other information available from a task context). If each value of a rule term matches the corresponding attribute of a task, the task is classified into the rule's target tskclass.

Rules which have inactive tskclasses as their targets are ignored.

The second functionality of a CE is to provide task event monitoring information to the user. The CE defines a table of callback functions that are invoked at significant system events (such as fork, exec, etc). Typically these

events can potentially cause a task to be reclassified.

CE's are optional. Tasks in the system can also be manually associated with `tskclasses`. If a CE exists, it must adhere to the Core–CE kernel API.

In addition to rules, most CE's would have a container for a set of rules called a policy. Policies can be created and rules added to them individually. When all required rules are added, a policy is committed. Correspondingly policy decommit and delete operations are also typically needed. Commands defining functions for creating, deleting, policies and rules and committing, ecommitting policies are specified in the User–CE API (implemented through an `ioctl`–like `syscall`).

1.5 Sample system

An example system containing the above components could look as follows:

CPU, Mem, InNet RCs are enabled on a system. Each defines two resclasses each (gold and silver) and has one mandatory default resclass. The shares of the classes are also listed

CPU : `cpudflt(10)`, `cpugold(50)`, `cpusilver(20)`

Hence tasks in `cpudflt` get $10/(10+50+20) = 12.5\%$ of CPU ticks systemwide.

Mem: `memdflt(20)`, `memgold(60)`, `memsilver(40)`

`memsilver` gets $40/(20+60+40) = 33.3\%$ of physical page frames systemwide.

AcceptQ:

`netdflt–gold(50)`, `netdflt–silver(30)`, `netdflt–other(10)`
`netdflt1–gold(70)`, `netdflt1–silver(50)`, `netdflt1–other(25)`
`netgold–sockA(40)`, `netsilver–sockA(20)`
`netgold–sockB(60)`, `netsilver–sockB(20)`

`tskclasses` could be defined as follows:

`tskclsA` : {`cpugold`, `memgold`} containing all tasks with `uid==500`
`tskclsB` : {`cpugold`, `memsilver`} containing all tasks with `cmd=="bash"`
and `uid!=500`
`tskclsC` : {`cpudflt`, `memsilver`} containing all tasks with `cmd=="gcc"`
and `uid!=500`
`tskclsD` : {`cpugold`, `netdflt1–gold`, `netdflt1–silver`}
`tskclsE` : {`cpugold`, `netdflt2–silver`}

Note that `tskclsB` & `C` share the same Mem resclass (so all their tasks collectively get 33.3% of physical memory) ;

`tskclsA` & `B` share the same CPU resclass and `tskclsC` doesn't care about its CPU share (and has chosen to go with the default CPU resclass)

`netgold–sock*` and `netsilver–sock*` do not form part of any `tskclass` but are the targets of `iptables` rules that classify incoming connections to a listening

socket into these classes. Note that these resclasses are per–socket. For socket A (owned by some pid), the connections classified as netgold will be preferentially accepted over those classified as netsilver in the ratio 2:1. For socket B, this ratio is 3:1.

When a task is classified into a tskclass that includes a netdflt* resclass, the sockets created by the task will be initialized from the netdflt* class shares.

1.6 Control flow

The following sequence illustrates the control flow while using CKRM

- Core is active as soon as the kernel is booted.
- resource controller 1 registers
- resource controller 2 registers
- |
- |
- No task is classified, resource controllers handle tasks in default mode
- |
- |
- User defines multiple resource classes for resource type 1, 2, and so on
- User sets shares for each of the resource classes defined
- User defines task classes comprising of different resource classes
- User defines the state of the tasks classes
- |
- |
- Classification engine registers
- |
- |
- User defines rules and policies and to associate tasks with active task classes
- User may call reclassify to classify all tasks according the classification engine
- Tasks are allocated resources based on the resource class shares of the task class they belong to
- User gets resource usage of different resource classes
- |
- |

Having specified the general concepts we now specify the various API that govern the interaction between the following components:

- (a) user/system management
- (b) CKRM core
- (c) Classification Engine
- (d) resource controllers
- (e) resource presentation / system information

2.0 User–Core API

=====

In the following we provide the definition and prototypes of the sys calls to the CKRM Core. The other set of syscalls is part of the User–CE API. Unless otherwise mentioned, in order to execute these syscalls, the caller requires CAP_CKRM_AUTHORITY.

2.1 Resource classes

=====

```
a) int sys_rescls_create (int res_type,
void *restype_priv );
```

Creates a new resclass for res_type resource.

restype_priv is additional information that the res_type RC might need to create the resclass. For primary restypes that are also leaf nodes in the RC hierarchy e.g. CPU, Mem, I/O this parameter would be null.

For AcceptQ, something like

```
void * ==> struct {
    pid_t pid;
    struct sockaddr;
}
```

would be needed since the resclass being created is defined per socket.

Restypes like InNet use the same interface to create resclasses that cannot belong to any tskclass. Passing a null restype_priv to such restypes will indicate that such a resclass is being created.

The function returns resclsid, a handle to the created resclass (–1 on failure e.g. res_type not handled by RC).

```
b) int sys_rescls_del (int resclsid)
```

Delete resclass with given handle.

```
c) int sys_rescls_setshare (int res_type, int count,
    struct ckrm_share_value values[/*count*/] );
    int sys_rescls_getshare( int res_type, int count,
    struct ckrm_share_value values[/*count*/] );
```

```
struct ckrm_share_value {
    int resclsid;
    int shares; // shares to set
};
```

Set/get shares for a group of resclasses managed by RC res_type. Returns 0 (success), –1 if any of the resclass shares could not be set.

```
d) int sys_rescls_getid(int res_type, int tskcls_id, void *restype_priv);
```

Returns the id of a resclass for res_type resource within the context of a tskcls_id.

restype_priv parameter is the same as described in sys_rescls_create. For res_type == CPU, Mem, I/O, this would be null. In case the res_type identifies an RC that does not join task classes the restype_priv parameter must be specified and used to obtain the id.

2.2 Task classes

=====

```
a) int sys_tskcls_add (int tskcls_id, int count,
                    int rescls_ids[/* count */])
   int sys_tskcls_del (int tskcls_id, int count,
                    int rescls_ids[/* count */])
```

Overloaded function to add/delete resource classes from a tskclass and create/delete tskclasses.

tskcls_id !=0, count !=0
add/delete resclasses in rescls_ids from existing tskcls tskcls_id

tskcls_id !=0, count ==0
sys_tskcls_add does nothing.
sys_tskcls_del deletes given tskcls.

tskcls_id == 0
sys_tskcls_add creates new tskcls with resclasses in rescls_ids
and returns tskclass_id.
sys_tskcls_del does nothing.

Resource classes do not get deleted even if no tskclass contains them. Library functions can be created to distinguish resclass addition/deletion and tskclass creation/deletion.

Only primary resclasses can be added to a tskclass. Resclasses are added in sequential order and can hence overwrite previous settings.

Returns 0 on success, -1 on failure e.g. if one of the specified resclasses has its join_flag set to 0, it cannot be added to any tskclass.

```
b) int sys_tskcls_modify ( int tskcls_id, int state )
```

Change state (0=inactive, 1=active) of a tskclass.
Tasks of a deactivated tskclasses are reclassified to the default tskclass.

```
c) int sys_rescls_getusage ( int res_type, int reset,
int numres, struct ckrm_rusage[/*numres*/]);
```

```
struct ckrm_rusage {
int resclsid;
```

```

unsigned long long usage;
unsigned long long wait;
}

```

Get the usage information for resclasses of a res_type. rusage is an input/output parameter. Input is with resclsid filled in and output contains the corresponding average usage and wait times.

2.3 Generic

```

=====

```

a) int sys_tsk_reclassify (int pid, int newtskcls_id)

Change tskclass of task <pid> to newtskcls_id

pid==–1 reclassifies all tasks in the system. This is typically done after changing CE rule(s), loading a new CE etc.

pid==0 reclassifies the calling task.

newtskcls_id==–1 will use the CE to get the task's new tskclass.

Return new tskcls_id (success), –1 (failure)

b) int sys_tsk_settag (int pid, int len, void *tag)

int sys_tsk_gettag (int pid, int len, void *tag)

Set/get the tag field of <pid>'s task_struct

pid==0 identifies the calling task.

The caller must have CAP_CKRM_AUTHORITY to change any tag including its own. Typically, capability will be granted only to a trusted user level process.

2.4 sysfs

```

=====

```

The res_type hierarchy is exported through sysfs under /sys/ckrm.

Leaf nodes (RC/resources) have the following entries defined:

info:

some generic description of the resource controller

type:

relative offset of type within next level on controller.

units:

measures what (e.g. num–core–pages, time in ticks, accepts..)

shares–range:

what are legal ranges to set shares

supports–wait:

does this resource collect wait information

mode–supported:

does this resource support control or only monitoring
 0 = monitor only 1= manage and monitoring

mode:

returns current mode (monitor/manage) on read
 If mode–supported==1, writing to this file sets the mode
 (0=monitor only, 1=manage and monitor)

Non–Leaf nodes have the following entry formats

info: see above

bits: number of bits reserved for this level of restypes.

Here is an example tree..

```
/sys/.../ckrm/
/sys/.../ckrm/CPU/
/sys/.../ckrm/NET
/sys/.../ckrm/NET/acceptQ
/sys/.../ckrm/VirtRes
/sys/.../ckrm/VirtRes/filedes
/sys/.../ckrm/VirtRes/shmsegs
```

3.0 User–CE API

=====

The User–CE API is conceptual in that the sole sys call's stub is implemented by the Core patch but the real functionality is available through functions defined by the CE.

a) int sys_ckrm_ce_ctl(int cmd, void *arg)

system call to create and manipulate policies/rule sets There is a well defined set of commands with associated structured arguments as described in the following:

cmd | arg | returns

-----+-----+-----

```
CKRM_CE_CREATE_POLICY: int policy_id;
CKRM_CE_DELETE_POLICY: int policy_id;
CKRM_CE_COMMIT_POLICY: int policy_id;
CKRM_CE_DECOMMIT_POLICY: int policy_id;
/* stop classifying through this policy */
```

CKRM_CE_CURRENT_POLICY: void ; returns id;

```
CKRM_CE_CREATE_RULE:
CKRM_CE_CHANGE_RULE:
```

```

struct rbce_rule {
    int rule_id;
    int policy_id;
    int target_clsId;
    int append;
    int numruleterms;
    struct rbce_rule_term ruleterms[ varlen ];
}

```

```

struct rbce_rule_term {
    int cmd;
    union {
        char *cmdstr;
        char *tag;
        long id;
        int deprule;
        char *user_defined;
    }
}

```

cmds have a predefined set currently to match
EXECPATH, CMD, uid, gid, euid, egid, tag.

CKRM_CE_DELETE_RULE: int ruleid;

|

The above command set is expected to be useful to most classification engines. However, they are all optional and a CE could return -ENOSYS or similar. CE's can also extend the above command set by defining their own operations.

4.0 Core-CE API

=====

A CE registers with the Core by providing a function call table (which also serves as a handle for the CE during deregistration).

```

int ckrm_register_engine(ckrm_eng_callback_t *cb);
int ckrm_deregister_engine(ckrm_eng_callback_t *cb);

```

The callbacks are used to notify the CE of significant task transition events that might affect future classification of tasks or are simply of interest in to a CE that is monitoring said events. The few callbacks are created at their respective places in the kernel using a generic kernel hooking mechanism such as GKHI.

```

struct ckrm_eng_callback {

    /* task transition events */
    struct task_class* (*fork_cb) (task_t*);
    struct task_class* (*exec_cb) (task_t*, const char *path_to_file);
}

```

```

struct task_class* (*exit_cb) (task_t*);
struct task_class* (*uid_cb) (task_t*);
struct task_class* (*gid_cb) (task_t*);
struct task_class* (*app_tag_cb) (task_t*);

/* classification */
struct task_class* (*reclassify_cb) (task_t*);

/* task class changed outside CE through sys_ckrm_reclassify */
void (*manual_classified_cb) (task_t*);

/* task class notification */
(*add_tskcls_cb)(int tskcls_id, struct task_class*);
(*del_tskcls_cb)(int tskcls_id);
(*setmode_tskcls_cb)(int tskcls_id, int state);

/* sys_ce_ctl call back */
/* ioctl for all other CE specific functionality */
(*ce_ctl) (ce_op_t op, void *data)

};

```

5.0 Core–ResCtrlr API

```
=====
```

```

a) int ckrm_register_res_ctrlr(int res_type,
    ckrm_res_callback_t *res_cb)
    int ckrm_deregister_res_ctrlr(int res_type)

```

Register/deregister RC res_type with Core. res_cb is a table of callbacks and utility functions to manage resclasses.

```

b) void *ckrm_get_res (int res_type, struct task_struct *tsk)

```

Return the resclass handles from tsk's current tskclass for resource res_type. If a resclass of the specified res_type had not been included in tskclass, the res_type's default resclass handle will be returned.

The callbacks defined in res_cb are functions to be called by the CKRM core to manage the RC's resource class objects such as creation, deletion, share setting/getting and usage information

```

i) int (*rescls_alloc)(int restype)

```

Create resclass of given type with a default share defined by the RC (and modifiable through sysfs).

Returns handle to resclass.

```

ii) void (*rescls_free)(int restype, int rescls_id)

```

Linux-Kernel: [RFC] Revised CKRM API

Free the resclass rescls_id.

iii) void (*set_resshare)(int restype, int rescls_id, int shares)
int (*get_resshare)(int restype, int rescls_id)

set/get relative share of resclass

iv) void (*get_resusage)(int restype, int rescls_id, int reset,
int numres, struct ckrm_usage *res)

Return usage/wait times of resclass.

v) void (*change_tskcls)(int rescls_id, struct task_struct *tsk)

A class has changed its tskcls, hence all the RC need to be notified that the tsk is now assigned to the primary rescls_id under this RC.

--- End ---

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>