一个资源管理系统的设计--解析linux的cgroup实现

将实体打散成不可再分的微粒，这样就可以使设计灵活化，最大限度的减少数据冗余。以CRM系统为例，虽然管理是基于一组控制元素而不是一个控制元素的，设计的时

linux的cgroup系统可谓 容器”的概念，也许也是对linux本身“命名空间”的一种冲击。它是分层的，也可以说是树形的结构， 控制一组元素，在这个ROOT当中可以建立很多的“组”(cgroup)，每一个组还可以建立下级的组...。 up_subsys，比如cpuset，memory，ns等，怎么确定呢？这是通过文件系统的mount实现的，当你执行

mount -t cgroup cgroup -o

的时候，你就建立一个RC 以在my目录中执行mkdir group1 group2，这样就建立了两个cgroup，实际上，当你mount的时候，系统就建立了一个虚拟的组group-root，如果你执行cat my/tasks，你会发现它包含了所有的进程，如果你执行cat my/group1/tasks，你会发现它是空的，因为还没有任何的进程被加入进去。现在执行echo 761>my/group1/tasks，那么pid为761的进程将被加入到group1，通过修改group1目录下的文件就可以对这个group1中当前tasks文件中包含的所有的进程进行控制了。这一切是如何实现的？实际上linux内核代码的cgroup子系统实现了类似数据库的结构，包括表结构和查询引擎，通过阅读代码可以看出，每一个进程task包含一个css_set类型的字段：

```
struct css_set {
```

```
    struct kref ref;
    struct hlist_node hlist;
    struct list_head tasks;      //解决冗余，包含所有使用这个set的进程
    struct list_head cg_links;   //解决冗余，包含所有参与管理这个set的cgroup。
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
};
```

既然它被一组subsys控制，为何不直接将每个subsys本身包含在task中呢？这是为了解决数据冗余的问题，因为其他的进程也可以受这些subsys控制。

有必要说一下上面的css_set结构中的cg_links字段，它包含了所有的参与管理这个set的cgroup，为何会这样呢？难道一个set不是由一个cgroup管理的吗？不是的，要知道管理是基于“一组”subsys的，而这一组并不一定是全部的编译进内核也就是内核支持的subsys。比如你分别用-o参数cpu，memory...mount了5个cgroup文件系统，那么一个进程关联的css_set就会由5个cgroup管理，每一个mount的ROOT会管理一个，这个ROOT会用文件系统的方式管理进程分别属于该ROOT的哪个cgroup。

下面看一下cgroup_subsys_state，这个结构可以看作静态cgroup_subsys结构的动态实例，静态的cgroup_subsys中包含了一些通用的方法，而动态的cgroup_subsys_state则仅仅是一个父类，具体的数据和额外的方法通过继承它来实现，比如：

```
struct mem_cgroup {
    struct cgroup_subsys_state css;
    struct res_counter res;
    struct mem_cgroup_lru_info info;
    int    prev_priority;    /* for recording reclaim priority */
    struct mem_cgroup_stat stat;
};
```

任何时候，只要你得到了一个cgroup_subsys_state，并且根据其subsys_id确认它是关于memory的，那么就可以通过：

```
static inline struct cgroup_subsys_state *task_subsys_state(
    struct task_struct *task, int subsys_id)
{
```

```
    return rcu_dereference(task->cgroups->subsys[subsys_id]);
}
```

和

```
container_of(task_subsys_state(p, mem_cgroup_subsys_id), struct mem_cgroup, css);
```

来取得这个可被称为子类实例的mem_cgroup，接下来就可以操作它的数据了。终于可以看一下cgroup_subsys_state了：

```
struct cgroup_subsys_state {
    struct cgroup *cgroup;
    atomic_t refcnt;
    unsigned long flags;
};
```

这个结构很简单，cgroup是它绑定的一个cgroup实例，从名称上也可以看出cgroup_subsys_state结构是动态的，它表示进程的subsys的state，由于它是被cgroup管理的，因此它也只能有一个cgroup与其绑定。

接下来看一下cgroup结构，这好像是一个重量级的结构，其实不然，它仅仅起到一个粘合的作用，换句话说就是管理者：

```
struct cgroup {
    unsigned long flags;
    atomic_t count;
    struct list_head sibling;     //此和以下几个实现了树型结构
    struct list_head children;
    struct cgroup *parent;
    struct dentry *dentry;
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT]; //这里仅包含对应ROOT相关的subsys
    struct cgroupfs_root *root;     //对应的那个ROOT
    struct cgroup *top_cgroup;
    struct list_head css_sets;    //包含所有的它参与管理的css_set
    struct list_head release_list;
```

}

前面提到过，进程要显式加入一个ROOT的cgroup，在加入的时候可能会为进程绑定一个新的css_set(必须保证css_set的subsys数组完全相同才能重用，如果之前没有这样的css_set建立，只好新建立一个)，只要绑定了一个新的css_set，这个set就要加入到cgroup的css_sets链表中，最简单的可以在css_set中添加一个字段用于此目的，与此同时cgroup中也要增加一个list_head结构用来链接css_set的cg_links字段，这样做为何不好呢？它增加了两个数据结构的耦合性，同时也增加了数据的冗余性，因为一个cgroup的ROOT负责一组subsys，一个进程也是和一组subsys关联，因此只需要一个进程的一组subsys中被同一个ROOT管理的第一个加入到cgroup链表中就可以表示一个进程受到了这个cgroup的管理，比如进程p1加入group1，该group1的ROOT管理cpu和memory，那么其css_set的subsys数组中只需要cpu_id的这个subsys加入cgroup的css_sets链表就可以了，为了代码的简单，因此引入了一个中间结构，那就是cg_cgroup_link：

```
struct cg_cgroup_link {
    struct list_head cgrp_link_list;    //代表一个css_set加入到cgroup
    struct list_head cg_link_list;      //代表一个cgroup加入到css_set
    struct css_set *cg;                 //指回css_set
    ... //后续的内核还要指回cgroup，这里的内核是2.6.26
};
```

在此必须说一下为何要有cg_cgroup_link这个结构体。如果在css_set中和cgroup中直接加入链表元素是解决不了多对多问题的，比如所有参与管理一个css_set的cgroup都将其链表元素加入 css_set的链表，反过来css_set的链表元素也应该加入一个cgroup的链表，代表它是该cgroup管理的css_set之一，现在问题来了，前面说过一个css_set可以属于很多ROOT，那么它到底加入哪个cgroup的链表呢？毕竟css_set和cgroup之间是如此单项一对多的关系耦合，因此解除耦合的办法就是设计一个中间结构，那就是cg_cgroup_link。

很多时候，很多人在网上写了一大堆关于分析“linux内核”的文章，很多文章都是仅仅分析代码流程，但是很少有文章能说明为何这么做(除非文章的作者着手提交一个补丁或者其它...)。其实linux内核就是一个数据库设计的教程，它不但展示了表结构，而且还有查询引擎，故而linux内核绝对是绝妙的哦！比如在input子系统中，input_handle这个结构体也是和cg_cgroup_link意义一样的，也是为了解决多对多的问题而设置的，还有一个明显的例子，那就是linux内核中的总线驱动架构。

由此看来，学习linux内核可以学到两大当今时髦的东西，一个就是OO，另一个就是数据库的设计，千万不要以为linux内核仅仅是底层的东西，搞应用的人不用学习，其实各个领域是相通的，我相信，当一个顶级的文学家听说了广义相对论的时候，他也一定会提出一些自己的看法的。我经常看历史著作，那些作者们看起来对任何领域都很感兴趣...

附：看一下cgroup的静态数据结构们吧

首先看一下一个静态的超类，那就是cgroup_subsys，它包含了一系列的接口，但是没有实现！

```
struct cgroup_subsys {
    struct cgroup_subsys_state *(*create)(struct cgroup_subsys *ss,
                        struct cgroup *cgrp);
    ...//类似的接口
    int subsys_id;
    int active;
    int disabled;
    int early_init;
#define MAX_CGROUP_TYPE_NAMELEN 32
    const char *name;
    struct cgroupfs_root *root;
    struct list_head sibling; //用于挂载一系列的state
    void *private;            //用于扩展
};
```

每一个挂载(mount)的cgroup文件系统都有一个cgroupfs_root实例：

```
struct cgroupfs_root {
    struct super_block *sb;
    unsigned long subsys_bits;
    unsigned long actual_subsys_bits;
    struct list_head subsys_list;      //本ROOT关注的subsys
    struct cgroup top_cgroup;
    int number_of_cgroups;
    struct list_head root_list;
```

```
    unsigned long flags;
    char release_agent_path[PATH_MAX];
};
```


- 上一篇 关于linux内核以及原始人的想法
- 下一篇 使用jni接口完成android本地程序的运行

主题推荐 设计 管理 linux linux内核 文件系统

猜你在找

QEMU的PCI总线与设备下
C++ 标准模板库STL multimap 使用方法与应用介绍一
cloud foundry dea源码解析
Vuforia SDK---- AR空气净化器项目总结
java 中关于信号的处理在linux下的实现
ubuntu 1404 编译ogre
Gerrit审核流程管理系统搭建Gerrit + H2 + Git + OpenDJ
慎用strip
openstack 命令行管理十二 - 内部网络instance专用管理 备忘
OpenStack Nova虚拟机初始化user-data & Cloud-init

查看评论

honkiko 2楼 2013-07-10 12:55发表

我读了您关于memory cgroup的总结，受益匪浅。

关于cg_cgroup_link这个结构，我有两个疑问。
注释说这个cg_cgroup_link是为了cgroup和css_set之间的关联。
cgroup跟css_set之间应该是多对多的关系。
一个css_set，记录一个进程对于各个subsystem, 分别属于哪个cgroup。这个信息记录在css_set->subsys数组里面。
一个cgroup，可能被很多歌css_set所引用。这个信息，通过在cgroup里面放一个css_set的链表就可以了。

实际的代码中，没有这样做，而是引入了一个cg_cgroup_link结构。定义如下：

```
/* Link structure for associating css_set objects with cgroups */
struct cg_cgroup_link {
struct list_head cgrp_link_list; /*关联于同一个css_set的cg_cgroup_link->cgrp_link_list, 都挂在cgroup->css_sets链表头上*/
struct cgroup *cgrp;
struct list_head cg_link_list; /*关联于同一个cgroup的cg_cgroup_link->cg_link_list, 都挂在css_set->cg_links链表头上*/
struct css_set *cg;
};
```

问题一： 我对这个cg_cgroup_link结构存在的必要性始终不太明白。通过把cg_cgroup_link里面的cgrp_link_list嵌入到css_set里面，把cg_link_list嵌入到cgroup里面，完全可以达到同样的功能。无法理解引入cg_cgroup_link带来了什么好处。
问题二：css_set->cg_links好像也不需要。因为通过css_set->subsys这个数组，就能知道这个css_set对应哪些cgroup了。

---

lxl123 1楼 2012-03-28 16:42发表

好文!

Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap