Linux Cgroups 详解

王喆锋 zhefwang@gmail.com

Cgroups 是什么？

Cgroups 是 control groups 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组（process groups）所使用的物理资源（如：cpu,memory,IO 等等）的机制。最初由 google 的工程师提出，后来被整合进 Linux 内核。Cgroups 也是 LXC 为实现虚拟化所使用的资源管理手段，可以说没有 cgroups 就没有 LXC。

Cgroups 可以做什么？

Cgroups 最初的目标是为资源管理提供的一个统一的框架，既整合现有的 cpuset 等子系统，也为未来开发新的子系统提供接口。现在的 cgroups 适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化（OS Level Virtualization）。Cgroups 提供了一下功能：
1.限制进程组可以使用的资源数量（Resource limiting ）。比如：memory 子系统可以为进程组设定一个 memory 使用上限，一旦进程组使用的内存达到限额再申请内存，就会出发 OOM（out of memory）。
2.进程组的优先级控制（Prioritization ）。比如：可以使用 cpu 子系统为某个进程组分配特定 cpu share。
3.记录进程组使用的资源数量（Accounting ）。比如：可以使用 cpuacct 子系统记录某个进程组使用的 cpu 时间
4.进程组隔离（isolation）。比如：使用 ns 子系统可以使不同的进程组使用不同的 namespace，以达到隔离的目的，不同的进程组有各自的进程、网络、文件系统挂载空间。
5.进程组控制（control）。比如：使用 freezer 子系统可以将进程组挂起和恢复。

Cgroups 相关概念及其关系

相关概念

1.任务（task）。在 cgroups 中，任务就是系统的一个进程。
2.控制族群（control group）。控制族群就是一组按照某种标准划分的进程。Cgroups 中的资源控制都是以控制族群为单位实现。一个进程可以加入到某个控制族群，也从一个进程组迁移到另一个控制族群。一个进程组的进程可以使用 cgroups 以控制族群为单位分配的资源，同时受到 cgroups 以控制族群为单位设定的限制。
3.层级（hierarchy）。控制族群可以组织成 hierarchical 的形式，既一颗控制族群树。控制族群树上的子节点控制族群是父节点控制族群的孩子，继承父控制族群的特定的属性。
4.子系统（subsytem）。一个子系统就是一个资源控制器，比如 cpu 子系统就是控制 cpu 时间分配的一个控制器。子系统必须附加（attach）到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制。

相互关系

1.每次在系统中创建新层级时，该系统中的所有任务都是那个层级的默认 cgroup（我们称之为 root cgroup ，此cgroup在创建层级时自动创建，后面在该层级中创建的cgroup都是此cgroup的后代）的初始成员。
2.一个子系统最多只能附加到一个层级。
3.一个层级可以附加多个子系统。
4.一个任务可以是多个cgroup的成员，但是这些cgroup必须在不同的层级。
5.系统中的进程（任务）创建子进程（任务）时，该子任务自动成为其父进程所在 cgroup 的成员。然后可根据需要将该子任务移动到不同的 cgroup 中，但开始时它总是继承其父任务的cgroup。

Cgroups 子系统介绍

blkio -- 这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等等）。
cpu -- 这个子系统使用调度程序提供对 CPU 的 cgroup 任务访问。
cpuacct -- 这个子系统自动生成 cgroup 中任务所使用的 CPU 报告。
cpuset -- 这个子系统为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。
devices -- 这个子系统可允许或者拒绝 cgroup 中的任务访问设备。
freezer -- 这个子系统挂起或者恢复 cgroup 中的任务。
memory -- 这个子系统设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。
net_cls -- 这个子系统使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（tc）识别从具体 cgroup 中生成的数据包。
ns -- 名称空间子系统。

Cgroups 如何实现

数据结构

我们从进程出发来剖析 cgroups 相关数据结构之间的关系。
在 Linux 中，管理进程的数据结构是 task_struct，其中与 cgroups 有关的：

```
#ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock */
    struct css_set *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock */
    struct list_head cg_list;
#endif
```

其中cgroups指针指向了一个css_set结构，而css_set存储了与进程相关的cgroups信息。Cg_list

是一个嵌入的list_head结构，用于将连到同一个css_set的进程组织成一个链表。下面我们来看css_set的结构：

```
struct css_set {
    atomic_t refcount;
    struct hlist_node hlist;
    struct list_head tasks;
    struct list_head cg_links;
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    struct rcu_head rcu_head;
};
```

其中refcount是该css_set的引用数，因为一个css_set可以被多个进程共用，只要这些进程的cgroups信息相同，比如：在所有已创建的层级里面都在同一个cgroup里的进程。

hlist是嵌入的hlist_node，用于把所有css_set组织成一个hash表，这样内核可以快速查找特定的css_set。

tasks指向所有连到此css_set的进程连成的链表。

cg_links指向一个由struct cg_cgroup_link连成的链表。

Subsys 是一个指针数组，存储一组指向 cgroup_subsys_state 的指针。一个cgroup_subsys_state就是进程与一个特定子系统相关的信息。通过这个指针数组，进程就可以获得相应的cgroups控制信息了。

下面我们就来看cgroup_subsys_state的结构：

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

cgroup指针指向了一个cgroup结构，也就是进程属于的cgroup。进程受到子系统的控制，实际上是通过加入到特定的cgroup实现的，因为cgroup在特定的层级上，而子系统又是附加到曾经上的。通过以上三个结构，进程就可以和cgroup连接起来了：task_struct->css_set->cgroup_subsys_state->cgroup。

下面我们再来看cgroup的结构：

```
struct cgroup {
    unsigned long flags;
    atomic_t count;
    struct list_head sibling;
    struct list_head children;
    struct cgroup *parent;
    struct dentry *dentry;
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
    struct cgroupfs_root *root;
    struct cgroup *top_cgroup;
    struct list_head css_sets;
    struct list_head release_list;
    struct list_head pidlists;
```

```
    struct mutex pidlist_mutex;
    struct rcu_head rcu_head;
    struct list_head event_list;
    spinlock_t event_list_lock;
};
```

sibling,children和parent三个嵌入的list_head负责将同一层级的cgroup连接成一颗cgroup树。

subsys是一个指针数组，存储一组指向cgroup_subsys_state的指针。这组指针指向了此cgroup跟各个子系统相关的信息，这个跟css_set中的道理是一样的。

root指向了一个cgroupfs_root的结构，就是cgroup所在的层级对应的结构体。这样以来，之前谈到的几个cgroups概念就全部联系起来了。

top_cgroup指向了所在层级的根cgroup，也就是创建层级时自动创建的那个cgroup。

css_set指向一个由struct cg_cgroup_link连成的链表，跟css_set中cg_links一样。

下面我们来分析一个css_set和cgroup之间的关系。我们先看一下 cg_cgroup_link的结构

```
struct cg_cgroup_link {
    struct list_head cgrp_link_list;
    struct cgroup *cgrp;
    struct list_head cg_link_list;
    struct css_set *cg;
};
```

cgrp_link_list连入到cgroup->css_set指向的链表，cgrp则指向此cg_cgroup_link相关的cgroup。

Cg_link_list则连入到css_set->cg_links指向的链表,cg则指向此cg_cgroup_link相关的css_set。

那为什么要这样设计呢？

那是因为cgroup和css_set是一个多对多的关系，必须添加一个中间结构来将两者联系起来，这跟数据库模式设计是一个道理。cg_cgroup_link中的cgrp和cg就是此结构体的联合主键，而cgrp_link_list和cg_link_list分别连入到cgroup和css_set相应的链表，使得能从cgroup或css_set都可以进行遍历查询。

那为什么cgroup和css_set是多对多的关系呢？

一个进程对应css_set，一个css_set就存储了一组进程（应该有可能被几个进程共享，所以是一组）跟各个子系统相关的信息，但是这些信息有可能不是从一个cgroup那里获得的，因为一个进程可以同时属于几个cgroup，只要这些cgroup不在同一个层级。举个例子：我们创建一个层级A，A上面附加了cpu和memory两个子系统，进程B属于A的根cgroup；然后我们再创建一个层级C，C上面附加了ns和blkio两个子系统，进程B同样属于C的根cgroup；那么进程B对应的cpu和memory的信息是从A的根cgroup获得的，ns和blkio信息则是从C的根cgroup获得的。因此，一个css_set存储的cgroup_subsys_state可以对应多个cgroup。另一方面，cgroup也存储了一组cgroup_subsys_state，这一组cgroup_subsys_state则是cgroup从所在的层级附加的子系统获得的。一个cgroup中可以有多个进程，而这些进程的css_set不一定都相同，因为有些进程可能还加入了其他cgroup。但是同一个cgroup中的进程与该cgroup关联的cgroup_subsys_state都受到该cgroup的管理（cgroups中进程控制是以cgroup为单位的）的，所以一个cgrouop也可以对应多个css_set。

那为什么要这样一个结构呢？

从前面的分析，我们可以看出从task到cgroup是很容易定位的，但是从cgroup获取此cgroup

的所有的task就必须通过这个结构了。每个进程都会指向一个css_set，而与这个css_set关联的所有进程都会链入到css_set->tasks链表.而cgroup又通过一个中间结构cg_cgroup_link来寻找所有与之关联的所有css_set，从而可以得到与cgroup关联的所有进程。

最后让我们看一下层级和子系统对应的结构体。层级对应的结构体是cgroupfs_root:

```
struct cgroupfs_root {
    struct super_block *sb;
    unsigned long subsys_bits;
    int hierarchy_id;
    unsigned long actual_subsys_bits;
    struct list_head subsys_list;
    struct cgroup top_cgroup;
    int number_of_cgroups;
    struct list_head root_list;
    unsigned long flags;
    char release_agent_path[PATH_MAX];
    char name[MAX_CGROUP_ROOT_NAMELEN];
};
```

sb指向该层级关联的文件系统超级块

subsys_bits和actual_subsys_bits分别指向将要附加到层级的子系统和现在实际附加到层级的子系统，在子系统附加到层级时使用

hierarchy_id是该层级唯一的id

top_cgroup指向该层级的根cgroup

number_of_cgroups记录该层级cgroup的个数

root_list是一个嵌入的list_head，用于将系统所有的层级连成链表

子系统对应的结构体是cgroup_subsys:

```
struct cgroup_subsys {
    struct cgroup_subsys_state *(*create)(struct cgroup_subsys *ss,
                        struct cgroup *cgrp);
    int (*pre_destroy)(struct cgroup_subsys *ss, struct cgroup *cgrp);
    void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cgrp);
    int (*can_attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
              struct task_struct *tsk, bool threadgroup);
    void (*cancel_attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
              struct task_struct *tsk, bool threadgroup);
    void (*attach)(struct cgroup_subsys *ss, struct cgroup *cgrp,
            struct cgroup *old_cgrp, struct task_struct *tsk,
            bool threadgroup);
    void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
    void (*exit)(struct cgroup_subsys *ss, struct task_struct *task);
    int (*populate)(struct cgroup_subsys *ss,
            struct cgroup *cgrp);
    void (*post_clone)(struct cgroup_subsys *ss, struct cgroup *cgrp);
    void (*bind)(struct cgroup_subsys *ss, struct cgroup *root);
```

```
    int subsys_id;
    int active;
    int disabled;
    int early_init;
    bool use_id;
#define MAX_CGROUP_TYPE_NAMELEN 32
    const char *name;
    struct mutex hierarchy_mutex;
    struct lock_class_key subsys_key;
    struct cgroupfs_root *root;
    struct list_head sibling;
    struct idr idr;
    spinlock_t id_lock;
    struct module *module;
};
```

Cgroup_subsys定义了一组操作，让各个子系统根据各自的需要去实现。这个相当于C++中抽象基类，然后各个特定的子系统对应cgroup_subsys则是实现了相应操作的子类。类似的思想还被用在了cgroup_subsys_state中，cgroup_subsys_state并未定义控制信息，而只是定义了各个子系统都需要的共同信息，比如该cgroup_subsys_state从属的cgroup。然后各个子系统再根据各自的需要去定义自己的进程控制信息结构体，最后在各自的结构体中将cgroup_subsys_state包含进去，这样通过Linux内核的container_of等宏就可以通过cgroup_subsys_state来获取相应的结构体。

Cgroup 文件系统

Cgroups 用户空间管理

Cgroups 用户空间的管理是通过 cgroup 文件系统实现的。
比如要创建一个层级：

```
mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
```

这个命令就创建一个名为 cpu_and_mem 的层级，这个层级上附加了 cpu,cpuset,memory 三个子系统，并把层级挂载到了/cgroup/cpu_and_mem.
创建一个 cgroup：

```
cd /cgroup/cpu_and_mem
mkdir foo
```

通过以上两个命令，我们就在刚才创建的层级下创建了一个叫 foo 的 cgroup。
你再 cd foo，然后 ls
你会发现一些文件，这是 cgroups 相关子系统的控制文件，你可以读取这些控制文件，这些控制文件存储的值就是对相应的 cgrouop 的控制信息，你也可以写控制文件来更改控制信息。
在这些文件中，有一个叫 tasks 的文件，里面的包含了所有属于这个 cgroup 的进程的进程号。在刚才创建的 foo 下，你 cat tasks，应该是空的，因为此时这个 cgroup 里面还没有进程。你 cd /cgroup/cpu_and_mem 再 cat tasks，你可以看到系统中所有进程的进程号，这是因为每创建一个层级的时候，系统的所有进程都会自动被加到该层级的根 cgroup 里面。Tasks 文

件不仅可以读，还可以写，你将一个进程的进程号写入到某个 cgroup 目录下的 tasks 里面，你就将这个进程加入了相应的 cgroup。

Cgroup 文件系统的实现

在讲 cgroup 文件系统的实现之前，必须简单的介绍一下 Linux VFS。
VFS 是所谓的虚拟文件系统转换，是一个内核软件层，用来处理与 Unix 标准文件系统的所有系统调用。VFS 对用户提供统一的读写等文件操作调用接口，当用户调用读写等函数时，内核则调用特定的文件系统实现。具体而言，文件在内核内存中是一个 file 数据结构来表示的。这个数据结构包含一个 f_op 的字段，该字段中包含了一组指向特定文件系统实现的函数指针。当用户执行 read()操作时，内核调用 sys_read(),然后 sys_read（）查找到指向该文件属于的文件系统的读函数指针，并调用它，即 file->f_op->read().
VFS 其实是面向对象的，在这里，对象是一个软件结构，既定义数据也定义了之上的操作。处于效率，Linux 并没有采用 C++之类的面向对象的语言，而是采用了 C 的结构体，然后在结构体里面定义了一系列函数指针，这些函数指针对应于对象的方法。
VFS 文件系统定义了以下对象模型：
超级块对象(superblock object)
存放已安装文件系统的有关信息。
索引节点对象(inode object)
存放关于具体文件的一般信息。
文件对象（file object）
存放打开文件与进程之间的交互信息
目录项对象(dentry object)
存放目录项与对应文件进行链接的有关信息。

基于 VFS 实现的文件系统，都必须实现定义这些对象，并实现这些对象中定义的函数指针。cgroup 文件系统也不例外，下面我们来看 cgroups 中这些对象的定义。
cgroup 文件系统的定义：

```
static struct file_system_type cgroup_fs_type = {
    .name = "cgroup",
    .get_sb = cgroup_get_sb,
    .kill_sb = cgroup_kill_sb,
};
```

这里有定义了两个函数指针，定义了一个文件系统必须实现了的两个操作 get_sb,kill_sb，即获得超级块和释放超级块。这两个操作会在使用 mount 系统调用挂载 cgroup 文件系统时使用。
cgroup 超级块的定义：

```
static const struct super_operations cgroup_ops = {
    .statfs = simple_statfs,
    .drop_inode = generic_delete_inode,
    .show_options = cgroup_show_options,
    .remount_fs = cgroup_remount,
};
```

Cgroup 索引块定义：

```
static const struct inode_operations cgroup_dir_inode_operations = {
    .lookup = simple_lookup,
    .mkdir = cgroup_mkdir,
    .rmdir = cgroup_rmdir,
    .rename = cgroup_rename,
};
```

在 cgroup 文件系统中，使用 mkdir 创建 cgroup 或者用 rmdir 删除 cgroup 时，就会调用相应的函数指针指向的函数。比如：使用 mkdir 创建 cgroup 时，会调用 cgroup_mkdir，然后在 cgroup_mkdir 中再调用具体实现的 cgroup_create 函数。

Cgroup 文件操作定义：

```
static const struct file_operations cgroup_file_operations = {
    .read = cgroup_file_read,
    .write = cgroup_file_write,
    .llseek = generic_file_llseek,
    .open = cgroup_file_open,
    .release = cgroup_file_release,
};
```

在 cgroup 文件系统中，对目录下的控制文件进行操作时，会调用该结构体中指针指向的函数。比如：对文件进行读操作时，会调用 cgroup_file_read，在 cgroup_file_read 中，会根据需要调用该文件对应的 cftype 结构体定义的对应读函数。

我们再来看 cgroup 文件系统中的 cgroups 控制文件。Cgroups 定义一个 cftype 的结构体来管理控制文件。下面我们来看 cftype 的定义：

```
struct cftype {
    char name[MAX_CFTYPE_NAME];
    int private;   /*
    mode_t mode;
    size_t max_write_len;

    int (*open)(struct inode *inode, struct file *file);
    ssize_t (*read)(struct cgroup *cgrp, struct cftype *cft,
            struct file *file,
            char __user *buf, size_t nbytes, loff_t *ppos);
    u64 (*read_u64)(struct cgroup *cgrp, struct cftype *cft);
    s64 (*read_s64)(struct cgroup *cgrp, struct cftype *cft);
    int (*read_map)(struct cgroup *cont, struct cftype *cft,
            struct cgroup_map_cb *cb);
    int (*read_seq_string)(struct cgroup *cont, struct cftype *cft,
                struct seq_file *m);

    ssize_t (*write)(struct cgroup *cgrp, struct cftype *cft,
            struct file *file,
            const char __user *buf, size_t nbytes, loff_t *ppos);
    int (*write_u64)(struct cgroup *cgrp, struct cftype *cft, u64 val);
    int (*write_s64)(struct cgroup *cgrp, struct cftype *cft, s64 val);
```

```
    int (*write_string)(struct cgroup *cgrp, struct cftype *cft,
                const char *buffer);
    int (*trigger)(struct cgroup *cgrp, unsigned int event);

    int (*release)(struct inode *inode, struct file *file);
    int (*register_event)(struct cgroup *cgrp, struct cftype *cft,
            struct eventfd_ctx *eventfd, const char *args); /*
    void (*unregister_event)(struct cgroup *cgrp, struct cftype *cft,
            struct eventfd_ctx *eventfd);
};
```

cftype 中除了定义文件的名字和相关权限标记外，主要是定义了对文件进行操作的函数指针。不同的文件可以有不同的操作，对文件进行操作时，相关函数指针指向的函数会被调用。

综合上面的分析，cgroups 通过实现 cgroup 文件系统来为用户提供管理 cgroup 的工具，而 cgroup 文件系统是基于 Linux VFS 实现的。相应地，cgroups 为控制文件定义了相应的数据结构 cftype，对其操作由 cgroup 文件系统定义的通过操作捕获，再调用 cftype 定义的具体实现。

子系统的实现

cpu 子系统

cpu子系统用于控制cgroup中所有进程可以使用的cpu时间片。附加了cpu子系统的hierarchy下面建立的cgroup的目录下都有一个cpu.shares的文件，对其写入整数值可以控制该cgroup获得的时间片。例如：在两个 cgroup 中都将 cpu.shares 设定为 1 的任务将有相同的 CPU 时间，但在 cgroup 中将 cpu.shares 设定为 2 的任务可使用的 CPU 时间是在 cgroup 中将 cpu.shares 设定为 1 的任务可使用的 CPU 时间的两倍。

cpu子系统是通过Linux CFS调度器实现的。所以在介绍cpu子系统之前，先简单说一下CFS调度器。按照作者Ingo Molnar的说法："CFS百分之八十的工作可以用一句话概括：CFS在真实的硬件上模拟了完全理想的多任务处理器"。在“完全理想的多任务处理器”下，每个进程都能同时获得CPU的执行时间。当系统中有两个进程时，CPU的计算时间被分成两份，每个进程获得50%。然而在实际的硬件上，当一个进程占用CPU时，其它进程就必须等待。所以CFS将惩罚当前进程，使其它进程能够在下次调度时尽可能取代当前进程。最终实现所有进程的公平调度。

CFS调度器将所有状态为RUNABLE的进程都被插入红黑树。在每个调度点，CFS调度器都会选择红黑树的最左边的叶子节点作为下一个将获得cpu的进程。那红黑树的键值是怎么计算的呢？红黑树的键值是进程所谓的虚拟运行时间。一个进程的虚拟运行时间是进程时间运行的时间按整个红黑树中所有的进程数量normalized的结果。

每次tick中断，CFS调度器都要更新进程的虚拟运行时间，然后调整当前进程在红黑树中的位置，调整完成后如果发现当前进程不再是最左边的叶子，就标记need_resched 标志，中断返回时就会调用scheduler()完成进程切换。

最后再说一下，进程的优先级和进程虚拟运行时间的关系。前面提到了，每次tick中断，CFS调度器都要更新进程的虚拟运行时间。那这个时间是怎么计算的呢？CFS首先计算出进程的时间运行时间delta_exec，然后计算normalized后的delta_exec_weighted，最后再将delta_exec_weighted 加到进程的虚拟运行时间上。跟进程优先级有关的就是

delta_exec_weighted，delta_exec_weighted=delta_exec_weighted*NICE_0_LOAD/se->load，其中NICE_0_LOAD是个常量，而se->load跟进程的nice值成反比，因此进程优先级越高（nice值越小）则se->load越大，则计算出来的delta_exec_weighted越小，这样进程优先级高的进程就可以获得更多的cpu时间。

介绍完CFS调度器，我们开始介绍cpu子系统是如何通过CFS调度器实现的。CFS调度器不仅支持基于进程的调度，还支持基于进程组的组调度。CFS中定义了一个task_group的数据结构来管理组调度。

```
struct task_group {
    struct cgroup_subsys_state css;

#ifdef CONFIG_FAIR_GROUP_SCHED
    /* schedulable entities of this group on each cpu */
    struct sched_entity **se;
    /* runqueue "owned" by this group on each cpu */
    struct cfs_rq **cfs_rq;
    unsigned long shares;
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;

    struct rt_bandwidth rt_bandwidth;
#endif

    struct rcu_head rcu;
    struct list_head list;

    struct task_group *parent;
    struct list_head siblings;
    struct list_head children;
    };
```

task_group中内嵌了一个cgroup_subsys_state，也就是说进程可以通过cgroup_subsys_state来获取它所在的task_group，同样地cgroup也可以通过cgroup_subsys_state来获取它对应的task_group，因此进程和cgroup都存在了一组cgroup_subsys_state指针。

struct sched_entity **se是一个指针数组，存了一组指向该task_group在每个cpu的调度实体（即一个struct sched_entity）。

struct cfs_rq **cfs_rq也是一个指针数组，存在一组指向该task_group在每个cpu上所拥有的一个可调度的进程队列。

Parent、siblings和children三个指针负责将task_group 连成一颗树，这个跟cgroup树类似。

有了这个数据结构，我们来CFS在调度的时候是怎么处理进程组的。我们还是从CFS对tick中断的处理开始。

CFS对tick中断的处理在task_tick_fair中进行，在task_tick_fair中有：

```
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }
```

我们首先来看一下在组调度的情况下，for_each_sched_entity是怎么定义的：

```
#define for_each_sched_entity(se) \
        for (; se; se = se->parent)
```

即从当前进程的se开始，沿着task_group树从下到上对se调用entity_tick，即更新各个se的虚拟运行时间。

在非组调度情况下，

```
#define for_each_sched_entity(se) \
        for (; se; se = NULL)
```

即只会对当前se做处理。

CFS处理完tick中断后，如果有必要就会进行调度，CFS调度是通过pick_next_task_fair函数选择下一个运行的进程的。在pick_next_task_fair中有：

```
    do {
        se = pick_next_entity(cfs_rq);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);
```

在这个循环中，首先从当前的队列选一个se，这个跟非组调度一样的（红黑树最左边的节点），再将se设置成下一个运行的se，再从该se获取该se对应的task_group拥有的cfs_rq（如果该se对应一个进程而非一个task_group的话，cfs_rq会变成NULL），继续这个过程直到cfs_rq为空，即当se对应的是一个进程。

简而言之，同一层的task_group跟进程被当成同样的调度实体来选择，当被选到的是task_group时，则对task_group的孩子节点重复这个过程，直到选到一个运行的进程。因此当设置一个cgroup的shares值时，该cgroup当作一个整体和剩下的进程或其他cgroup分享cpu时间。比如,我在根cgroup下建立cgroup A，将其shares值设1024，再建立cgroup B,将其shares设为2048，再将一些进程分别加入到这两个cgroup中，则长期调度的结果应该是A:B:C=1:2:1（即cpu占用时间,其中C是系统中为加入到A或B的进程）。

引起CFS调度的除了tick中断外，还有就是有新的进程加入可运行队列这种情况。CFS处理这个情况的函数是enqueue_task_fair,在enqueue_task_fair中有：

```
    for_each_sched_entity(se) {
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, flags);
        flags = ENQUEUE_WAKEUP;
    }
```

我们前面已经看过for_each_sched_entity在组调度下的定义了，这里是将当前se和se的直系祖先节点都加入到红黑树（enqueue_entity），而在非组调度情况下，只需要将当前se本身加入即可。造成这种差异的原因，在于在pick_next_task_fair中选择se时，是从上往下的，如果一个se的祖先节点不在红黑树中，它永远都不会被选中。而在非组调度的情况下，se之间并没有父子关系，所有se都是平等独立，在pick_next_task_fair，第一

次选中的肯定就是进程，不需要向下迭代。
类似的处理还发生在将一个se出列（dequeue_task_fair）和put_prev_task_fair中。
以上是cpu系统通过CFS调度器实现以cgroup为单位的cpu时间片分享，下面我们来看一下cpu子系统本身。cpu子系统通过一个cgroup_subsys结构体来管理：

```
struct cgroup_subsys cpu_cgroup_subsys = {
    .name       = "cpu",
    .create     = cpu_cgroup_create,
    .destroy    = cpu_cgroup_destroy,
    .can_attach     = cpu_cgroup_can_attach,
    .attach     = cpu_cgroup_attach,
    .populate   = cpu_cgroup_populate,
    .subsys_id = cpu_cgroup_subsys_id,
    .early_init     = 1,
};
```

Cpu_cgroup_subsys其实是对抽象的cgroup_subsys的实现，其中的函数指针指向了特定于cpu子系统的实现。这里再说一下，Cgroups的整体设计。当用户使用cgroup文件系统，创建cgroup的时候，会调用cgroup目录操作的mkdir指针指向的函数，该函数调用了cgroup_create，而cgroup_create会根据该cgroup关联的子系统，分别调用对应的子系统实现的create指针指向的函数。即做了两次转换，一次从系统通用命令到cgroup文件系统，另一次从cgroup文件系统再特定的子系统实现。
Cgroups中除了通用的控制文件外，每个子系统还有自己的控制文件，子系统也是通过cftype来管理这些控制文件。Cpu系统很重要的一个文件就是cpu.shares文件，因为就是通过这个文件的数值来调节cgroup所占用的cpu时间。Shares文件对应的cftype结构为：

```
#ifdef CONFIG_FAIR_GROUP_SCHED
    {
        .name = "shares",
        .read_u64 = cpu_shares_read_u64,
        .write_u64 = cpu_shares_write_u64,
    },
#endif
```

当对cgroup目录下的文件进行操作时，该结构体中定义的函数指针指向的函数就会被调用.下面我们就在看看这个两个函数的实现吗，从而发现shares文件的值是如何起作用的。

```
static u64 cpu_shares_read_u64(struct cgroup *cgrp, struct cftype *cft)
{
    struct task_group *tg = cgroup_tg(cgrp);

    return (u64) tg->shares;
}
```

比较简单，简单的读取task_group中存储的shares就行了。

```
static int cpu_shares_write_u64(struct cgroup *cgrp, struct cftype *cftype,
                u64 shareval)
{
    return sched_group_set_shares(cgroup_tg(cgrp), shareval);
}
```

则是设定cgroup对应的task_group的shares值。
那这个shares值是怎么起作用的呢？在sched_group_set_shares中有：

```
    tg->shares = shares;
    for_each_possible_cpu(i) {
        /*
         * force a rebalance
         */
        cfs_rq_set_shares(tg->cfs_rq[i], 0);
        set_se_shares(tg->se[i], shares);
    }
```

cfs_rq_set_shares强制做一次cpu SMP负载均衡。真正起作用的是在set_se_shares中，它调用了__set_se_shares，在__set_se_shares中有：

```
    se->load.weight = shares;
    se->load.inv_weight = 0;
```

根据之前我们分析的CFS的调度原理可以知道，load.weight的值越大，算出来的虚拟运行时间越小，进程能使用的cpu时间越多。这样以来，shares值最终就是通过调度实体的load值来起作用的。

cpuset 子系统

cpuset 子系统为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。Cpuset 子系统为定义了一个叫 cpuset 的数据结构来管理 cgroup 中的任务能够使用的 cpu 和内存节点。Cpuset 定义如下：

```
struct cpuset {
    struct cgroup_subsys_state css;

    unsigned long flags;      /* "unsigned long" so bitops work */
    cpumask_var_t cpus_allowed;  /* CPUs allowed to tasks in cpuset */
    nodemask_t mems_allowed;  /* Memory Nodes allowed to tasks */

    struct cpuset *parent;       /* my parent */

    struct fmeter fmeter;     /* memory_pressure filter */

    /* partition number for rebuild_sched_domains() */
    int pn;

    /* for custom sched domain */
    int relax_domain_level;

    /* used for walking a cpuset heirarchy */
    struct list_head stack_list;
};
```

其中 css 字段用于 task 或 cgroup 获取 cpuset 结构。
cpus_allowed 和 mems_allowed 定义了该 cpuset 包含的 cpu 和内存节点。
Parent 字段用于维持 cpuset 的树状结构，stack_list 则用于遍历 cpuset 的层次结构。
Pn 和 relax_domain_level 是跟 Linux 调度域相关的字段，pn 指定了 cpuset 的调度域的分区号，而 relax_domain_level 表示进行 cpu 负载均衡寻找空闲 cpu 的策略。
除此之外，进程的 task_struct 结构体里面还有一个 cpumask_t cpus_allowed 成员，用以存储进程的 cpus_allowed 信息;一个 nodemask_t mems_allowed 成员，用于存储进程的 mems_allowed 信息。
Cpuset 子系统的实现是通过在内核代码加入一些 hook 代码。由于代码比较散，我们逐条分析。
在内核初始化代码（即 start_kernel 函数）中插入了对 cpuset_init 调用的代码，这个函数用于 cpuset 的初始化。
下面我们来看这个函数：

```
int __init cpuset_init(void)
{
    int err = 0;

    if (!alloc_cpumask_var(&top_cpuset.cpus_allowed, GFP_KERNEL))
        BUG();

    cpumask_setall(top_cpuset.cpus_allowed);
    nodes_setall(top_cpuset.mems_allowed);

    fmeter_init(&top_cpuset.fmeter);
    set_bit(CS_SCHED_LOAD_BALANCE, &top_cpuset.flags);
    top_cpuset.relax_domain_level = -1;

    err = register_filesystem(&cpuset_fs_type);
    if (err < 0)
        return err;

    if (!alloc_cpumask_var(&cpus_attach, GFP_KERNEL))
        BUG();

    number_of_cpusets = 1;
    return 0;
}
```

cpumask_setall 和 nodes_setall 将 top_cpuset 能使用的 cpu 和内存节点设置成所有节点。紧接着，初始化 fmeter，设置 top_cpuset 的 load balance 标志。最后注册 cpuset 文件系统，这个是为了兼容性，因为在 cgroups 之前就有 cpuset 了，不过在具体实现时，对 cpuset 文件系统的操作都被重定向了 cgroup 文件系统。
除了这些初始化工作，cpuset 子系统还在 do_basic_setup 函数（此函数在 kernel_init 中被调用）中插入了对 cpuset_init_smp 的调用代码，用于 smp 相关的初始化工作。
下面我们看这个函数：

```
void __init cpuset_init_smp(void)
{
    cpumask_copy(top_cpuset.cpus_allowed, cpu_active_mask);
    top_cpuset.mems_allowed = node_states[N_HIGH_MEMORY];

    hotcpu_notifier(cpuset_track_online_cpus, 0);
    hotplug_memory_notifier(cpuset_track_online_nodes, 10);

    cpuset_wq = create_singlethread_workqueue("cpuset");
    BUG_ON(!cpuset_wq);
}
```

首先，将 top_cpuset 的 cpu 和 memory 节点设置成所有 online 的节点，之前初始化时还不知道有哪些 online 节点所以只是简单设成所有，在 smp 初始化后就可以将其设成所有 online 节点了。然后加入了两个 hook 函数，cpuset_track_online_cpus 和 cpuset_track_online_nodes，这个两个函数将在 cpu 和 memory 热插拔时被调用。

cpuset_track_online_cpus 函数中调用 scan_for_empty_cpusets 函数扫描空的 cpuset，并将其下的进程移到其非空的 parent 下，同时更新 cpuset 的 cpus_allowed 信息。cpuset_track_online_nodes 的处理类似。

那cpuset又是怎么对进程的调度起作用的呢？这个就跟task_struct中cpu_allowed字段有关了。首先，这个cpu_allowed和进程所属的cpuset的cpus_allowed保持一致；其次，在进程被fork出来的时候，进程继承了父进程的cpuset和cpus_allowed字段；最后，进程被fork出来后，除非指定CLONE_STOPPED标记，都会被调用wake_up_new_task唤醒，在wake_up_new_task中有：

```
    cpu = select_task_rq(rq, p, SD_BALANCE_FORK, 0);
    set_task_cpu(p, cpu);
```

即为新 fork 出来的进程选择运行的 cpu，而 select_task_rq 会调用进程所属的调度器的函数，对于普通进程，其调度器是 CFS，CFS 对应的函数是 select_task_rq_fair。在 select_task_rq_fair 返回选到的 cpu 后，select_task_rq 会对结果和 cpu_allowed 比较：

```
    if (unlikely(!cpumask_test_cpu(cpu, &p->cpus_allowed) ||
            !cpu_online(cpu)))
        cpu = select_fallback_rq(task_cpu(p), p);
```

这就保证了新 fork 出来的进程只能在 cpu_allowed 中的 cpu 上运行。

对于被 wake up 的进程来说，在被调度之前，也会调用 select_task_rq 选择可运行的 cpu。这就保证了进程任何时候都只会在 cpu_allowed 中的 cpu 上运行。最后说一下，如何保证 task_struct 中的 cpus_allowd 和进程所属的 cpuset 中的 cpus_allowed 一致。首先，在 cpu 热插拔时，scan_for_empty_cpusets 会更新 task_struct 中的 cpus_allowed 信息，其次对 cpuset 下的控制文件写入操作时也会更新 task_struct 中的 cpus_allowed 信息，最后当一个进程被 attach 到其他 cpuset 时，同样会更新 task_struct 中的 cpus_allowed 信息。

在 cpuset 之前，Linux 内核就提供了指定进程可以运行的 cpu 的方法。通过调用 sched_setaffinity 可以指定进程可以运行的 cpu。Cpuset 对其进行了扩展，保证此调用设定的 cpu 仍然在 cpu_allowed 的范围内。在 sched_setaffinity 中，插入了这样两行代码：

```
    cpuset_cpus_allowed(p, cpus_allowed);
    cpumask_and(new_mask, in_mask, cpus_allowed);
```

其中cpuset_cpus_allowed返回进程对应的cpuset中的cpus_allowed，cpumask_and则将cpus_allowed和调用sched_setaffinity时的参数in_mask相与得出进程新的cpus_allowed。

通过以上代码的嵌入，Linux内核实现了对进程可调度的cpu的控制。下面我们来对memory节点的控制。

Linux中内核分配物理页框的函数有6个，alloc_pages,alloc_page,__get_free_pages,__get_free_page,get_zeroed_page,__get_dma_pages,这些函数最终都通过alloc_pages实现，而alloc_pages又通过__alloc_pages_nodemask实现，在__alloc_pages_nodemask中，调用get_page_from_freelist从zone list中分配一个page，在get_page_from_freelist中调用cpuset_zone_allowed_softwall判断当前节点是否属于mems_allowed。通过附加这样一个判断，保证进程从mems_allowed中的节点分配内存。

Linux在cpuset出现之前，也提供了mbind, set_mempolicy来限定进程可用的内存节点。Cpuset子系统对其做了扩展，扩展的方法跟扩展sched_setaffinity类似，通过导出cpuset_mems_allowed，返回进程所属的cupset允许的内存节点，对mbind，set_mempolicy的参数进行过滤。

最后让我们来看一下，cpuset子系统最重要的两个控制文件：

```
    {
        .name = "cpus",
        .read = cpuset_common_file_read,
        .write_string = cpuset_write_resmask,
        .max_write_len = (100U + 6 * NR_CPUS),
        .private = FILE_CPULIST,
    },

    {
        .name = "mems",
        .read = cpuset_common_file_read,
        .write_string = cpuset_write_resmask,
        .max_write_len = (100U + 6 * MAX_NUMNODES),
        .private = FILE_MEMLIST,
    },
```

通过cpus文件，我们可以指定进程可以使用的cpu节点，通过mems文件，我们可以指定进程可以使用的memory节点。

这两个文件的读写都是通过cpuset_common_file_read和cpuset_write_resmask实现的，通过private属性区分。

在cpuset_common_file_read中读出可用的cpu或memory节点；在cpuset_write_resmask中则根据文件类型分别调用update_cpumask和update_nodemask更新cpu或memory节点信息。

memory 子系统

memory 子系统可以设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。memory子系统是通过linux的resource counter机制实现的。下面我们就先来看一下resource counter机制。

resource counter是内核为子系统提供的一种资源管理机制。这个机制的实现包括了用于记录资源的数据结构和相关函数。Resource counter定义了一个res_counter的结构体来管理特定资源，定义如下：

```
struct res_counter {
    unsigned long long usage;
    unsigned long long max_usage;
    unsigned long long limit;
    unsigned long long soft_limit;
    unsigned long long failcnt;  /*
    spinlock_t lock;
    struct res_counter *parent;
};
```

Usage 用于记录当前已使用的资源，max_usage 用于记录使用过的最大资源量，limit 用于设置资源的使用上限，进程组不能使用超过这个限制的资源，soft_limit 用于设定一个软上限，进程组使用的资源可以超过这个限制，failcnt 用于记录资源分配失败的次数，管理可以根据这个记录，调整上限值。Parent 指向父节点，这个变量用于处理层次性的资源管理。

除了这个关键的数据结构，resource counter 还定义了一系列相关的函数。下面我们来看几个关键的函数。

```
void res_counter_init(struct res_counter *counter, struct res_counter
*parent)
{
    spin_lock_init(&counter->lock);
    counter->limit = RESOURCE_MAX;
    counter->soft_limit = RESOURCE_MAX;
    counter->parent = parent;
}
```

这个函数用于初始化一个 res_counter。

第二个关键的函数是 `int res_counter_charge(struct res_counter *counter, unsigned long val,struct res_counter **limit_fail_at)`。当资源将要被分配的时候，资源就要被记录到相应的res_counter里。这个函数作用就是记录进程组使用的资源。在这个函数中有：

```
    for (c = counter; c != NULL; c = c->parent) {
        spin_lock(&c->lock);
        ret = res_counter_charge_locked(c, val);
        spin_unlock(&c->lock);
        if (ret < 0) {
            *limit_fail_at = c;
            goto undo;
```

```
        }
    }
```

在这个循环里，从当前res_counter开始，从下往上逐层增加资源的使用量。我们来看一下res_counter_charge_locked这个函数，这个函数顾名思义就是在加锁的情况下增加使用量。实现如下：

```
{
    if (counter->usage + val > counter->limit) {
        counter->failcnt++;
        return -ENOMEM;
    }

    counter->usage += val;
    if (counter->usage > counter->max_usage)
        counter->max_usage = counter->usage;
    return 0;
}
```

首先判断是否已经超过使用上限，如果是的话就增加失败次数，返回相关代码；否则就增加使用量的值，如果这个值已经超过历史最大值，则更新最大值。

第三个关键的函数是**void res_counter_uncharge(struct** res_counter *counter, **unsigned long** val)。当资源被归还到系统的时候，要在相应的res_counter减轻相应的使用量。这个函数作用就在于在于此。实现如下：

```
    for (c = counter; c != NULL; c = c->parent) {
        spin_lock(&c->lock);
        res_counter_uncharge_locked(c, val);
        spin_unlock(&c->lock);
    }
```

从当前counter开始，从下往上逐层减少使用量，其中调用了res_counter_uncharge_locked，这个函数的作用就是在加锁的情况下减少相应的counter的使用量。

有这些数据结构和函数，只需要在内核分配资源的时候，植入相应的charge函数，释放资源时，植入相应的uncharge函数，就能实现对资源的控制了。

介绍完resource counter，我们再来看memory子系统是利用resource counter实现对内存资源的管理的。

memory子系统定义了一个叫mem_cgroup的结构体来管理cgroup相关的内存使用信息，定义如下：

```
struct mem_cgroup {
    struct cgroup_subsys_state css;
    struct res_counter res;
    struct res_counter memsw;
    struct mem_cgroup_lru_info info;
    spinlock_t reclaim_param_lock;
    int prev_priority;
    int last_scanned_child;
    bool use_hierarchy;
```

```
    atomic_t   oom_lock;
    atomic_t   refcnt;
    unsigned int   swappiness;
    int     oom_kill_disable;
    bool        memsw_is_minimum;
    struct mutex thresholds_lock;
    struct mem_cgroup_thresholds thresholds;
    struct mem_cgroup_thresholds memsw_thresholds;
    struct list_head oom_notify;
    unsigned long  move_charge_at_immigrate;
    struct mem_cgroup_stat_cpu *stat;
};
```

跟其他子系统一样，mem_cgroup也包含了一个cgroup_subsys_state成员，便于task或cgroup获取mem_cgroup。

mem_cgroup中包含了两个res_counter成员，分别用于管理memory资源和memory+swap资源，如果memsw_is_minimum为true，则res.limit=memsw.limit，即当进程组使用的内存超过memory的限制时，不能通过swap来缓解。

use_hierarchy则用来标记资源控制和记录时是否是层次性的。

oom_kill_disable则表示是否使用oom-killer。

oom_notify指向一个oom notifier event fd链表。

另外memory子系统还定义了一个叫page_cgroup的结构体：

```
struct page_cgroup {
    unsigned long flags;
    struct mem_cgroup *mem_cgroup;
    struct page *page;
    struct list_head lru;     /* per cgroup LRU list */
};
```

此结构体可以看作是mem_map的一个扩展，每个page_cgroup都和所有的page关联，而其中的mem_cgroup成员，则将page与特定的mem_cgroup关联起来。

我们知道在linux系统中，page结构体是用来管理物理页框的，一个物理页框对应一个page结构体，而每个进程中的task_struct中都有一个mm_struct来管理进程的内存信息。每个mm_struct知道它属于的进程，进而知道所属的mem_cgroup，而每个page都知道它属于的page_cgroup，进而也知道所属的mem_cgroup，而内存使用量的计算是按cgroup为单位的，这样以来，内存资源的管理就可以实现了。

memory子系统既然是通过resource counter实现的，那肯定会在内存分配给进程时进行charge操作的。下面我们就来看一下这些charge操作：

1.page fault发生时，有两种情况内核需要给进程分配新的页框。一种是进程请求调页（demand paging），另一种是copy on write。内核在handle_pte_fault中进行处理。其中，do_linear_fault处理pte不存在且页面线性映射了文件的情况，do_anonymous_page处理pte不存在且页面没有映射文件的情况，do_nonlinear_fault处理pte存在且页面非线性映射文件的情况，do_wp_page则处理copy on write的情况。其中do_linear_fault和do_nonlinear_fault都会调用__do_fault来处理。Memory子系统则__do_fault、do_anonymous_page、do_wp_page植入mem_cgroup_newpage_charge来进行charge操作。

2.内核在handle_pte_fault中进行处理时，还有一种情况是pte存在且页又没有映射文件。这种情况说明页面之前在内存中，但是后面被换出到swap空间了。内核用do_swap_page函数处理这种情况，memory子系统在do_swap_page加入了mem_cgroup_try_charge_swapin函数进行charge。mem_cgroup_try_charge_swapin是处理页面换入时的charge的，当执行swapoff系统调用（关掉swap空间），内核也会执行页面换入操作，因此mem_cgroup_try_charge_swapin也被植入到了相应的函数中。

3.当内核将page加入到page cache中时，也需要进行charge操作，mem_cgroup_cache_charge函数正是处理这种情况，它被植入到系统处理page cache的add_to_page_cache_locked函数中。

4.最后mem_cgroup_prepare_migration是用于处理内存迁移中的charge操作。

除了charge操作，memory子系统还需要处理相应的uncharge操作。下面我们来看一下uncharge操作：

1.mem_cgroup_uncharge_page用于当匿名页完全unmaped的时候。但是如果该page是swap cache的话，uncharge操作延迟到mem_cgroup_uncharge_swapcache被调用时执行。

2.mem_cgroup_uncharge_cache_page用于page cache从radix-tree删除的时候。但是如果该page是swap cache的话，uncharge操作延迟到mem_cgroup_uncharge_swapcache被调用时执行。

3.mem_cgroup_uncharge_swapcache用于swap cache从radix-tree删除的时候。Charge的资源会被算到swap_cgroup，如果mem+swap controller被禁用了，就不需要这样做了。

4.mem_cgroup_uncharge_swap用于swap_entry的引用数减到0的时候。这个函数主要在mem+swap controller可用的情况下使用的。

5.mem_cgroup_end_migration用于内存迁移结束时相关的uncharge操作。

Charge函数最终都是通过调用__mem_cgroup_try_charge来实现的。在__mem_cgroup_try_charge函数中，调用res_counter_charge(&mem->res, csize, &fail_res)对memory进行charge，调用res_counter_charge(&mem->memsw, csize, &fail_res)对memory+swap进行charge。

Uncharge函数最终都是通过调用__do_uncharge来实现的。在__do_uncharge中，分别调用res_counter_uncharge(&mem->res,PAGE_SIZE)和res_counter_uncharge(&mem->memsw, PAGE_SIZE)来uncharge memory和memory+swap。

跟其他子系统一样，memory子系统也实现了一个cgroup_subsys。

```
struct cgroup_subsys mem_cgroup_subsys = {
    .name = "memory",
    .subsys_id = mem_cgroup_subsys_id,
    .create = mem_cgroup_create,
    .pre_destroy = mem_cgroup_pre_destroy,
    .destroy = mem_cgroup_destroy,
    .populate = mem_cgroup_populate,
    .can_attach = mem_cgroup_can_attach,
    .cancel_attach = mem_cgroup_cancel_attach,
    .attach = mem_cgroup_move_task,
    .early_init = 0,
    .use_id = 1,
};
```

Memory子系统中重要的文件有
memsw.limit_in_bytes

```
    {
        .name = "memsw.limit_in_bytes",
        .private = MEMFILE_PRIVATE(_MEMSWAP, RES_LIMIT),
        .write_string = mem_cgroup_write,
        .read_u64 = mem_cgroup_read,
    },
```

这个文件用于设定memory+swap上限值。
Limit_in_bytes

```
    {
        .name = "limit_in_bytes",
        .private = MEMFILE_PRIVATE(_MEM, RES_LIMIT),
        .write_string = mem_cgroup_write,
        .read_u64 = mem_cgroup_read,
    },
```

这个文件用于设定memory上限值。

blkio 子系统

块 I/O（blkio）子系统控制并监控 cgroup 中的任务对块设备的 I/O 访问。在部分控制文件中写入值可限制访问或者带宽，且从这些控制文件中读取值可提供 I/O 操作信息。对文件 blkio.weight 写入相应值可以指定 cgroup 默认可用访问块 I/O 的相对比例（加权），范围在 100 到 1000。 这个跟 cpu 子系统里面的 cpu.shares 文件类似，一个是控制进程占有的 IO 时间，一个是控制进程占有的 cpu 时间。此文件的定义：

```
    {
        .name = "weight",
        .read_u64 = blkiocg_weight_read,
        .write_u64 = blkiocg_weight_write,
    }
```

blkio 子系统定义了一个叫 blkio_cgroup 的结构来存储一个 cgroup 的 block IO 信息。

```
struct blkio_cgroup {
    struct cgroup_subsys_state css;
    unsigned int weight;
    spinlock_t lock;
    struct hlist_head blkg_list;
    struct list_head policy_list; /* list of blkio_policy_node */
};
```

更其他子系统一样，内嵌了一个 cgroup_subsys_state 成员。
weight 字段存储的就是 blkio.weight 的值，用于控制此 cgroup 中的进程可以占有的 IO 时间。
在 blkiocg_weight_read 和 blkiocg_weight_write 中分别实现对 weight 字段读写操作。
blkio 是通过 CFQ IO 调度器实现的。IO 调度器的作用是处理进程的 IO 请求，然后将其交

给相应的块设备处理。IO 调度器一般采用所谓的电梯算法，而 CFQ 就是其中一种。CFQ 的核心思想在于，每个进程有自己的 IO 请求队列，各个进程的队列之间则按时间片轮转来处理，以保证每个进程都能公平的获得 IO 带宽。CFQ 现在是 Linux 内核默认的 IO 调度算法。Blkio 子系统就是通过 CFQ 组调度实现的，这点跟 cpu 子系统是通过 CFS 组调度实现类似。CFQ 很复杂，代码很多，这里就不去分析了。

freezer 子系统

freezer 子系统用于挂起和恢复 cgroup 中的进程。freezer 有一个控制文件：freezer.state，将 FROZEN 写入该文件，可以将 cgroup 中的进程挂起，将 THAWED 写入该文件，可以将已挂起的进程恢复。该文件可能读出的值有三种，其中两种就是前面已提到的 FROZEN 和 THAWED，分别代表进程已挂起和已恢复（正常运行），还有一种可能的值为 FREEZING，显示该值表示该 cgroup 中有些进程现在不能被 frozen。当这些不能被 frozen 的进程从该 cgroup 中消失的时候，FREEZING 会变成 FROZEN，或者手动将 FROZEN 或 THAWED 写入一次。

Freezer 子系统用来管理 cgroup 状态的数据结构：

```
struct freezer {
    struct cgroup_subsys_state css;
    enum freezer_state state;
    spinlock_t lock; /* protects _writes_ to state */
    };
```

其中内嵌一个 cgroup_subsys_state，便于从 cgroup 或 task 获得 freezer 结构，另一个字段存储 cgroup 当前的状态。

Freezer 子系统是通过对 freezer.state 文件进行写入来控制进程的，那我们就从这个文件的 cftype 定义出发。

```
static struct cftype files[] = {
    {
        .name = "state",
        .read_seq_string = freezer_read,
        .write_string = freezer_write,
    },
        };
```

从文件读取是 freezer_read 实现的，该函数比较简单，主要就是从 freezer 结构体从读出状态，但是对 FREEZING 状态做了特殊处理：

```
    state = freezer->state;
    if (state == CGROUP_FREEZING) {
        /* We change from FREEZING to FROZEN lazily if the cgroup was
         * only partially frozen when we exitted write. */
        update_freezer_state(cgroup, freezer);
        state = freezer->state;
        }
```

如果是 FREEZING 状态，则需要更新状态（因为之前不能 frozen 的进程可能已经不在了）。我们来看 update_freezer_state：

```
    cgroup_iter_start(cgroup, &it);
    while ((task = cgroup_iter_next(cgroup, &it))) {
```

```
        ntotal++;
        if (is_task_frozen_enough(task))
            nfrozen++;
    }

    /*
     * Transition to FROZEN when no new tasks can be added ensures
     * that we never exist in the FROZEN state while there are unfrozen
     * tasks.
     */
    if (nfrozen == ntotal)
        freezer->state = CGROUP_FROZEN;
    else if (nfrozen > 0)
        freezer->state = CGROUP_FREEZING;
    else
        freezer->state = CGROUP_THAWED;
        cgroup_iter_end(cgroup, &it);
```

这里对该 cgroup 所有的进程迭代了一遍，分别统计进程数和已经 frozen 的进程数，然后根据统计结果改变状态。

下面我们来看对 freezer.state 写入的情况，该情况由 freezer_write 来处理，该函数中从写入值获取目标状态，然后调用 freezer_change_state(cgroup, goal_state) 来完成操作。在 freezer_change_state 中，根据 goal_state 分别调用不同的实现函数：

```
    switch (goal_state) {
    case CGROUP_THAWED:
        unfreeze_cgroup(cgroup, freezer);
        break;
    case CGROUP_FROZEN:
        retval = try_to_freeze_cgroup(cgroup, freezer);
        break;
    default:
        BUG();
        }
```

我们先来看 frozen 的情况，该情况由 try_to_freeze_cgroup 来处理，该函数中有：

```
    freezer->state = CGROUP_FREEZING;
    cgroup_iter_start(cgroup, &it);
    while ((task = cgroup_iter_next(cgroup, &it))) {
        if (!freeze_task(task, true))
            continue;
        if (is_task_frozen_enough(task))
            continue;
        if (!freezing(task) && !freezer_should_skip(task))
            num_cant_freeze_now++;
    }
    cgroup_iter_end(cgroup, &it);
```

```
    return num_cant_freeze_now ? -EBUSY : 0;
```

首先将当前状态设成 *CGROUP_FREEZING*，然后对 cgroup 中的进程进行迭代，while 循环中对进程进行 freeze 操作，如果成功直接进行下一次迭代，如果不成功则进行进一步的判断，如果是进程已经 frozen 了，那也直接进行下一次迭代，如果不是，则进行计数。最后根据计数结果进行返回，如果所有进程都顺利 frozen，则返回 0，否则返回-EBUSY 表示有进程不能被 frozen。

下面我们来看 free_task 这个函数，在这个函数中对 task 进行 freeze 操作。

```
if (!freezing(p)) {
    rmb();
    if (frozen(p))
        return false;

    if (!sig_only || should_send_signal(p))
        set_freeze_flag(p);
    else
        return false;
}

if (should_send_signal(p)) {
    if (!signal_pending(p))
        fake_signal_wake_up(p);
} else if (sig_only) {
    return false;
} else {
    wake_up_state(p, TASK_INTERRUPTIBLE);
}

    return true;
```

首先检查进程是不是已经被标记为正在 freezing，如果不是再做判断。如果进程已经被 frozen，则返回 false。如果进程不是 sig_only 的或者可以发送信号（即进程无 PF_FREEZER_NOSIG 标记），则设置进程的 TIF_FREEZE 标记。

然后根据进程是否有 PF_FREEZER_NOSIG 标记进行进一步处理，如果无这个标记，则给进程发送一个信号，唤醒进程，让进程处理 TIF_FREEZE，即进行 freeze 操作，如果有这个标记，则如果进程是 sig_only 的，返回 false（即不能完成 free 操作），否则直接唤醒进程去处理 TIF_FREEZE。

总结一下，对于我们这个 freezer 子系统的调用来说，sig_only=true，那么能成功的执行过程就是 set_freeze_flag(p)->fake_signal_wake_up(p)。

下面我们来看 thaw 进程的情况，该情况由 unfreeze_cgroup 处理，在 unfreeze_cgroup 中有

```
cgroup_iter_start(cgroup, &it);
while ((task = cgroup_iter_next(cgroup, &it))) {
    thaw_process(task);
}
```

```
    cgroup_iter_end(cgroup, &it);

        freezer->state = CGROUP_THAWED;
```

对该 cgroup 中所有的进程调用 thaw_process，我们来看 thaw_process。该函数中有：

```
    if (__thaw_process(p) == 1) {
        task_unlock(p);
        wake_up_process(p);
        return 1;
        }
```

其中__thaw_process 中

```
    if (frozen(p)) {
        p->flags &= ~PF_FROZEN;
        return 1;
    }
        clear_freeze_flag(p);
```

如果进程已经 frozen，则清掉其 frozen 标记，如果不是的话，说明进程已经设置了 TIF_FREEZE，但还没有 frozen，所以只需要清掉 TIF_FREEZE 即可。

回到 thaw_process 中，清掉了相关标记后，只需要唤醒进程，然后内核会自动处理。

最后，我们再来看看 freezer 子系统结构体的定义：

```
struct cgroup_subsys freezer_subsys = {
    .name       = "freezer",
    .create     = freezer_create,
    .destroy    = freezer_destroy,
    .populate   = freezer_populate,
    .subsys_id = freezer_subsys_id,
    .can_attach     = freezer_can_attach,
    .attach     = NULL,
    .fork       = freezer_fork,
    .exit       = NULL,
    };
```

这里说一下 can_attach，can_attach 是在一个进程加入到一个 cgroup 之前调用的，检查是否可以 attach，freezer_can_attach 中对 cgroup 当前的状态做了检查，如果是 frozen 就返回错误，这说明不能将一个进程加入到一个 frozen 的 cgroup。

devices 子系统

使用devices 子系统可以允许或者拒绝cgroup中的进程访问设备。devices子系统有三个控制文件：devices.allow,devices.deny,devices.list。devices.allow用于指定cgroup中的进程可以访问的设备,devices.deny用于指定cgroup中的进程不能访问的设备，devices.list用于报告cgroup中的进程访问的设备。devices.allow文件中包含若干条目，每个条目有四个字段：type、major、minor 和 access。type、major 和 minor

字段中使用的值对应 Linux 分配的设备。
type指定设备类型：
a - 应用所有设备，可以是字符设备，也可以是块设备
b- 指定块设备
c - 指定字符设备
major和minor指定设备的主次设备号。
access 则指定相应的权限：
r - 允许任务从指定设备中读取
w - 允许任务写入指定设备
m - 允许任务生成还不存在的设备文件
devices 子系统是通过提供 device whilelist 来实现的。与其他子系统一样，devices 子系统也有一个内嵌了 cgroup_subsystem_state 的结构来管理资源。在 devices 子系统中，这个结构是：

```
struct dev_cgroup {
    struct cgroup_subsys_state css;
    struct list_head whitelist;
    };
```

这个结构体除了通用的 cgroup_subsystem_state 之外，就只有一个链表指针，而这个链表指针指向了该 cgroup 中的进程可以访问的 devices whilelist。
下面我们来看一下 devices 子系统如何管理 whilelist。在 devices 子系统中，定义了一个叫 dev_whitelist_item 的结构来管理可以访问的 device,对应于 devices.allow 中的一个条目。这个结构体的定义如下：

```
struct dev_whitelist_item {
    u32 major, minor;
    short type;
    short access;
    struct list_head list;
    struct rcu_head rcu;
    };
```

major，minor用于指定设备的主次设备号，type用于指定设备类型，type取值可以是：

```
#define DEV_BLOCK 1
#define DEV_CHAR  2
#define DEV_ALL   4
```

对应于之前 devices.allow 文件中三种情况。
access 用于相应的访问权限,access 取值可以是:

```
#define ACC_MKNOD 1
#define ACC_READ  2
#define ACC_WRITE 4
```

也和之前 devices.allow 文件中的情况对应。
List 字段用于将该结构体连到相应的 dev_cgroup 中 whitelist 指向的链表。
通过以上数据结构，devices 子系统就能管理一个 cgroup 的进程可以访问的 devices 了。
光有数据结构还不行，还要有具体实现才行。devices 子系统通过实现两个函数供内核调用来实现控制 cgroup 中的进程能够访问的 devices。首先我们来第一个函数：

```
int devcgroup_inode_permission(struct inode *inode, int mask)
```

```
{
    struct dev_cgroup *dev_cgroup;
    struct dev_whitelist_item *wh;

    dev_t device = inode->i_rdev;
    if (!device)
        return 0;
    if (!S_ISBLK(inode->i_mode) && !S_ISCHR(inode->i_mode))
        return 0;

    rcu_read_lock();

    dev_cgroup = task_devcgroup(current);

    list_for_each_entry_rcu(wh, &dev_cgroup->whitelist, list) {
        if (wh->type & DEV_ALL)
            goto found;
        if ((wh->type & DEV_BLOCK) && !S_ISBLK(inode->i_mode))
            continue;
        if ((wh->type & DEV_CHAR) && !S_ISCHR(inode->i_mode))
            continue;
        if (wh->major != ~0 && wh->major != imajor(inode))
            continue;
        if (wh->minor != ~0 && wh->minor != iminor(inode))
            continue;

        if ((mask & MAY_WRITE) && !(wh->access & ACC_WRITE))
            continue;
        if ((mask & MAY_READ) && !(wh->access & ACC_READ))
            continue;
found:
        rcu_read_unlock();
        return 0;
    }

    rcu_read_unlock();

    return -EPERM;
}
```

我们来简单分析一下这个函数，首先如果该 inode 对应的不是 devices，直接返回 0，如果既不是块设备也不是字符设备，也返回 0，因为 devices 只控制块设备和字符设备的访问，其他情况不管。接着获得当前进程的 dev_cgroup，然后在 dev_cgroup 中 whitelist 指针的链表中查找，如果找到对应设备而且 mask 指定的权限和设备的权限一致就返回 0，如果没有找到就返回错误。

这个函数是针对 inode 节点存在的情况，通过对比权限来控制 cgroup 中的进程能够访问的 devices。还有一个情况是 inode 不存在，在这种情况下，一个进程要访问一个设备就必须通过 mknod 建立相应的设备文件。为了达到对这种情况的控制，devices 子系统导出了第二个函数：

```
int devcgroup_inode_mknod(int mode, dev_t dev)
{
    struct dev_cgroup *dev_cgroup;
    struct dev_whitelist_item *wh;

    if (!S_ISBLK(mode) && !S_ISCHR(mode))
        return 0;

    rcu_read_lock();

    dev_cgroup = task_devcgroup(current);

    list_for_each_entry_rcu(wh, &dev_cgroup->whitelist, list) {
        if (wh->type & DEV_ALL)
            goto found;
        if ((wh->type & DEV_BLOCK) && !S_ISBLK(mode))
            continue;
        if ((wh->type & DEV_CHAR) && !S_ISCHR(mode))
            continue;
        if (wh->major != ~0 && wh->major != MAJOR(dev))
            continue;
        if (wh->minor != ~0 && wh->minor != MINOR(dev))
            continue;

        if (!(wh->access & ACC_MKNOD))
            continue;
found:
        rcu_read_unlock();
        return 0;
    }

    rcu_read_unlock();

    return -EPERM;
}
```

这个函数的实现跟第一个函数类似，这里就不赘述了。

下面我们再来看一下 devices 子系统本身的一些东西。跟其他子系统一样，devices 同样实现了一个 cgroup_subsys：

```
struct cgroup_subsys devices_subsys = {
    .name = "devices",
```

```
    .can_attach = devcgroup_can_attach,
    .create = devcgroup_create,
    .destroy = devcgroup_destroy,
    .populate = devcgroup_populate,
    .subsys_id = devices_subsys_id,
};
```

devices 相应的三个控制文件：

```
static struct cftype dev_cgroup_files[] = {
    {
        .name = "allow",
        .write_string  = devcgroup_access_write,
        .private = DEVCG_ALLOW,
    },
    {
        .name = "deny",
        .write_string = devcgroup_access_write,
        .private = DEVCG_DENY,
    },
    {
        .name = "list",
        .read_seq_string = devcgroup_seq_read,
        .private = DEVCG_LIST,
    },
};
```

其中 allow 和 deny 都是通过 devcgroup_access_write 实现的，只是通过 private 字段区分，因为二者的实现逻辑有相同的地方。devcgroup_access_write 最终通过调用 devcgroup_update_access 来实现。在 devcgroup_update_access 根据写入的内容构造一个 dev_whitelist_item ，然后根据文件类型做不同的处理：

```
    switch (filetype) {
    case DEVCG_ALLOW:
        if (!parent_has_perm(devcgroup, &wh))
            return -EPERM;
        return dev_whitelist_add(devcgroup, &wh);
    case DEVCG_DENY:
        dev_whitelist_rm(devcgroup, &wh);
        break;
    default:
        return -EINVAL;
    }
```

allow 的话，就将 item 加入 whitelist，deny 的话，就将 item 从 whitelist 中删去。

ns 子系统

ns 子系统是一个比较特殊的子系统。特殊在哪儿呢，首先 ns 子系统没有自己的控制

文件，其次 ns 子系统没有属于自己的状态信息，这点从 ns 子系统的 ns_cgroup 的定义可以看出：

```
struct ns_cgroup {
    struct cgroup_subsys_state css;
};
```

它只有一个 cgroup_subsys_state 成员。

最后 ns 子系统的实现也比较简单，只是提供了一个 ns_cgroup_clone 函数，在 copy_process 和 unshare_nsproxy_namespaces 被调用。而 ns_cgroup_clone 函数本身的实现也很简单，只是在当前的 cgroup 下创建了一个子 cgroup，该子 cgroup 完全 clone 了当前 cgroup 的信息，然后将当前的进程移到新建立的 cgroup 中。

这样看来，好像 ns 子系统没什么意义，其实不然。要想了解 ns 子系统的意义，就要分析一下 ns_cgroup_clone 被调用的时机了。我们来看 copy_process 中的代码：

```
    if (current->nsproxy != p->nsproxy) {
        retval = ns_cgroup_clone(p, pid);
        if (retval)
            goto bad_fork_free_pid;
    }
```

copy_process 是在 do_fork 中被调用的，作用在于为子进程复制父进程的相关信息。这段意思就是当前进程（即父进程）和子进程的命名空间不同时，调用 ns_cgroup_clone。这样以来，ns 子系统的作用就清楚了，ns 子系统实际上是提供了一种同命名空间的进程聚类的机制。具有相同命名空间的进程会在相同 cgroup 中。

那什么时候，父进程 fork 出的子进程会拥有不同的命名空间呢，这就设计到了 Linux 的命名空间的机制了，在这里就不详细讲了。简单说来就是，在调用 fork 时，加入了特殊 flag（比如 NEWPID,NEWNS）时，内核会为子进程创建不同的命令空间。

除了这种情况外，ns_cgroup_clone 在 unshare_nsproxy_namespaces 用到了。unshare_nsproxy_namespaces 函数被 sys_unshare 调用，实际上是对 unshare 系统调用的实现。当指定相应标记时，unshare 系统调用会为调用的进程创建不同的命名空间，因此调用 ns_cgroup_clone 为其创建新的 cgroup。