Linux的IO调度

Linux的IO调度

Hi，我是Zorro。这是我的博客地址，我会不定期在这里更新文章，欢迎来一起探讨，
我的微博地址，有兴趣可以来关注我呦。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

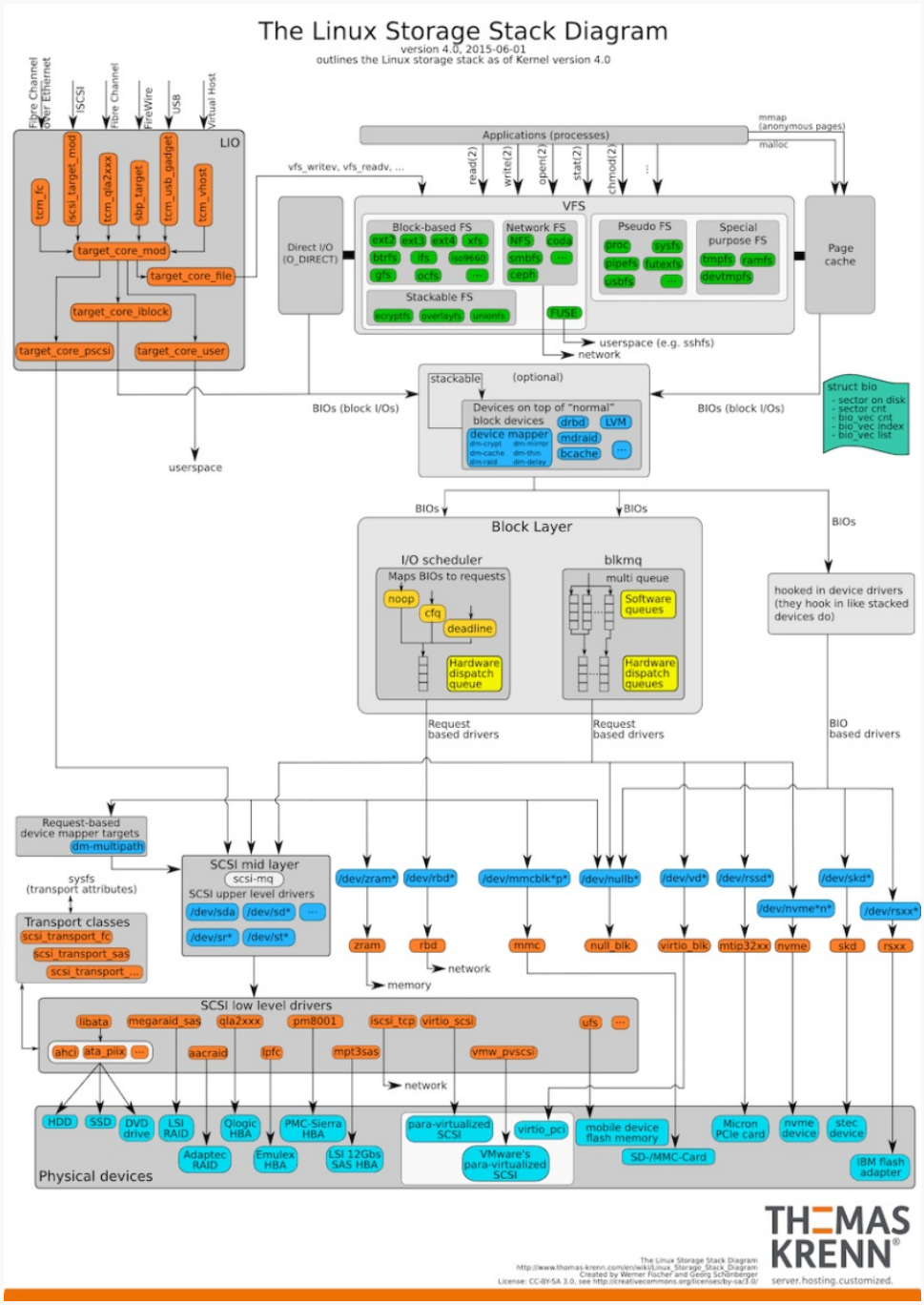
今天我们来谈谈：

Linux的IO调度

IO调度发生在Linux内核的IO调度层。这个层次是针对Linux的整体IO层次体系来说的。从read()或者write()系统调用的角度来说，Linux整体IO体系可以分为七层，它们分别是：

1. VFS层：虚拟文件系统层。由于内核要跟多种文件系统打交道，而每一种文件系统所实现的数据结构和相关方法都可能不尽相同，所以，内核抽象了这一层，专门用来适配各种文件系统，并对外提供统一操作接口。
2. 文件系统层：不同的文件系统实现自己的操作过程，提供自己特有的特征，具体不多说了，大家愿意的话自己去看代码即可。
3. 页缓存层：负责真对page的缓存。
4. 通用块层：由于绝大多数情况的io操作是跟块设备打交道，所以Linux在此提供了一个类似vfs层的块设备操作抽象层。下层对接各种不同属性的块设备，对上提供统一的Block IO请求标准。
5. IO调度层：因为绝大多数的块设备都是类似磁盘这样的设备，所以有必要根据这类设备的特点以及应用的不同特点来设置一些不同的调度算法和队列。以便在不同的应用环境下有针对性的提高磁盘的读写效率，这里就是大名鼎鼎的Linux电梯所起作用的地方。针对机械硬盘的各种调度方法就是在这实现的。
6. 块设备驱动层：驱动层对外提供相对比较高级的设备操作接口，往往是C语言的，而下层对接设备本身的操作方法和规范。
7. 块设备层：这层就是具体的物理设备了，定义了各种真对设备操作方法和规范。

有一个已经整理好的Linux IO结构图，非常经典，一图胜千言：



我们今天要研究的内容主要在IO调度这一层。它要解决的核心问题是，如何提高块设备IO的整体性能？这一层也主要是针对机械硬盘结构而设计的。众所周知，机械硬盘的存储介质是磁盘，磁头在盘片上移动进行磁道寻址，行为类似播放一张唱片。这种结构的特点是，顺序访问时吞吐量较高，但是如果一旦对盘片有随机访问，那么大量的时间都会浪费在磁头的移动上，这时候就会导致每次IO的响应时间变长，极大的降低IO的响应速度。磁头在盘片上寻道的操作，类似电梯调度，如果在寻道的过程中，能把顺序路过的相关磁道的数据请求都“顺便”处理掉，那么就可以在比较小影响响应速度的前提下，提高整体IO的吞吐量。这就是我们问什么要设计IO调度算法的原因。在最开始的时期，Linux把这个算法命名为Linux电梯算法。目前在内核中默认开启了三种算法，其实严格算应该是两种，因为第一种叫做noop，就是空操作调度算法，也就是没有任何调度操作，并不对io请求进行排序，仅仅做适当的io合并的一个fifo队列。

目前内核中默认的调度算法应该是cfq，叫做完全公平队列调度。这个调度算法人如其名，它试图给所有进程提供一个完全公平的IO操作环境。它为每个进程创建一个同步IO调度队列，并默认以时间片和请求数限定的方式分配IO资源，以此保证每个进程的IO资源占用是公平的，cfq还实现了针对进程级别的优先级调度，这个我们后面会详细解释。

查看和修改IO调度算法的方法是：

```
[zorro@zorrozou-pc0 ~]$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
[zorro@zorrozou-pc0 ~]$ echo cfq > /sys/block/sda/queue/scheduler
```

cfq是通用服务器比较好的IO调度算法选择，对桌面用户也是比较好的选择。但是对于很多IO压力较大的场景就并不是很适应，尤其是IO压力集中在某些进程上的场景。因为这种场景我们需要更多的满足某个或者某几个进程的IO响应速度，而不是让所有的进程公平的使用IO，比如数据库应用。

deadline调度（最终期限调度）就是更适合上述场景的解决方案。deadline实现了四个队列，其中两个分别处理正常read和write，按扇区号排序，进行正常io的合并处理以提高吞吐量.因为IO请求可能会集中在某些磁盘位置，这样会导致新来的请求一直被合并，可能会有其他磁盘位置的io请求被饿死。因此实现了另外两个处理超时read和write的队列，按请求创建时间排序，如果有超时的请求出现，就放进这两个队列，调度算法保证超时（达到最终期限时间）的队列中的请求会优先被处理，防止请求被饿死。

不久前，内核还是默认标配四种算法，还有一种叫做as的算法（Anticipatory scheduler），预测调度算法。一个高大上的名字，搞得我一度认为Linux内核都会算命了。结果发现，无非是在基于deadline算法做io调度的之前等一小会时间，如果这段时间内有可以合并的io请求到来，就可以合并处理，提高deadline调度的在顺序读写情况下的数据吞吐量。其实这根本不是啥预测，我觉得不如叫撞大运调度算法，当然这种策略在某些特定场景差效果不错。但是在大多数场景下，这个调度不仅没有提高吞吐量，还降低了响应速度，所以内核干脆把它从默认配置里删除了。毕竟Linux的宗旨是实用，而我们也就不再这个调度算法上多费口舌了。

CFQ完全公平队列

CFQ是内核默认选择的IO调度队列，它在桌面应用场景以及大多数常见应用场景下都是很好的选择。如何实现一个所谓的完全公平队列（Completely Fair Queueing）？首先我们要理解所谓的公平是对谁的公平？从操作系统的角度来说，产生操作行为的主体都是进程，所以这里的公平是针对每个进程而言的，我们要试图让进程可以公平的占用IO资源。那么如何让进程公平的占用IO资源？我们需要先理解什么是IO资源。当我们衡量一个IO资源的时候，一般喜欢用的是两个单位，一个是数据读写的带宽，另一个是数据读写的IOPS。带宽就是以时间为单位的读写数据量，比如，100Mbyte/s。而IOPS是以时间为单位的读写次数。在不同的读写情境下，这两个单位的表现可能不一样，但是可以确定的是，两个单位的任何一个达到了性能上限，都会成为IO的瓶颈。从机械硬盘的结构考虑，如果读写是顺序读写，那么IO的表现是可以通过比较少的IOPS达到较大的带宽，因为可以合并很多IO，也可以通过预读等方式加速数据读取效率。当IO的表现是偏向于随机读写的时候，那么IOPS就会变得更大，IO的请求的合并可能性下降，当每次io请求数据越少的时候，带宽表现就会越低。从这里我们可以理解，针对进程的IO资源的主要表现形式有两个，进程在单位时间内提交的IO请求个数和进程占用IO的带宽。其实无论哪个，都是跟进程分配的IO处理时间长度紧密相关的。

有时业务可以在较少IOPS的情况下占用较大带宽，另外一些则可能在较大IOPS的情况下占用较少带宽，所以对进程占用IO的时间进行调度才是相对最公平的。即，我不管你是IOPS高还是带宽占用高，到了时间咱就换下一个进程处理，你爱咋样咋样。所以，cfq就是试图给所有进程分配等同的块设备使用的时间片，进程在时间片内，可以将产生的IO请求提交给块设备进行处理，时间片结束，进程的请求将排进它自己的队列，等待下次调度的时候进行处理。这就是cfq的基本原理。

当然，现实生活中不可能有真正的“公平”，常见的应用场景下，我们很肯能需要人为的对进程的IO占用进行人为指定优先级，这就像对进程的CPU占用设置优先级的概念一样。所以，除了针对时间片进行公平队列调度外，cfq还提供了优先级支持。每个进程都可以设置一个IO优先级，cfq会根据这个优先级的设置情况作为调度时的重要参考因素。优先级首先分成三大类：RT、BE、IDLE，它们分别是实时（Real Time）、最佳效果（Best Try）和闲置（Idle）三个类别，对每个类别的IO，cfq都使用不同的策略进行处理。另外，RT和BE类别中，分别又再划分了8个子优先级实现更细节的QOS需求，而IDLE只有一个子优先级。

另外，我们都知道内核默认对存储的读写都是经过缓存（buffer/cache）的，在这种情况下，cfq是无法区分当前处理的请求是来自哪一个进程的。只有在进程使用同步方式（sync read或者sync wirte）或者直接IO（Direct IO）方式进行读写的时候，cfq才能区分出IO请求来自哪个进程。所以，除了针对每个进程实现的IO队列以外，还实现了一个公共的队列用来处理异步请求。

当前内核已经实现了针对IO资源的cgroup资源隔离，所以在以上体系的基础上，cfq也实现了针对cgroup的调度支持。关于cgroup的blkio功能的描述，请看我之前的文章Cgroup – Linux的IO资源隔离。总的来说，cfq用了一系列的数据结构实现了以上所有复杂功能的支持，大家可以通过源代码看到其相关实现，文件在源代码目录下的block/cfq-iosched.c。

CFQ设计原理

在此，我们对整体数据结构做一个简要描述：首先，cfq通过一个叫做cfq_data的数据结构维护了整个调度器流程。在一个支持了cgroup功能的cfq中，全部进程被分成了若干个contral group进行管理。每个cgroup在cfq中都有一个cfq_group的结构进行描述，所有的cgroup都被作为一个调度对象放进一个红黑树中，并以vdisktime为key进行排序。vdisktime这个时间纪录的是当前cgroup所占用的io时间，每次对cgroup进行调度时，总是通过红黑树选择当前vdisktime时间最少的cgroup进行处理，以保证所有cgroups之间的IO资源占用“公平”。当然我们知道，cgroup是可以对blkio进行资源比例分配的，其作用原理就是，分配比例大的cgroup占用vdisktime时间增长较慢，分配比例小的vdisktime时间增长较快，快慢与分配比例成正比。这样就做到了不同的cgroup分配的IO比例不一样，并且在cfq的角度看来依然是“公平“的。

选择好了需要处理的cgroup（cfq_group）之后，调度器需要决策选择下一步的service_tree。service_tree这个数据结构对应的都是一系列的红黑树，主要目的是用来实现请求优先级分类的，就是RT、BE、IDLE的分类。每一个cfq_group都维护了7个service_trees，其定义如下：

```
struct cfq_rb_root service_trees[2][3];
struct cfq_rb_root service_tree_idle;
```

其中service_tree_idle就是用来给IDLE类型的请求进行排队用的红黑树。而上面二维数组，首先第一个维度针对RT和BE分别各实现了一个数组，每一个数组中都维护了三个红黑树，分别对应三种不同子类型的请求，分别是：SYNC、SYNC_NOIDLE以及ASYNC。我们可以认为SYNC相当于SYNC_IDLE并与SYNC_NOIDLE对应。idling是cfq在设计上为了尽量合并连续的IO请求以达到提高吞吐量的目的而加入的机制，我们可以理解为是一种“空转”等待机制。空转是指，当一个队列处理一个请求结束后，会在发生调度之前空等一小会时间，如果下一个请求到来，则可以减少磁头寻址，继续处理顺序的IO请求。为了实现这个功能，cfq在service_tree这层数据结构这实现了SYNC队列，如果请求是同步顺序请求，就入队这个service tree，如果请求是同步随机请求，则入队SYNC_NOIDLE队列，以判断下一个请求是否是顺序请求。所有的异步写操作请求将入队ASYNC的service tree，并且针对这个队列没有空转等待机制。此外，cfq还对SSD这样的硬盘有特殊调整，当cfq发现存储设备是一个ssd硬盘这样的队列深度更大的设备时，所有针对单独队列的空转都将不生效，所有的IO请求都将入队SYNC_NOIDLE这个service tree。

每一个service tree都对应了若干个cfq_queue队列，每个cfq_queue队列对应一个进程，这个我们后续再详细说明。

cfq_group还维护了一个在cgroup内部所有进程公用的异步IO请求队列，其结构如下：

```
struct cfq_queue *async_cfqq[2][IOPRIO_BE_NR];
struct cfq_queue *async_idle_cfqq;
```

异步请求也分成了RT、BE、IDLE这三类进行处理，每一类对应一个cfq_queue进行排队。BE和RT也实现了优先级的支持，每一个类型有IOPRIO_BE_NR这么多个优先级，这个值定义为8，数组下标为0-7。我们目前分析的内核代码版本为Linux 4.4，可以看出，从cfq的角度来说，已经可以实现异步IO的cgroup支持了，我们需要定义一下这里所谓异步IO的含义，它仅仅表示从内存的buffer/cache中的数据同步到硬盘的IO请求，而不是aio(man 7 aio)或者linux的native异步io以及libaio机制，实际上这些所谓的“异步”IO机制，在内核中都是同步实现的（本质上冯诺伊曼计算机没有真正的“异步”机制）。

我们在上面已经说明过，由于进程正常情况下都是将数据先写入buffer/cache，所以这种异步IO都是统一由cfq_group中的async请求队列处理的。那么为什么在上面的service_tree中还要实现和一个ASYNC的类型呢？这当然是为了支持区分进程的异步IO并使之可以“完全公平”做准备喽。实际上在最新的cgroup v2的blkio体系中，内核已经支持了针对buffer IO的cgroup限速支持，而以上这些可能容易混淆的一堆类型，都是在新的体系下需要用到的类型标记。新体系的复杂度更高了，功能也更加强大，但是大家先不要着急，正式的cgroup v2体系，在Linux 4.5发布的时候会正式跟大家见面。

我们继续选择service_tree的过程，三种优先级类型的service_tree的选择就是根据类型的优先级来做选择的，RT优先级最高，BE其次，IDLE最低。就是说，RT里有，就会一直处理RT，RT没了再处理BE。每个service_tree对应一个元素为cfq_queue排队的红黑树，而每个cfq_queue就是内核为进程（线程）创建的请求队列。每一个cfq_queue都会维护一个rb_key的变量，这个变量实际上就是这个队列的IO服务时间（service time）。这里还是通过红黑树找到service time时间最短的那个cfq_queue进行服务，以保证“完全公平”。

选择好了cfq_queue之后，就要开始处理这个队列里的IO请求了。这里的调度方式基本跟deadline类似。cfq_queue会对进入队列的每一个请求进行两次入队，一个放进fifo中，另一个放进按访问扇区顺序作为key的红黑树中。默认从红黑树中取请求进行处理，当请求的延时时间达到deadline时，就从红黑树中取等待时间最长的进行处理，以保证请求不被饿死。

这就是整个cfq的调度流程，当然其中还有很多细枝末节没有交代，比如合并处理以及顺序处理等等。

CFQ的参数调整

理解整个调度流程有助于我们决策如何调整cfq的相关参数。所有cfq的可调参数都可以在/sys/class/block/sda/queue/iosched/目录下找到，当然，在你的系统上，请将sda替换为相应的磁盘名称。我们来看一下都有什么：

```
[root@zorrozou-pc0 zorro]# echo cfq > /sys/block/sda/queue/scheduler
[root@zorrozou-pc0 zorro]# ls /sys/class/block/sda/queue/iosched/
back_seek_max  back_seek_penalty  fifo_expire_async  fifo_expire_sync  group_idle
low_latency  quantum  slice_async  slice_async_rq  slice_idle  slice_sync
target_latency
```

这些参数部分是跟机械硬盘磁头寻道方式有关的，如果其说明你看不懂，请先补充相关知识：

back_seek_max:磁头可以向后寻址的最大范围，默认值为16M。

back_seek_penalty:向后寻址的惩罚系数。这个值是跟向前寻址进行比较的。

以上两个是为了防止磁头寻道发生抖动而导致寻址过慢而设置的。基本思路是这样，一个io请求到来的时候，cfq会根据其寻址位置预估一下其磁头寻道成本。首先设置一个最大值back_seek_max，对于请求所访问的扇区号在磁头后方的请求，只要寻址范围没有超过这个值，cfq会像向前寻址的请求一样处理它。然后再设置一个评估成本的系数back_seek_penalty，相对于磁头向前寻址，向后寻址的距离为1/2(1/back_seek_penalty)时，cfq认为这两个请求寻址的代

价是相同。这两个参数实际上是cfq判断请求合并处理的条件限制，凡事复合这个条件的请求，都会尽量在本次请求处理的时候一起合并处理。

fifo_expire_async:设置异步请求的超时时间。同步请求和异步请求是区分不同队列处理的，cfq在调度的时候一般情况都会优先处理同步请求，之后再处理异步请求，除非异步请求符合上述合并处理的条件限制范围内。当本进程的队列被调度时，cfq会优先检查是否有异步请求超时，就是超过fifo_expire_async参数的限制。如果有，则优先发送一个超时的请求，其余请求仍然按照优先级以及扇区编号大小来处理。

fifo_expire_sync:这个参数跟上面的类似，区别是用来设置同步请求的超时时间。

slice_idle:参数设置了一个等待时间。这让cfq在切换cfq_queue或service tree的时候等待一段时间，目的是提高机械硬盘的吞吐量。一般情况下，来自同一个cfq_queue或者service tree的IO请求的寻址局部性更好，所以这样可以减少磁盘的寻址次数。这个值在机械硬盘上默认为非零。当然在固态硬盘或者硬RAID设备上设置这个值为非零会降低存储的效率，因为固态硬盘没有磁头寻址这个概念，所以在这样的设备上应该设置为0，关闭此功能。

group_idle:这个参数也跟上一个参数类似，区别是当cfq要切换cfq_group的时候会等待一段时间。在cgroup的场景下，如果我们沿用slice_idle的方式，那么空转等待可能会在cgroup组内每个进程的cfq_queue切换时发生。这样会如果这个进程一直有请求要处理的话，那么直到这个cgroup的配额被耗尽，同组中的其它进程也可能无法被调度到。这样会导致同组中的其它进程饿死而产生IO性能瓶颈。在这种情况下，我们可以将slice_idle = 0而group_idle = 8。这样空转等待就是以cgroup为单位进行的，而不是以cfq_queue的进程为单位进行，以防止上述问题产生。

low_latency:这个是用来开启或关闭cfq的低延时（low latency）模式的开关。当这个开关打开时，cfq将会根据target_latency的参数设置来对每一个进程的分片时间（slice time）进行重新计算。这将有利于对吞吐量的公平（默认是对时间片分配的公平）。关闭这个参数（设置为0）将忽略target_latency的值。这将使系统中的进程完全按照时间片方式进行IO资源分配。这个开关默认是打开的。

我们已经知道cfq设计上有“空转”（idling）这个概念，目的是为了可以让连续的读写操作尽可能多的合并处理，减少磁头的寻址操作以便增大吞吐量。如果有进程总是很快的进行顺序读写，那么它将因为cfq的空转等待命中率很高而导致其它需要处理IO的进程响应速度下降，如果另一个需要调度的进程不会发出大量顺序IO行为的话，系统中不同进程IO吞吐量的表现就会很不均衡。就比如，系统内存的cache中有很多脏页要写回时，桌面又要打开一个浏览器进行操作，这时脏页写回的后台行为就很可能会大量命中空转时间，而导致浏览器的小量IO一直等待，让用户感觉浏览器运行响应速度变慢。这个low_latency主要是对这种情况进行优化的选项，当其打开时，系统会根据target_latency的配置对因为命中空转而大量占用IO吞吐量的进程进行限制，以达到不同进程IO占用的吞吐量的相对均衡。这个开关比较合适在类似桌面应用的场景下打开。

target_latency:当low_latency的值为开启状态时，cfq将根据这个值重新计算每个进程分配的IO时间片长度。

quantum:这个参数用来设置每次从cfq_queue中处理多少个IO请求。在一个队列处理事件周期中，超过这个数字的IO请求将不会被处理。这个参数只对同步的请求有效。

slice_sync:当一个cfq_queue队列被调度处理时，它可以被分配的处理总时间是通过这个值来作为一个计算参数指定的。公式为：time_slice = slice_sync + (slice_sync/5 * (4 – prio))。这个参数对同步请求有效。

slice_async:这个值跟上一个类似，区别是对异步请求有效。

slice_async_rq:这个参数用来限制在一个slice的时间范围内，一个队列最多可以处理的异步请求个数。请求被处理的最大个数还跟相关进程被设置的io优先级有关。

CFQ的IOPS模式

我们已经知道，默认情况下cfq是以时间片方式支持的带优先级的调度来保证IO资源占用的公平。高优先级的进程将得到更多的时间片长度，而低优先级的进程时间片相对较小。当我们的存储是一个高速并且支持NCQ（原生指令队列）的设备的时候，我们最好可以让其可以从多个cfq队列中处理多路的请求，以便提升NCQ的利用率。此时使用时间片的分配方式分配资源就显得不合时宜了，因为基于时间片的分配，同一时刻最多能处理的请求队列只有一个。这时，我们需要切换cfq的模式为IOPS模式。切换方式很简单，就是将slice_idle=0即可。内核会自动检测你的存储设备是否支持NCQ，如果支持的话cfq会自动切换为IOPS模式。

另外，在默认的基于优先级的时间片方式下，我们可以使用ionice命令来调整进程的IO优先级。进程默认分配的IO优先级是根据进程的nice值计算而来的，计算方法可以在man ionice中看到，这里不再废话。

DEADLINE最终期限调度

deadline调度算法相对cfq要简单很多。其设计目标是，在保证请求按照设备扇区的顺序进行访问的同时，兼顾其它请求不被饿死，要在一个最终期限前被调度到。我们知道磁头对磁盘的寻道是可以进行顺序访问和随机访问的，因为寻道延时时间的关系，顺序访问时IO的吞吐量更大，随机访问的吞吐量小。如果我们想为一个机械硬盘进行吞吐量优化的话，那么就可以让调度器按照尽量复合顺序访问的IO请求进行排序，之后请求以这样的顺序发送给硬盘，就可以使IO的吞吐量更大。但是这样做也有另一个问题，就是如果此时出现了一个请求，它要访问的磁道离目前磁头所在磁道很远，应用的请求又大量集中在目前磁道附近。导致大量请求一直会被合并和插队处理，而那个要访问比较远磁道的请求将因为一直不能被调度而饿死。deadline就是这样一种调度器，能在保证IO最大吞吐量的情况下，尽量使远端请求在一个期限内被调度而不被饿死的调度器。

DEADLINE设计原理

为了实现上述目标，deadline调度器实现了两类队列，一类负责对请求按照访问扇区进行排序。这个队列使用红黑树组织，叫做sort_list。另一类对请求的访问时间进行排序。使用链表组织，叫做fifo_list。

由于读写请求的明显处理差异，在每一类队列中，又按请求的读写类型分别分了两个队列，就是说deadline调度器实际上有4个队列：

1. 按照扇区访问顺序排序的读队列。
2. 按照扇区访问顺序排序的写队列。
3. 按照请求时间排序的读队列。
4. 按照请求时间排序的写队列。

deadline之所以要对读写队列进行分离，是因为要实现读操作比写操作更高的优先级。从应用的角度来看，读操作一般都是同步行为，就是说，读的时候程序一般都要等到数据返回后才能做下一步的处理。而写操作的同步需求并不明显，一般程序都可以将数据写到缓存，之后由内核负责同步到存储上即可。所以，对读操作进行优化可以明显的得到收益。当然，deadline在这样的情况下必然要对写操作会饿死的情况进行考虑，保证其不会被饿死。

deadline的入队很简单：当一个新的IO请求产生并进行了必要的合并操作之后，它在deadline调度器中会分别按照扇区顺序和请求产生时间分别入队sort_list和fifo_list。并再进一步根据请求的读写类型入队到相应的读或者写队列。

deadline的出队处理相对麻烦一点：

1. 首先判断读队列是否为空，如果读队列不为空并且写队列没发生饥饿（starved < writes_starved）则处理读队列，否则

处理写队列（第4部）。
2. 进入读队列处理后，首先检查fifo_list中是否有超过最终期限（read_expire）的读请求，如果有则处理该请求以防止被饿死。
3. 如果上一步为假，则处理顺序的读请求以增大吞吐。
4. 如果第1部检查读队列为空或者写队列处于饥饿状态，那么应该处理写队列。其过程和读队列处理类似。
5. 进入写队列处理后，首先检查fifo_list中是否有超过最终期限（write_expire）的写请求，如果有则处理该请求以防止被饿死。
6. 如果上一步为假，则处理顺序的写请求以增大吞吐。

整个处理逻辑就是这样，简单总结其原则就是，读的优先级高于写，达到deadline时间的请求处理高于顺序处理。正常情况下保证顺序读写，保证吞吐量，有饥饿的情况下处理饥饿。

DEADLINE的参数调整

deadline的可调参数相对较少，包括：

```
[root@zorrozou-pc0 zorro]# echo deadline > /sys/block/sdb/queue/scheduler
[root@zorrozou-pc0 zorro]# ls /sys/block/sdb/queue/iosched/
fifo_batch  front_merges  read_expire  write_expire  writes_starved
```

read_expire:读请求的超时时间设置，单位为ms。当一个读请求入队deadline的时候，其过期时间将被设置为当前时间 + read_expire，并放倒fifo_list中进行排序。

write_expire:写请求的超时时间设置，单位为ms。功能根读请求类似。

fifo_batch:在顺序（sort_list）请求进行处理的时候，deadline将以batch为单位进行处理。每一个batch处理的请求个数为这个参数所限制的个数。在一个batch处理的过程中，不会产生是否超时的检查，也就不会产生额外的磁盘寻道时间。这个参数可以用来平衡顺序处理和饥饿时间的矛盾，当饥饿时间需要尽可能的符合预期的时候，我们可以调小这个值，以便尽可能多的检查是否有饥饿产生并及时处理。增大这个值当然也会增大吞吐量，但是会导致处理饥饿请求的延时变长。

writes_starved:这个值是在上述deadline出队处理第一步时做检查用的。用来判断当读队列不为空时，写队列的饥饿程度是否足够高，以时deadline放弃读请求的处理而处理写请求。当检查存在有写请求的时候，deadline并不会立即对写请求进行处理，而是给相关数据结构中的starved进行累计，如果这是第一次检查到有写请求进行处理，那么这个计数就为1。如果此时writes_starved值为2，则我们认为此时饥饿程度还不足够高，所以继续处理读请求。只有当starved >= writes_starved的时候，deadline才回去处理写请求。可以认为这个值是用来平衡deadline对读写请求处理优先级状态的，这个值越大，则写请求越被滞后处理，越小，写请求就越可以获得趋近于读请求的优先级。

front_merges:当一个新请求进入队列的时候，如果其请求的扇区距离当前扇区很近，那么它就是可以被合并处理的。而这个合并可能有两种情况，一个是向当前位置后合并，另一种是向前合并。在某些场景下，向前合并是不必要的，那么我们就可以通过这个参数关闭向前合并。默认deadline支持向前合并，设置为0关闭。

NOOP调度器

noop调度器是最简单的调度器。它本质上就是一个链表实现的fifo队列，并对请求进行简单的合并处理。调度器本身并没有提供任何可疑配置的参数。

各种调度器的应用场景选择

根据以上几种io调度算法的分析，我们应该能对各种调度算法的使用场景有一些大致的思路了。从原理上看，cfq是一种比较通用的调度算法，它是一种以进程为出发点考虑的调度算法，保证大家尽量公平。deadline是一种以提高机械硬盘吞吐量为思考出发点的调度算法，尽量保证在有io请求达到最终期限的时候进行调度，非常适合业务比较单一并且IO压力比较重的业务，比如数据库。而noop呢？其实如果我们把我们的思考对象拓展到固态硬盘，那么你就会发现，无论cfq还是deadline，都是针对机械硬盘的结构进行的队列算法调整，而这种调整对于固态硬盘来说，完全没有意义。对于固态硬盘来说，IO调度算法越复杂,额外要处理的逻辑就越多，效率就越低。所以，固态硬盘这种场景下使用noop是最好的，deadline次之，而cfq由于复杂度的原因，无疑效率最低。

三月 14, 2016 / Linux / 留下评论

关于情商

中午根同事一起吃饭，同事因为看了我的博客上的《你爱我，与我何干？》而感慨说：唉，你的情商也太低了。怎么能跟长辈这么说该你屁事这样的话？碰巧，当我跟伯父说这话的时候，父亲就提醒过我，也说我这样说话情商低，在外容易得罪人。其实几乎每次我放假回家跟父母在一起的时候，他们都会不断的提醒我，说我们家人都是直脾气，不懂得圆滑处事，他们年轻的是候就吃过很多亏，所以希望我能吸取教训，不要重蹈覆辙，做人要圆滑。那么我理解的这个“圆滑”，其实就是所谓的高情商，也就是说，我在他们眼里也是低情商。曾经那个不太懂事的我也一直以来我也都以为自己知道自己是低情商、不圆滑的人，并且曾以此为个性。直到有一天，我认真思考了一下这个问题：我的情商为什么低？于是随之而来的另一个问题就是：到底什么是情商？于是我就找了很多资料，其中知乎上的几个帖子让我茅塞顿开：
https://www.zhihu.com/question/21112415
https://www.zhihu.com/question/20543245

不知道有多少人思考过这个问题：什么是情商？

情商的概念属于心理学范畴我们这里不去细节讨论定义，如果只从大家方便理解的方面考虑，情商主要是指以下两个能力：

1、控制自己情绪的能力。
2、控制别人情绪的能力。

人类是一个社会动物，我们需要协作，那么必然需要交流，而所谓的情商也就表现在这个过程中。以我为例，我表达自己的观点时，说的是：关你屁事！我为什么要这样说？因为这个语境和用词可以最直观的表达出我对此事的情绪和看法，这是成本最低的表达方式。当然，当时的上下文语境也并不是跟谁吵架，而仅仅是单纯的表达态度。我本身说这句话并不带情绪，选词也是深思熟虑过的。并且我也做好了长辈会发作的心里预期。而长辈此时确实因为我的用词和语气而发作，认为我是不礼貌的。当然之后我们没有不欢而散，而是我陪上笑脸赶紧道歉，最后长辈也明白了我要表达的意思。在这个过程中，究竟谁情商更高呢？

如果仅从自我的情绪控制上看，我毫不谦虚的说，本人情商很高，我可以在整个过程中控制好自己的情绪。该表达观点的时候表达，该哄长辈开心的时候就哄。而相反是长辈情绪控制相对较差，因为自己辈份大而不能接受平等的沟通，于是就以愤怒来对待，这是情商差的表现。当然，我这位大伯毕竟生活阅历丰富，还是哄的好的，在我的道歉和劝说下，他还是能够控制住自己情绪的，还是有一定情商的。在相同的场景下，很多人是连哄也哄不好的，情绪自控能力当然更差。

通过这个例子来看，我发现其实并不是我的情商有多低，而往往是别人控制自己情绪的能力太差。这些人因为一些原因不能跟你平等沟通，当你直接表达自己意见的时候，他们就可能受不了。这仍然与我国的传统文化密不可分，我们强调三纲五常，君君臣臣、父父子子。我们的文化中没有平等，所有人都应在在自己的位置上，面对别人的时候要么是跪着唯唯诺诺，要么是站着颐指气使。我们的传统文化中根本没有情商的概念，所以对于中华文化来说，所谓的高情商、圆滑、老油条其实就是装孙子。就是我不管你怎么样，我先跪下行不？你总不能说我情商低了吧？

请别理解成我是给自己的不礼貌找借口，起码的礼貌我们当然要讲，起码的尊老爱幼我们当然要坚坚持。尊敬领导，尊重同事这些基本的道德必须有。我很清楚我根长辈说关你屁事是不礼貌的，我道歉并也认为自己是错的。长辈当然有权利生气，但是不要混淆概念，当我们讨论情商的时候，就是指我们再某些情况下能否控制自己的情绪。

这只是情商的一方面，是从管理自己情绪来说的。情商还有管理别人情绪能力的要求。说白了就是控制别人情绪的能力。一般情况我们都是希望让别人高兴的，因为绝大多数场景都是我们有求于别人，让人开心了才能获得利益。但是不排除让人生气也可能获得利益的情况，比如：三气周瑜。你会觉得诸葛亮在气周瑜的时候是情商低么？在这个例子中，诸葛亮恰恰是情商管理大师。所以说，控制别人的情绪最重要的是同理心，就是很清楚别人什么情况下会生气，什么情况下会开心，并以此决定自己的行为，来达到目的。从这一点说，当你在说某一句话之前，就知道对方会是什么反应的时候，你的情商实际上就是高人一筹了。但是也会有很多人因为对自己的情绪掌握不好，而在明知道对方会不开心的情况下，还非要这样做，导致一个损人不利己的结果，这必然是低情商的表现。所以，我觉得提高情商的核心方法仍然是修炼对自己情绪的控制。

在管理学的著作中，情商也更加强调控制自己情绪，因为别人的情绪虽然可以被引导，但是想要在管理上真正可控是不可能的，毕竟那是别人的情绪。我认为，管理学强调对自己的情绪的控制是很好的思路，因为在一个合作场景中，如果每个人都可以提升自己情绪的控制能力，那么其他人就可以减少在控制别人情绪方面所付出的成本，而使沟通变得更直接和顺畅。这也就是现代组织形式和传统体制化的组织形式的人际关系的差别：当组织不存在竞争压力的时候，其体制就会腐化为强调伦常道德，官本位等体制化方式运作，因为资源都集中在上层，你只有让上层的人舒服了，你才能分到资源，你要么跪着，要么站着。有人形容这样的体制就像是一群猴子爬树，上面的猴子看下面，一群笑脸。下面的猴子看上面，全是屁股。其实全都只是屁股还好，有时你还要接着屁和屎。当组织是在自由市场竞争，需要通过不断提升服务能力和降低成本而盈利而生存的时候，当然内部沟通成本越少越好，这时大家需要把精力都用在做事上，所以需要高情商的人合作，大家都能控制自己的情绪，减少不必要的沟通成本。

这也许也就是乔布斯说的那句话的原因：一流人才的自尊心，不需要你呵护。

二月 23, 2016 / Uncategorized / 留下评论

你爱我，与我何干?

春节期间休息在家陪父母，今天又是姑姑生日，于是按照传统要去她家聚会，这几乎是每年的惯例。饭桌上酒过三巡菜过五味后，大伯表达了一下他对我们这一代人的看法，大概的意思就是说我们这一代人对人际关系的处理比较差劲，比如跟亲戚之间有隔阂，堂兄弟之间也不爱多联系；比如对待爱情也不成熟，处理不好夫妻男女朋友之间的关系。总之，在他眼中，我们这一代人就是比较矫情。由于之前他也一直好奇我为什么总不爱说话，所以我决定在这一话题上代表我们这“一代人”表示一下我的看法。我不知道我的答案是不是能代表我们这一代人，我说的是：“关你屁事？”

结果呢？自然是大伯大怒，当场怒斥我不礼貌。我呢？当然是立即承认错误，之后继续不说话，然后听大伯和我父亲以及其它亲戚结成同盟批评我：“不养儿不知父母恩”等等等等大家可以自行补脑的话。

春节对于我来说，从小到大就是这样的纠结的一个节。在这样的喜庆日子里，亲友团聚的时刻却总是让我感觉到无地自处，全身都不适应。每年，都会经历这样一轮节日的洗礼，这种洗礼的结果就是，你总是在不断的反省是不是自己什么地

方做错了？需要改正什么地方？因为按照他们的说法，由于我们的各种不“正常”，我们早该在这个社会上被淘汰掉了，我们早该没有任何生存能力养活自己而回家继续啃老了。不过我是如此幸运，社会是如此宽容，我还能养的活自己。但是事实是这样么，该反省的真的是我们么？

看看日历恰逢今天又是情人节，所以觉得很适合讨论一个话题，就是爱情。其实爱情是我们国家传统文化中的最核心的内容，儒家倡导的最核心的思想就是仁。仁的意思就是“仁者爱人”，就是说白了就是要爱别人的意思。仁字也设计的很形象，二人世界。于是我们的传统文化中所倡导的很多关系都是这种二人世界的演化版，比如：孝、忠、悌。仔细分析一下，中国人所有的关系似乎都只是二人世界，而自己的独立生活空间往往是没有的。就比如什么是孝？顺者为孝。就是一切按照老人的想法做就是顺，于是就做到了孝。就比如我，由于怕说实话伤害了长辈的感情而不孝，所以就不说话。于是人家觉得你是不是有心事？有什么难处？让人操心了，我就又是不孝。我说了实话，于是就是不礼貌，不注意讲话，又是不孝。当然我也可以选择说假话，赞成大伯的说法，说我们这代人就是矫情，就是处理不好各种关系，那我是不是也太让他们不省心了？先不说我自己是不是能接受这种自我摧残，即便说了承认了自己不省心这样的话骗他们是不是也是不孝？

分析这样的问题，我有一个笨办法，就是看看我能选择的所有情况是不是都是错误答案，如果是，那就说明是题出错了。这明显就是题出错了。那么究竟错在哪里？我爱你，错了么？

对于爱，曾经最喜欢张爱玲的一句话：我爱你，与你无关。曾经我认为这是最健康的爱的方式。但是我从没有仔细想过这句话也可以用在亲情的爱上。中国人对于爱，有着太多的憧憬。这种憧憬实际上更多的是对回报的憧憬。中国人对爱的隐含意思是：既然我这么爱你，你就应该以我觉得合适的方式和量级来回报给我。如果不是这样，那就是你不爱我。

我想这就是这个社会中绝大多数人对爱的逻辑。于是才有了逼婚，才有了结婚后逼生孩子。有多少人连自己的结婚日期都不是自己定的，而是按照父母的时间表由父母进行安排，搞得自己都搞不清楚是不是自己结婚？甚至有的父母连你考什么大学，报什么专业都帮你确定好了的？更有甚者，父母从小就开始培养孩子的某一个技能，以便让孩子长大以后就从事相关工作。在他们眼里，孩子已经不是孩子了，只是他们实现自己梦想的工具而已。他们以爱的名义不同程度的侵入了孩子的私人空间，并美其名曰，我爱你！搞得你无论怎么选都是错的，因为你不可以有自己的选择。这不是你的人生，这是他们的人生。

说到这里，有些不近人情了。父母可能真的并不觉得这是子女的私事，他们也真的觉得这是爱你。并且会觉得我们不养儿不知父母恩，等你养了孩子你就知道了。我确实还没养孩子，我也确实可能不理解他们的心情，但是关于这一点，我想我更赞同胡适的看法，在胡适答汪长禄的信中他这样写道：

““父母于子无恩”的话，从王充、孔融以来，也很久了。从前有人说我曾提倡这话，我实在不能承认。直到今年我自己生了一个儿子，我才想到这个问题上去。我想这个孩子自己并不曾自由主张要生在我家，我们做父母的不曾得他的同意，就糊里糊涂的给了他一条生命。况且我们也并不曾有意送给他这条生命。我们既无意，如何能居功？如何能自以为有恩于他？他既无意求生，我们生了他，我们对他只有抱歉，更不能“市恩”了。我们糊里糊涂的替社会上添了一个人，这个人将来一生的苦乐祸福，这个人将来在社会上的功罪，我们应该负一部分的责任。说得偏激一点，我们生一个儿子，就好比替他种下了祸根，又替社会种下了祸根。他也许养成坏习惯，做一个短命浪子；他也许更堕落下去，做一个军阀派的走狗。所以我们“教他养他”，只是我们自己减轻罪过的法子，只是我们种下祸根之后自己补过弥缝的法子。这可以说是恩典吗?

所说的，是从做父母的一方面设想的，是从我下人对于我自己的儿子设想的，所以我的题目是“我的儿子”。我的意思是要我这个儿子晓得我对他只有抱歉，决不居功，决不市恩。至于我的儿子将来怎样待我，那是他自己的事。我决不期望他报答我的恩，因为我已宣言无恩于他。”

等我有了孩子之后，我将要以这样的心情教他养他。也做一个像张爱玲、胡适那样有着自由主义爱情观的人。

也许我们这一代的人的责任就是，承接好上一代对我们“无微不至”的爱，并把自由主义爱情观通过行动传递给下一代，让他们有独立的人格和生活空间。

那么今天，请允许我说：你爱我，与我何干？

二月 14, 2016 / Uncategorized / 留下评论

Cgroup - Linux的网络资源隔离

Cgroup – Linux的网络资源隔离

Hi，我是Zorro。这是我的微博地址，如果你有兴趣，可以来关注我呦。

这是我的博客地址，我会不定期在这里更新文章，如有谬误，欢迎随时指正。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

由于本文不会涉及一些网络基础知识的讲解以及iproute2相关命令的使用的讲解，建议如果想要更好理解本文，之前应该对网络知识、tc命令和LARTC的文档有一定了解。如果本文中有什么知识点让不够清楚，可以结合LARTC文档一起服用。

想要直接上手配置cgroup的网络资源隔离的人，可以直接看本文倒数第二部分：**使用cgroup限制网络流量**。

本文PDF

今天我们来谈谈：

##Linux的网络资源隔离

如果说Linux内核的cgroup算是个新技术的话，那么它的网络资源隔离部分的实现算是个不折不扣的老技术了。实际上是先有的网络资源的隔离技术，才有的cgroup。或者说是先有的网络资源的隔离才有的2.4、2.6版本的Linux内核，而现在的最主流的内核版本应该是3.10了（考虑到android手机的出货量，你公司那几千几万台服务器真的算是个零头对吧？）。好吧，Linux早在内核2.2版本就已经引入了网络QoS的机制，并且网络资源的隔离功能只是其所实现功能的一部分而已。无论如何，cgroup并没有再重新搞一套网络资源隔离的实现，而是直接使用了Linux的iproute2的traffic control（tc）功能。实际上网络资源隔离的文档真的不用我再多写什么了，我最亲爱的前同事＋朋友＋导师——johnbull同志早已经在2003年的非典期间就因为无聊而完成了非常高质量的相关技术文档翻译工作，将这方面最权威的LARTC（Linux Advanced Routing & Traffic Control）文档翻译成了中文版。

英文版链接

中文版链接

曾经chinaunix的资深版主johnbull同志现在在新浪微博工作，所以经常在微博出没，如果对以上文档有兴趣和疑问的人可以直接去找他对质，传送门在此。

其实原则上说，本技术文章已经讲完了，但是为了不让大家有种上当受骗的感觉，我觉得我还是有必要从cgroup的角度再来讲讲tc，也算是对TC近几年发展做一个补充。

###什么是队列规则

tc命令引入了一系列概念，其中我们最需要先理解的就是**队列规则**。它的英文名字叫做queueing discipline，在tc命令中也叫qdisc，或者直接简写为qd。我们先来看看它，有个感性的认识：

在我的虚拟机的centos7中，它是这样的：

```
[root@localhost Desktop]# tc qd ls
qdisc pfifo_fast 0: dev eno16777736 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0
1 1 1 1 1 1 1 1
```

在我的台式机上装的archlinux（更新到了当前最新版的4.3.3内核）以及fedora 23上是这样的：

```
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp2s0 root refcnt 2 limit 10240p flows 1024 quantum 1514
target 5.0ms interval 100.0ms ecn
```

在公司的服务器上是这样的：

```
[root@tencent64 /data/home/zorrozou]# tc qd ls
qdisc mq 0: dev eth1 root
qdisc pfifo_fast 0: dev tun0 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1
1 1 1 1
qdisc pfifo_fast 0: dev veth213_121_54 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0
0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_135_194 root refcnt 2 bands 3 priomap  1 2 2 2 1 2
0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_25 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0
0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_121_112 root refcnt 2 bands 3 priomap  1 2 2 2 1 2
0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_207 root refcnt 2 bands 3 priomap  1 2 2 2 1 2
0 0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_123_82 root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0
0 1 1 1 1 1 1 1 1
qdisc pfifo_fast 0: dev veth213_117_111 root refcnt 2 bands 3 priomap  1 2 2 2 1 2
0 0 1 1 1 1 1 1 1 1
```

从以上输出大家应该可以判断出来，这个所谓的qdisc是针对网卡的，每有一个网卡就会有一个qdisc。而且如果你用过ip命令并且比较细心的话，应该早就注意到ip ad sh的时候也会出现相关的信息：

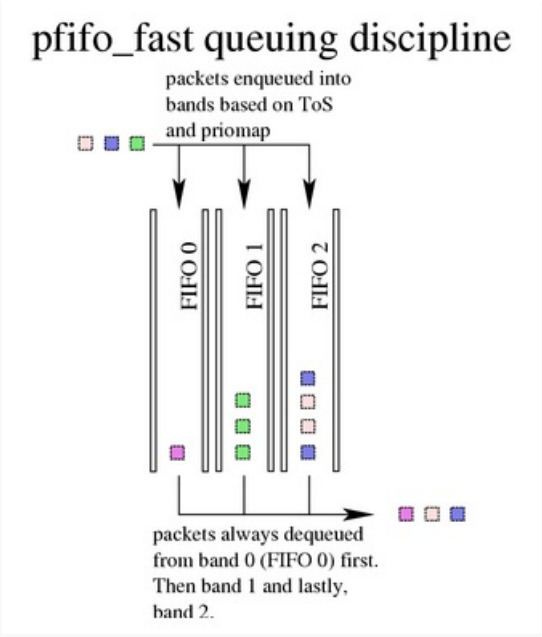
```
[zorro@zorrozou-pc0 ~]$ ip ad sh
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    link/ether 34:64:a9:15:a2:17 brd ff:ff:ff:ff:ff:ff
    inet 10.18.73.69/24 brd 10.18.73.255 scope global dynamic enp2s0
       valid_lft 28283sec preferred_lft 28283sec
    inet6 fe80::3664:a9ff:fe15:a217/64 scope link
       valid_lft forever preferred_lft forever
```

虽然看上去有些高深莫测，但是qdisc其实是个挺简单的概念，它就是它字面的意思：队列规则，或者叫做排队规则。我们都知道，网络数据都是被封装成一个一个的数据包进行传输的。如果网卡相当于数据包要出发的大门的话，那么qdisc无非就是规定了这些包在出发前如果需要排队的话该怎么排。我们先拿这个叫做pfifo_fast的队列规则来举例子描述一下吧，这个qdisc实现了一个以数据包（package）为单位的fifo队列，实际上可以认为是实现了三个队列（叫做bands），给每个队列定了一个优先级，以实现带优先级的排队规则。我们举个现实中的例子再来说明一下，大家都应该有去公交车站排队的经验吧？（神马？作为中国人你从来不排队？）无论怎样，我们假定你是排队的。每来一次公交车，就相当于网卡处理一次队列中的数据包，而每个人就是一个数据包。那么我们一般人到了公交站，如果发现前面已经排了一队人，此时根据fifo（first in first out）的规则，我们会排在队列尾部。如果来车了，就从队列头的人先上车，车满就走，没上完的人继续等待。但是我们也知道，如果此时来了个孕妇或者大爷大娘等一些按照我们社会美德要求应该让他们优先的乘客的话，这些人应该有权利优先上车。那么怎么办呢？我们公交站台的解决办法一般是直接让他们去队列头插队就好，但是如果空间允许的话，我们可以考虑多建立一个队列。让这些可以优先上车的人排一个队，正常人排一个队，车来了先上优先级比较高的那个队列中的人，他们都上完了再让一般队列中上人车。这样就实现了一个简单的队列规则，大家根据自己的情况去选择排队就好了。

pfifo_fast实现了一个类似上述描述的队列规则，区别是它实现了3个优先级的队列（bands），每个数据包来了都根据自己的情况选择一个band进行排队，每个band都是fifo方式处理数据包。它总是先处理优先级最高的band，直到没有数据包了再处理下一个优先级的band，直到三个都处理完，或者本次处理不完，继续等着下次处理。那么数据包按什么规则进行选择自己该进入哪个band呢？这就是后面显示的**priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1**的含义，这个字段描述了一个priomap，可以理解为优先级位图，后面的16个不同的位，表示相关制如果为真时的数据包应该进入哪个队列，一共有0、1、2三个队列。而这个16位的位图标记，针对的就是我们IP报头中的TOS字段。根据IP协议的定义我们知道，TOS字段8位中的4位分别是用来标示最小延时、最大吞吐量、最大可靠性和最小消费四种数据包类型的，IP协议原则上会根据这些标示的不同以不同的QOS对上层交付不同的服务质量。这些不同的搭配理论上一躬有16种，这就是priomap所映射的优先级的概念。

> *如果你对TOS的概念还不熟悉，请自行补充网络相关基础知识。推荐的教材是《TCP／IP详解卷1》。*

pfifo_fast队列处理过程如图所示：

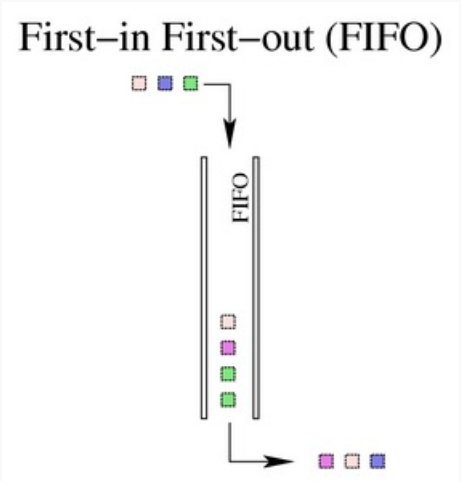


pfifo_fast一般情况下是内核对网卡默认选择的qdisc，它虽然提供了简单的优先级分类的支持，但是并没有提供可供修改的参数，就是说默认的优先级分类设置不能更改，也没有提供相关限速的功能。这个队列规则在一般情况下工作的都很稳定，但是最近Linux已经开始放弃使用这个qd作为默认的队列规则而改用一种叫做fq_codel的qdisc了。主要原因是，由于移动互联网的广泛应用，一种叫做Bufferbloat的现象影响越来越大了。

###Bufferbloat

Bufferbloat现象最初是用来形容在一个分组交换网络上，路由器为防止丢包，往往buffer缓冲区都会实现的很大，但是这种过大的fifo缓冲区可能导致数据包buffer中等待时间过长而导致很多问题（后面会有分析）。再加上网络上TCP的拥塞控制算法的影响，以及很多商业操作系统甚至并不实现拥塞控制，导致数据传输质量抖动很大（全局同步），甚至于达到服务不可用的状态。

后来我们发现，Bufferbloat这种现象比较广泛的存在在各种系统中，只要系统中使用了类似队列、缓存等机制的时候，就在某些极端状态下出现这种类似雪崩的现象。我们简要描述一下这个状态。我们先简单构建一个试用buffer的场景，如图所示：



根据图的描述，我们假定这个简单的fifo就是我们要的buffer系统，它在两个处理过程之间充当缓冲区的作用。每个请求从队列的上面进入排队，然后依次被下面的处理程序处理。大家应该知道buffer的作用：一个缓冲器的作用主要是弥补两个处理系统之间的速度差异，能够在一定程度的请求速度抖动的时候缓解处理速度慢而导致的请求失败。假设，后段处理请求的速度为1000个/s，每个请求平均长度为100byte，队列长队为1Mbyte，此时，如果请求突然增加到了2000个/s，那么这个压力直接压给后端是处理不过来的，每秒钟就要丢弃1000个包。所以我们使用一个buffer，可以让这一秒钟来的请求先处理1000个，然后有1000个排在队列中，下一秒处理。只要来的请求的抖动范围还算正常，我们的系统将会工作的良好，

没有失败的请求。

对于一般的系统，我们发送的请求都是有延时要求的，鉴于我们的系统每秒钟可以处理1000个请求，所以每个请求的处理时间平均为1ms。而我们的系统基于目前的处理时间，对外提供了100ms的延时SLA，就是说，后端系统保证每个请求的处理时间是100ms以内，这已经很大了，是正常情况的100倍。于是前端的请求方，会根据后端给出的SLA在程序中设定一个超时时间，在这个例子中就应该是100ms，这可能意味着，程度调用后端系统，如果等待100ms还没有结果，那么将重试一次或者几次不等，之后应该会返回失败。场景就是这样一个场景，那么我们来看看究竟什么是bufferbloat？

假定现在因为业务问题，比如上线了一个秒杀的抢购活动，导致从前端发来的请求一瞬间远远大于后端的处理能力。比如，一秒钟内产生了10000次请求，这一万次请求都会立即进入队列中等待后端处理。因为后端的处理速度是1000次每秒，所以可以想像，当前在队列中的最后一个数据包至少要等待9秒钟才能处理到。实际上根本处理不到这最后一个请求，由于我们设置了100ms的超时时间，那么调用方将很快因为发现100ms中没有返回而重试一次，于是又来了将近10000个请求。这些请求都积压在了队列中，还没交给后端进行处理，如果交给了后端处理，后端肯定会因为压力变大处理变慢，而导致处理事件超过100ms的SLA，会在超时之后告诉前端本次请求失败（如果是这样实现的话），而现在由于队列的存在，并大量的积压请求，导致调用方不能明确的得知失败。所以一般都是等待至少一次超时重试一次再失败，当然也有很多情况会重试个4，5次也说不定。

无论如何，这突发的10000个请求的流量来了之后，如果平均每个请求100字节，这1M的缓冲区就已经满了，后续再有任何请求来，都会排在队列末尾，一直等到前面的请求处理完再处理这个请求，而此时因为整体处理时间很慢，要将此队列中的全部请求处理完需要9秒钟，无论如何，这个请求都已经超时失败了。这个时候后端服务一直满载的处理队列中的请求，而前端还不断有新请求源源不断的放进队列，但是由于超时，前端所有请求都是返回失败，后端所处理的请求也都是等待时间超过100ms的无效的请求，即使成功返回结果给前端，前端也不会要了。效果就是后端很忙，而整体服务却是不可用的。此时哪怕请求平均速度恢复到1000个每秒，服务也无法恢复。这就是一个典型的bufferbloat场景。

于是我们可以考虑一下这个场景会发生在什么地方？比如buffer比较大的路由器，由于tcp的流量控制和重试机制导致网络质量的抖动；比如一个后端的数据库系统为了能够承载更大的吞吐量而添加了队列系统；比如io调度；比如网卡调度；只要是大buffer的场景都会可能产生类似的问题。那么该如何解决这个问题呢？于是主动队列管理算法应运而出了。

###AQM

AQM就是主动队列管理（Active Queue Management）的英文缩写，其思路就是对buffer缓冲的队列管理采取有效的主动管理手段，并不等待队列满之后才被动丢弃请求，而是在某个条件触发的情况下主动对请求进行丢弃，以防止类似Bufferbloat现象的发生。最简单的AQM思路就是监控队列长度，当队列长度一直维持在最大长度的时候，开始对新入队的数据包进行丢弃，直到使拥塞恢复（根据上面的例子可以想像，不断减少队列长度，就可以让新来的请求等待时间变短，直到可以正常服务）。这种做法虽然可以最终使拥塞恢复，但是整个过程并不十分理想，bufferbloat现象仍然存在。由于是对新入队数据包进行丢弃，所以容易在类似TCP拥塞控制的使用场景下引发**全局同步**现象，在很多场景下还会有死锁。所以我们需要更先进的队列管理算法。

###RED算法

RED算法主要是为了解决全局同步现象而产生的算法，其基本思路是，通过监控平均队列长度来探测是否有拥塞，一旦发现开始拥塞，就以某一个概率从队列中（而不是队列尾）开始丢弃请求（在网络上也可以通过ecn通知连接有拥塞）。

对于RED来说，关键的可配置参数有这样几个：

min:最小队列长度。

max:最大队列长度。

probability:可能性，取值范围为0.00 – 1，一般可以理解为百分比，比如0.01为1%。

有了以上几个关键参数，RED算法就可以工作了，其工作的原理大概是这样的。首先，RED会对目前队列状态计算一个平均队列长度（算法采用的是指数加权平均算法计算的，在此不做更细节的说明），然后检查当前队列的平均长度是否：

1. 低于min：此时不做任何处理，队列压力较小，可以直接正常处理。
2. 在min和max之间：此时界定为队列感受到阻塞压力，开始按照某一几率P从队列中丢包，几率计算公式为：P = probability * (平均队列长度 – min)/(max – min)。
3. 高于max：此时新入队的请求也将丢弃。

所以probability可以理解为当队列感受到阻塞压力的时候，最大的丢包几率是多少。知道了这几个参数，我们就可以了解一下如何在Linux上进行RED的配置了，其实很简单，使用以下命令即可：

```
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp2s0 root refcnt 2 limit 10240p flows 1024 quantum 1514
target 5.0ms interval 100.0ms ecn
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 root red limit 200000 min 20000 max
40000 avpkt 1000 burst 30 ecn adaptive bandwidth 5Mbit
[root@zorrozou-pc0 zorro]# tc qd ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc red 8001: dev enp2s0 root refcnt 2 limit 200000b min 20000b max 40000b ecn
adaptive
```

这样我们就将默认的qdisc规则改为了RED，解释一下相关参数：

首先是命令前部分：

```
tc qd add dev enp2s0 root
```

这部分没什么可解释的，唯一需要说明的是root参数，这个参数表示**根节点**，修改了这个参数描述的队列一般表示我们整个这个网卡所发出的数据包都用的是指定的规则，暂时我们还用不到其他节点，所以就只是root就可以了。另外请注意，目前所学习的队列规则只对发出的数据包有效。

之后是red参数，在这里描述使用什么队列规则。在之后丢失red这个队列规则所要使用的参数描述，具体可以通过man tc-red找到帮助。我们简单解释一下：

limit:此队列的字节数硬限制。配置的长度应该比max大。但是需要注意的是max和min的单位是数据包个数而不是字节数。

avpkt:平均包长度。这个参数是用根burst参数一起来计算平均队列长度的参数，所以选择一个合适的值对整体效果的影响较大。一般的推荐值为1000。

burst:字面含义是队列可以容纳的爆发流量。但是我们知道，爆发流量的承载是根据队列容量上限（limit）决定的，当一个大于当前网络带宽处理能力的爆发流量来临时，不能及时发出的数据包将缓存在队列中，队列满了就会丢包。所以实际影响爆发流量承载能力的是limit参数。当然我们建议的limit长度应该是不少于max+burst的长度，这样才能有实际意义。但是这个参数将对平均队列长度的变化速度产生影响，可以想像，如果我们想要支持队列能处理尽可能大的爆发流量的

话，当队列突然变长的时候，应该让平均队列长度的计算结果变化没那么敏感，这样爆发流量来的时候丢包的可能性会减小。所以，这个值设置的越高，那么平均队列长度的计算敏感度就约小，变化速度将会变慢，反之变快。

bandwidth:用于在网络空闲的时候计算平均队列长度的参数，应该配置成你的网络的实际带宽大小。并不是说RED有限速作用。

ecn:ecn实际上是TCP/IP协议用来通知网络拥塞所实现的一个数据报字段。添加这个参数标示意味着，当RED检测到拥塞都是通过标记数据包的ecn字段来通知数据源端减少数据发送量，并且在实际队列长度达到limit限制之前丢不会丢弃数据包。

adaptive:算是一种更智能的probability参数的选择，添加了这个参数之后就可以不用人为指定一个固定的probability了，当平均队列长度超过(max-min)/2时，RED会动态的根据情况让probability的值在1%到50%之间变化。具体描述参见这里。

以上就是RED队列规则的配置方法和意义，其作用主要是缓解全局同步的问题。但是我们在实际使用的时候发现，RED的min、max、probability这些参数的选择在实际场景中可能会根据情况变化而改变才是最优的，但是RED的配置不能自适应这些变化。并且实际上在很多特定的网络负载下依然会导致TCP的全局同步。这些缺陷促使我们寻找更优秀的方式来解决这些问题。

> *内核还实现了另一个队列规则叫做choke，其所有配置参数跟RED完全一样，区别是，RED是通过字节为单位进行队列控制，而choke是以数据包为单位。更多帮助请：man tc-choke*

###CoDel算法

CoDel算法是另一种AQM算法，其全称是Controlled Delay算法。是由Van Jacobson和Kathleen Nichols在2012年实现的。具体描述参见Controlling Queue Delay。CoDel采用了另外一种角度来观察队列满载的问题，其出发点并不是对队列长度进行控制，而是对队列中的数据包的驻留时间进行控制。事实上如果我们将管理方式由队列长度控制变成等待时间控制的时候，bufferbloat就可以彻底解决了，这也是更先进的AQM算法所用的方式。我们仔细观察bufferbloat问题，会发现，引起这个问题的重要原因就是数据包在队列中的驻留时间过长，超过了有效的处理时间（SLA定义的时间或者重试时间），导致处理到的数据包都已经超时。

首先我们根据我们的业务设计，确定出请求在队列中正常情况应该驻留多久。我们还是假定这样一种场景，根上面bufferbloat中描述的例子差不多：后端处理速度是1000次每秒，就是1ms可以处理一个请求，而队列平均长度一般为5，就是说一个新请求进入队列之后，发现前面还有5个请求在等待，那么这个新请求的处理时间大约为6ms（在队列中等待5ms）。那么请求在队列中的驻留时间正常情况下基本为5ms。而我们服务的SLA确定的时间是100ms（由诸如服务超时时间或者所在网络的最大RTT时间等条件确定），就是说，服务应确保在100ms内给出反馈，这个时间叫做interval time，如果超过这个时间应该返回失败。针对这样的情况，我们可以根据请求驻留时间的情况来描述一个动态长度的队列，当一个请求入队之后，对其驻留时间（sojourn time）进行追踪，以正常的情况作为其目标驻留时间（target time），在这个例子中是5ms，就是说一般情况下，我们期望请求在队列中的驻留时间不高于5ms。由于业务的超时时间或者说我们提供的SLA处理时间是100ms，所以，在这个队列中驻留超过100ms的请求都应该丢弃（从队列头开始），因为即使处理完成它们也没有意义了。丢弃将持续到队列中的请求等待时间回到理想的target time为止，并且队列长度整体不大于队列容量上限。这样就根据驻留时间维持了一个动态长度的队列，这个队列中的所有请求理论上都应该等待100ms以内，要么被正常处理掉，要么被丢弃。这就是CoDel算法的基本思路。

为了有助于大家理解，我们再详细一点描述一下这个算法的处理过程：

CoDel算法对队列状态维护一个状态机，进行队列dequeue处理的时候，先判检查队列头请求的驻留时间（sojourn time）是否大于target time，如果不大于target time，就直接dequeue；如果大于（target time）的请求维持了interval time这么

长的时间，则队列应该进入dropping状态开始丢包。这种丢包状态将可能维持一段时间，这段时间的长度将根据情况而定（驻留时间一直处在target以上，并且下一个包丢弃的时间采用逆平方根运算（inverse-square-root）,公式为：

```
t（第一次取now，以后取上次的值） + interval / sqrt(count))
```

count的取值为丢弃包的个数，如果count大于2则count = count – 2，其他情况count取值为1。直到驻留时间小于target time，就退出dropping状态。

算法的伪代码描述参见这里。

> *我们之所以要如此详细的描述bufferbloat问题以及其解决方案，尤其是CoDel算法，原因是其不仅仅被用在网络的分组交换和路由的处理上。除了TC的队列规则外，CoDel当前还被用在了内核TCP协议栈的拥塞控制中，并且rabbitmq也已经把这个算法应用于消息队列的延时控制了,参见。这个算法在数据中心的应用场景下，是一个非常好的解决队列阻塞的方案。*

了解了以上知识之后，我们来看一下再Linux上如何配置一个CoDel的队列规则，我们刚才已经将队列规则改为RED了，此时如果要将其改为CoDel，需要先删除RED的队列规则，再添加新的队列规则：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc qdisc add dev enp2s0 root codel limit 100 target 4ms
interval 30ms ecn
[root@zorrozou-pc0 zorro]# tc qd ls dev enp2s0
qdisc codel 8002: root refcnt 2 limit 100p target 4.0ms interval 30.0ms ecn
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc codel 8002: root refcnt 2 limit 100p target 4.0ms interval 30.0ms ecn
 Sent 5546 bytes 39 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  count 0 lastcount 0 ldelay 0us drop_next 0us
  maxpacket 0 ecn_mark 0 drop_overlimit 0
```

tc的-s参数相信你已经明白什么意思了。来说一下codel队列规则的相关参数：

limit:队列长度上限，如果超过这个长度，新来的数据包将被直接丢弃。单位为字节数，默认值为1000.

target && **interval**:这两个参数相信大家已经明白是什么意思了，根据自己的场景进行配置就好了。

ecn && **noecn**:这个参数的含义根RED中的一样，默认是开启的ecn方式通知源端，不丢包。

大家也可以直接使用codel规则的默认参数，就是其他参数都省略即可。我们来看看什么效果：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc qdisc add dev enp2s0 root codel
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc codel 8003: root refcnt 2 limit 1000p target 5.0ms interval 100.0ms
 Sent 8613 bytes 33 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  count 0 lastcount 0 ldelay 0us drop_next 0us
  maxpacket 0 ecn_mark 0 drop_overlimit 0
```

###fq-codel队列规则

在比较老版本的Linux内核上，由于当时还没实现基于CoDel算法的队列规则，所以一直使用的是pfifo_fast作为默认队列规则。作为一个简单的队列规则，pfifo_fast越来越不能适应Linux的发展需要。这个发展主要指的是Linux作为android系统的操作系统内核被广泛用在了手机等移动互联设备上。在移动互联网的场景下，网络延时问题变的更普遍，而导致网络上的bufferbloat问题变成了急需解决的问题。于是，CoDel的算法引入变的非常必要。CoDel算法虽然比较高质量的解决了bufferbloat问题，但是并没有解决多链接处理的公平性问题。这个公平性问题其实也比较好理解，因为网络有不同的传输要求，某些传输数据量很大，但是延时要求不大，某些则是数据量很小，但是延时要求很高（IP协议TOS字段所描述的情况）。如果各种链接占用同一个队列，那么数据量大的的连接势必数据包就更多，那么从概率上讲，这样的连接挤占队列的能力就更强。而主动队列管理一般都是以ecn或者丢包为手段的，如果丢弃的是那些延时要求较高的连接的数据包，又会对用户的服务质量感受造成很大的影响。所以，最好的办法就是实现一个针对每一个数据流(flow)公平的CoDel队列规则，就是fq-codel。

fq-codel叫做flow queue codel调度，因为其特点也被叫做fair queue codel（完全公平）。fq-codel为每个需要使用网络的flow创建一个单独的队列（实际上是默认实现1024个队列，使用五元组hash给相关flow选择一个队列），队列之间使用针对字节的DRR(Deficit Round Robin)调度算法进行调度。它的工作方式是跟踪每个队列的当前差额（*deficit*）的字节个数。这个差额的初始值可以用quantum参数指定。每当一个队列获得发送数据（出队）的机会时就开始发送数据包，并根据发送的数据包的字节数减少deficit的值，直到这个值变为负值的时候，对其增加一个quantum的大小，并且本队列发送结束，调度下一个队列。

这意味着，如果目前有两个队列，一个队列中的数据包长度都是quantum/3这么大，而另一个队列中的数据包长度每个都是一个quantum长度的话，调度器处理第一个队列的时候，每次处理3个数据包，而第二个队列就只能处理1个数据包。这意味着DRR算法对每个队列发送数据的时候是针对字节数计数，不会因为数据包数的大小而有差别。

quantum取值的大小决定了调度周期的粒度，所以也就决定了调度器的调度开销。当网络带宽比较小的时候，推荐的设置是从默认的MTU的值来取quantum的值，并可以考虑适当减小这个值。

不同于标准DRR调度的地方是，我们的调度器将所有flow队列分成了两个sets。实际上可以认为所有队列有两个分类，一类里面都是new flow，针对新建的网络连接；而另一类是old flow，针对原来机已经建立的网络连接。

Interval

这个值的意义根CoDel算法中的语义完全一样，是用来确定最小延时时间的取值不至于导致数据包长时间在队列里堆积。最小延时的取值必须根据上一个周期interval检查的经验而得来，应该被设置为，数据包通过网络瓶颈点发给对端之后，能够接收到对端返回的确认的最差RTT时间。

默认间隔时间值为100ms。

Target

这个值的意义根CoDel算法中的语义完全一样，是用来设定在FQ – CoDel的每个队列中数据包的最小延时时间（可以等待的最长时间）的。最小延时时间是通过追踪本地最小队列延时的经验得来的。

默认的Target值为5ms，但是这个值应该根据本地的网络情况得来，最少应配制成本地网络的mtu长度的数据包在相应的带宽环境下发送的时间。（如：本地网卡mtu为1500，带宽为1Mbps的情况下，应配置为15ms。）

下面简述一下fq-codel的处理过程：

FQ-CoDel的入队（enqueue）：

入队由三个步骤组成：根据flow特点进行分类选择一个队列，记录数据包入队时间并记账（bookkeeping），另外如果队列满了还会丢弃数据包。

分类的时候会根据数据包的源、目的ip；源、目的端口和使用的协议（五元组）并参杂一个随机数，用这个值对队列个数取模运算，得出把这个flow放到哪个队列中。

FQ-CoDel的出队（dequeue）：

队列规则的绝大多数工作都是在出队的时候做的。分三个步骤：选择从那个队列发送数据包；dequeue数据包（在所选队列中处理CoDel算法）；记账（bookkeeping）；

在第一部分处理的过程中：调度器先查找new list队列，对这个list中的每个队列进行处理处理，如果队列有负的赤字（negative deficit）说明起已经被出队了至少一个quantum的字节数，那么就说明这个队列已经不再是new队列了，则追加到old list中，并且给其增加一个quantum的字节数的deficit，然后处理new list中的下一个队列。

如果选择的队列不是上述情况，就说明这是一个new队列，则对其dequeue。如果new列表为空，则开始处理old列表，处理过程根上述过程类似。

选择好处理哪个queue之后，CoDel算法就会作用于这个队列。这个算法可能在返回需要dequque的数据包之前，先删除队列中的一个或者多个数据包，数据包的删除是从队列头开始的。

最后，如果CoDel没有返回需要dequeue的数据包，或者队列为空，调度器将根据情况做这两件事的其中一个：如果队列是new列表中的队列，则将其移动到old列表的最后一个。如果队列是old列表中的队列，那么这个队列讲从old列表中删除，直到下次这个队列中有数据包需要处理的时候，就再把它加到new列表中。如果所有队列中都没有需要dequeue的数据包之后，就对所有队列重来一次上述调度过程。

如果调度算法返回了一个需要dequeue的数据包，处理过程将会先去处理deficit数字，然后对数据包进行相关dequeue处理。

检查new列表并把符合条件的队列移动到old列表这个过程会因为可能存在的无限循环而导致饥饿。则是因为当某一个数据流符合一个速率进行小包发送的时候，这个队列会在new列表中重现，而导致调度器一直无法处理old列表。预防这种饥饿的方法是，在第一次讲队列移动到old列表的时候，强制跳过不再检查。

以上过程更详细的描述参见。我们再来看看如何配置一个fq-codel队列规则。跟刚才步骤类似：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root
[root@zorrozou-pc0 zorro]# tc -s qd ls dev enp2s0
qdisc fq_codel 0: root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5.0ms
interval 100.0ms ecn
 Sent 7645 bytes 45 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
  maxpacket 0 drop_overlimit 0 new_flow_count 0 ecn_mark 0
  new_flows_len 0 old_flows_len 0
```

其实我们会发现，作为默认的队列规则，删除了原来配置的队列规则之后，显示的就是fq-codel了，默认参数就是显示的这样了。这个队列规则包含的参数包括：

limit

flows

target

interval

quantum

ecn | noecn

帮助可以参见man tc-fq_codel。唯一需要再稍作解释的就是flows，这个参数决定了有少个队列，默认1024。

另外，内核还提供了一个fq队列，实际上就是fq-codel不带codel的一个基于DRR算法的公平队列。这里没有更多参考，你可以直接使用这个队列。

> *本节涉及到了一个负载均衡算法，DRR－基于赤字的轮训算法。实际上内核也实现了一个专门的DRR调度队列，大家可以参考man tc-drr。关于这个算法本身的描述请自行查找资料。*

###SFQ随机公平队列

首先我要引用LARTC中文版中对SFQ队列的讲解，毕竟这已经足够权威了：

> *SFQ(Stochastic Fairness Queueing，随机公平队列)是公平队列算法家族中的一个简单实现。它的精确性不如其它的方法，但是它在实现高度公平的同时，需要的*
> *计算量却很少。SFQ 的关键词是"会话"(或称作"流")，主要针对一个 TCP 会话或者 UDP流。流量被分成相当多数量的 FIFO 队列中，每个队列对应一个会话。数据按照*
> *简单轮转的方式发送, 每个会话都按顺序得到发送机会。这种方式非常公平，保证了每一个会话都不会没其它会话所淹没。SFQ 之所以被称为"随机"，是因为它并不是真的为每一个会话创建一个队列，而是使用一个散列算法，把所有的会话映射到有限的几个队列中去。因为使用了散列，所以可能多个会话分配在同一个队列里，从而需要共享发包的机会，也就是共享带宽。为了不让这种效应太明显，SFQ 会频繁地改变散列算法，以便把这种效应控制在几秒钟之内。有很重要的一点需要声明：只有当你的出口网卡确实已经挤满了的时候，SFQ 才会起作用！否则在你的 Linux 机器中根本就不会有队列，SFQ 也就不会起作用。稍后我们会描述如何把 SFQ 与其它的队列规定结合在一起，以保证两种情况下都比较好的结果。特别地，在你使用 DSL modem 或者 cable modem 的以太网卡上设置 SFQ 而不进行任何进一步地流量整形是无谋的！*
>
> *SFQ 基本上不需要手工调整：*
> *perturb:多少秒后重新配置一次散列算法。如果取消设置，散列算法将永远不会重新配置（不建议这样做）。10 秒应该是一个合适的值。*
> *quantum:一个流至少要传输多少字节后才切换到下一个队列。却省设置为一个最大包的长度(MTU的大小)。不要设置这个数值低于 MTU！*
>
> *如果你有一个网卡，它的链路速度与实际可用速率一致——比如一个电话MODEM——如下配置可以提高公平性：*

```
# tc qdisc add dev ppp0 root sfq perturb 10
```

```
# tc -s -d qdisc ls

qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

> *“800c:”这个号码是系统自动分配的一个句柄号，“limit”意思是这个队列中可以有 128 个数据包排队等待。一共可以有 1024 个散列目标可以用于速率审计，而其中 128 个可以同时激活。(no more packets fit in the queue!)每隔 10 秒种散列算法更换一次。*

以上是对SFQ队列的权威解释，但是毕竟时过境迁，目前的实现稍有不同。现在的SFQ在原有队列的基础上实现了RED模式，就是针对每一个SFQ队列，都可以用RED算法来防止bufferbloat问题。目前的RED跟SFQ队列规则的关系有点像codel跟fq_codel队列规则之间的关系，它们一个是基础版算法的队列实现，另一个是其多队列版。

新版中需要解释的参数:

redflowlimit:用来限制在RED模式下的SFQ的每个队列的字节数上限。

perturb:默认值为0，表示不重新配置hash算法。原来为10，单位是秒。

depth:限制每一个队列的深度（长度），默认值127，只能减少，单位包个数。

如果需要配置一个RED模式的SFQ，操作方式如下：

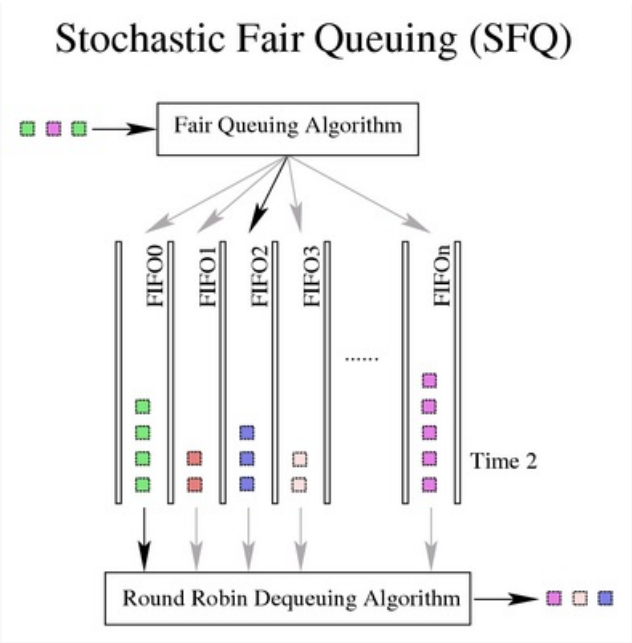
```
tc qdisc add dev eth0 parent 1:1 handle 10: sfq limit 3000 flows 512 divisor 16384
redflowlimit 100000 min 8000 max 60000 probability 0.20 ecn headdrop
```

更多的帮助情参阅:

```
man tc-sfq
```

> *内核还给我们提供了一个名叫sfb的随机公平队列，相对sfq来说，sfb的意思就是采用的blue算法对每个队列进程处理。什么是blue算法？这是相对于red来说的（有红的算法，也要有蓝的）。我们不对BLUE算法做更详细的解释了，大家有兴趣可以自行查找资料。*

SFQ的结构如下：



###PIE比例积分控制队列

PIE是Proportional Integral controller Enhanced的简写，其中文名称是加强的比例积分控制。比例积分控制是非常有名的一种工控算法。想要详细了解这个方法的，可以自行查阅相关资料。而在tc的队列规则中，pie是内核帮我们实现的另一个用来解决bufferbloat问题的AQM机制。其控制思路跟CoDel一样，都是针对请求的延时进行控制而不是队列长度，但是其对超时请求处理方法跟RED一样，都是随机对数据包进行丢弃。

PIE是根据队列中请求的延时情况而对不同级别的拥塞做出相关的相应动作的（比如丢包），严格来说，是根据队列中请求延时时间的变化率（就是当前延时时间与目标延时时间的差值与时间的积分）来判断。这就能做到影响算法参数值选择是根据稳态感受的变化而变化的，目的就是可以让算法本身在各种网络阻塞的情况下都能自动调节以优化性能表现。

PIE包括三个简单的必需组件：1.入队时的随机丢弃；2.周期的更新丢弃可能性比率（probability）；3.对延时（latency）进行计算。当一个请求到达队列时（入队之前），会被评估这个请求是否会被随机丢弃。丢弃的几率会根据目前的延时状态和目标延时（target）的差距（比例控制）以及队列的延时是否变长或者变短（积分控制）的状态，每隔一定时间周期（tupdate）进行更新。队列的延时是通过直接测量请求的等待时间或计算队列长度和出队速率获得的。

跟其他最先进的AQM算法一样，当一个数据包到达时PIE会根据一个随机丢弃的可能性p来丢弃数据包，p的计算方式如下：

1. 首先根据以下公式估计当前队列延时：
   est_del = qlen/depart_rate;
2. 计算丢弃可能性几率p：
   p = p + alpha * (est_del – target_del) + beta * (est_del – est_del_old);
   est_del_old = est_del.

以上计算过程会按一定时间周期进行估算，周期的时间由tupdate参数指定，est_del是当前周期的队列延时，est_del_old是上一个周期的队列延时，target_del是目标延时。qlen表示当前队列长度。

alpha是用来确定当前的延迟与目标延时的偏差将如何影响丢弃概率。beta值会对整个p的估算起到另一个校准作用，这个作用通过目前的延时是在上升还是在下降进行估算的。请注意p的运算是一个逐渐达到的过程（积分过程），并不是一步达到的。在运算p的时候，为了避免校准过程中比较大的波动，我们一般是对p做小的增量调整。假设p在1%的范围内，那么我们希望单步校准的幅度也比较小，比如0.1%，那么alpha和beta也都要足够小，。但是如果p的值更高了，比如说达到了10%，在这种情况下，我们的单步教准的幅度也希望更大，比如达到1%。所以我们在p取值的每一个量级范围内，都可能需

要一个单步的调教幅度的取值范围，在必要的情况下p可能会精确到0.0001%。这个单步调校的范围可以通过类似这样一个方式实现：

```
    if (drop_prob_ < 0.000001) {
        drop_prob_ /= 2048;
    } else if (drop_prob_ < 0.00001) {
        drop_prob_ /= 512;
    } else if (drop_prob_ < 0.0001) {
        drop_prob_ /= 128;
    } else if (drop_prob_ < 0.001) {
        drop_prob_ /= 32;
    } else if (drop_prob_ < 0.01) {
        drop_prob_ /= 8;
    } else if (drop_prob_ < 0.1) {
        drop_prob_ /= 2;
    } else {
        drop_prob_ = drop_prob_;
    }
```

对p进行调校的目标是让p稳定下来，稳定的条件就是当队列的当前延时等于目标延时，并且延时状态已经稳定的情况（就是说est_del等于est_del_old）。alpha和beta的取值实际上就是一个权重值，如果alpha较大则丢弃几率对延时偏移（latency offset即相对于目标延时的差距）更敏感，如果beta较大则丢弃几率p对延时抖动（latency jitter即相对于上周期延时的差距）更敏感。

计算周期tupdate参数也是一个让整个校准过程能够稳定发挥效果的重要参数，当我们配置更快的tupdate周期，并且alpha和beta的值相同时，则周期增益效果更明显。请注意alpha和beta的配置单位是hz，由于在上面的计算公式中表示的不明显，所以这可能会成为配置出错的地方。

请注意，丢弃可能性p的计算不仅与当前队列延时的估算有关，还与延时变化的方向有关，就是说，延时变大或者变小都会影响计算。延时变化的方向可以从当前队列延时和之前一个周期的队列延时进行比较来确定。这就是采用标准的比例积分控制算法对队列的延时进行控制。

队列的出队速率可能会经常波动，造成这种情况的原因是我们可能与其它队列共享同一个连接设备，活着链路的容量波动。在无线网络的情况下，链路的波动尤其常见。因此，我们通过以下方法直接测量出队速率：

当队列中有足够的数据时，才进入测量周期：

```
qlen > dq_threshold
```

进入测量周期之后，在数据包出队时：
dq_count = dq_count + deque_pkt_size;

然后判断dq_coune是不是高于采样阈值：
if dq_count > dq_threshold then

```
depart_rate = dq_count/(now-start);

dq_count = 0;
```

```
start = now;
```

我们只在队列中存在足够的数据的时候才计算出队速率，就是当队列长度超过deq_threshold这个阈值的时候。这是因为时不时出现的短的和非持久性的爆发数据流量进入空队列时会使测量不准确。参数dq_count表示从上次测量之后离开的字节数，一旦这个值超过了deq_threshold阈值，我们就得到一次有效的测量采样。在数据包长度在1k到1.5k长度的时候，我们建议dq_count的值为16k，这样的设置既可以让我们有足够长的时间周期来对出队速率做平均，也能够足够快的反馈出出队速率的突然变化。这个阈值并不影响系统的稳定性。

除了上面的基本算法描述以外，PIE算法该提供了一些其它增强功能来提升算法的性能：

网络流量往往都会有一定的自然波动，当队列的延时因为这样的波动而出现临时性的“虚假”上涨的时候，我们不希望在这样的情况下引起不必要的丢包。所以，PIE算法实现了一个自动开启和关闭算法的机制，当队列长度不足缓冲区长度的1/3时，算法是不会生效的，此时处于关闭状态，当队列中的数据量超过了1/3这个阈值的时候，算法自动打开，开始对队列中的数据进行处理。当阻塞情况完全恢复的时候，就是说丢弃概率、队列长度和队列延时都为0的时候，PIE的作用关闭。

虽然PIE采用随机丢弃的策略来处理入队的数据包，但是仍然可能会有几率因为丢弃的数据包很连续或者很稀疏而导致丢弃效果偏离丢弃几率p。这就好比抛硬币问题，虽然概率上出现正面或者反面的几率都是50%，但是当你真的去抛硬币的时候，仍然可能碰见连续多次的出现正面或者反面的情况。所以，我们引入了一种“去随机”的丢弃机制来防止这样的事情发生。我们引入了一个参数prob，当发生丢弃的时候，这个参数被重置为0，当数据包到达进行丢弃判断的时候，prob参数也会进行累加，累加的值是每次计算丢弃概率得到p这个值的总量。prob会有一个阈值下限和一个上限，当累计的prob低于阈值下线的时候，我们不丢包，直接入队，当高于阈值上限的时候，我们无论几率如何，强制丢包。只有当prob在阈值下限和上限之间时，我们才按照p的几率丢弃数据包。这样就能保证，如果几率导致连续没丢包，积累到一定程度后一定会丢包，另一方面，如果丢包，则prob一定在下限以下，则下一个包一定会入队，以防止问题的发生。

关于PIE更多的资料，可以参考这里。

###TBF队列

以上的算法主要都是解决bufferbloat问题的。我们可以看到Linux内核为了适应移动互联网的环境做了很多努力。而接下来我们要介绍的TBF（令牌痛过滤器）是我们遇到的第一个可以对流量进行整形（就是限速）的算法。自它诞生到现在，基本功能没有什么太大变化，毕竟token bucket filter算法已经是一个非常经典的限速算法了。所以我们只需要引用LARTC中的讲解即可：

令牌桶过滤器(TBF)是一个简单的队列规定：只允许以不超过事先设定的速率到来的数据包通过，但可能允许短暂突发流量朝过设定值。TBF 很精确,对于网络和处理器的影响都很小。所以如果您想对一个网卡限速，它应该成为您的第一选择！TBF 的实现在于一个缓冲器(桶)，不断地被一些叫做“令牌”的虚拟数据以特定速率填充着。 (token rate)。桶最重要的参数就是它的大小，也就是它能够存储令牌的数量。每个到来的令牌从数据队列中收集一个数据包，然后从桶中被删除。这个算法关联到两个流上——令牌流和数据流，于是我们得到 3 种情景：• 数据流以等于令牌流的速率到达 TBF。这种情况下，每个到来的数据包都能对应一个令牌，然后无延迟地通过队列。

• 数据流以小于令牌流的速度到达 TBF。通过队列的数据包只消耗了一部分令牌，剩下的令牌会在桶里积累下来，直到桶被装满。剩下的令牌可以在需要以高于令牌流速率发送数据流的时候消耗掉，这种情况下会发生突发传输。

• 数据流以大于令牌流的速率到达 TBF。这意味着桶里的令牌很快就会被耗尽。导致 TBF 中断一段时间，称为“越限”。如果数据包持续到来，将发生丢包。

最后一种情景非常重要，因为它可以用来对数据通过过滤器的速率进行整形。令牌的积累可以导致越限的数据进行短时间的突发传输而不必丢包，但是持续越限的话会导致传输延迟直至丢包。

请注意，实际的实现是针对数据的字节数进行的，而不是针对数据包进行的。

即使如此，你还是可能需要进行修改，TBF 提供了一些可调控的参数。第一个参数永远可用：

limit/latency

limit 确定最多有多少数据（字节数）在队列中等待可用令牌。你也可以通过设置 latency 参数来指定这个参数，latency 参数确定了一个包在 TBF中等待传输的最长等待时间。后者计算决定桶的大小、速率和峰值速率。

burst/buffer/maxburst

桶的大小，以字节计。这个参数指定了最多可以有多少个令牌能够即刻被使用。通常，管理的带宽越大，需要的缓冲器就越大。在 Intel 体系上，10 兆 bit/s 的速率需要至少 10k 字节的缓冲区才能达到期望的速率。如果你的缓冲区太小，就会导致到达的令牌没有地方放（桶满了），这会导致潜在的丢包。

mpu

一个零长度的包并不是不耗费带宽。比如以太网，数据帧不会小于 64 字节。Mpu(Minimum Packet Unit，最小分组单位)决定了令牌的最低消耗。

rate

速度操纵杆。参见上面的 limits！如果桶里存在令牌而且允许没有令牌，相当于不限制速率(缺省情况)。If the bucket contains tokens and is allowed to empty, by default it does so at infinite speed. 如果不希望这样，可以调整入下参数：

peakrate

如果有可用的令牌，数据包一旦到来就会立刻被发送出去，就象光速一样。那可能并不是你希望的，特别是你有一个比较大的桶的时候。峰值速率可以用来指定令牌以多块的速度被删除。用书面语言来说，就是：释放一个数据包，但后等待足够的时间后再释放下一个。我们通过计算等待时间来控制峰值速率然而，由于 UNIX 定时器的分辨率是10 毫秒，如果平均包长 10k bit，我们的峰值速率被限制在了 1Mbps。

mtu/minburst

但是如果你的常规速率比较高，1Mbps 的峰值速率对我们就没有什么价值。要实现更高的峰值速率，可以在一个时钟周期内发送多个数据包。最有效的办法就是：再创建一个令牌桶！这第二个令牌桶缺省情况下为一个单个的数据包，并非一个真正的桶。要计算峰值速率，用 mtu 乘以 100 就行了。 (应该说是乘以 HZ 数，Intel体系上是 100，Alpha 体系上是 1024)

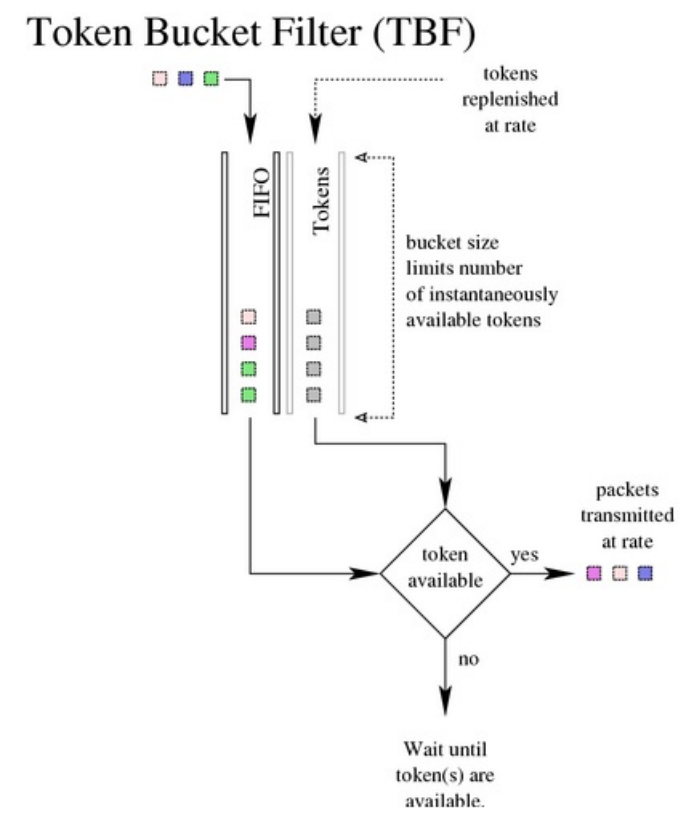
这是一个非常简单而实用的例子：

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

为什么它很实用呢？如果你有一个队列较长的网络设备，比如 DSL modem 或者cable modem 什么的，并通过一个快速设备(如以太网卡)与之相连，你会发现上载数据绝对会破坏交互性。这是因为上载数据会充满 modem 的队列，而这个队列为

了改善上载数据的吞吐量而设置的特别大。但这并不是你需要的，你可能为了提高交互性而需要一个不太大的队列。也就是说你希望在发送数据的时候干点别的事情。上面的一行命令并非直接影响了 modem 中的队列，而是通过控制 Linux 中的队列而放慢了发送数据的速度。把 220kbit 修改为你实际的上载速度再减去几个百分点。如果你的 modem 确实很快，就把“burst”值提高一点。

以上为引用原文内容，请原谅我的懒惰。TBF结构图如下：



###分类(class)、过滤器(filter)以及HTB

基于目前我们已经知道的这些内容，我们已经可以在一个运行着比较复杂的网络服务的系统环境中按照网络的数据流为调度对象，建立一个比较公平的队列环境了，并且还能避免bufferbloat现象。比如fq-codel、sfq等队列规则都能做到。这也是内核目前选择fq-codel作为默认队列规则的初衷。实际上这已经可以适应绝大多数场景了。

但是在一些QoS要求更高的场景中，我们可能需要对网络流量的服务做更细节的分类，来实现更多的功能。比如说我们有这样一个场景：我们的服务器上运行了一个web服务，对外服务端口是tcp的80，还运行了一个邮件服务，对外服务协议是smtp的tcp的25端口，可能还会开一个sshd以便管理员可以远程控制，其端口为22。我们的对外带宽一共为10Mbps。我们想要做到这样一种效果，当所有服务都很繁忙的需要占用带宽时，我们希望80端口上限不超过6Mbps，25端口上限不超过3Mbps，而22端口1Mbps足够了。当其它端口不忙的时候，某个端口可以突破自己的上限带宽设置能达到10Mbps的带宽。这种网络资源分配策略跟cgroup的cpushare方式的分配概念类似。

当我们的需求负载到类似这样的程度时，我们会发现以上的各种队列规则都不能满足需求，而能满足需求的队列规则都起吗必需实现一个功能，就是对数据包的分类（class）功能，并且这个分类要能够人为指定分类策略（实际上pfifo_fast本身对数据包进行了分类，但是并不能人为改变分类策略，所以我们仍然把它当成不可分类的队列规则）。比如针对当前的例子，我们就至少需要三个分类（可以认为就是三个队列），然后把从80端口发出的数据包都排进分类1里，从25端口发出的数据包排进分类2里，再将22端口发出的数据包放到分类3里。当然如果你的服务器还有别的服务也要用网络，可能还要额外配置一个分类或者共用以上某一个分类。

在这个描述中，我们会发现，当需求确定了，分类也就可以确定了，并且如何进行分类（过滤方法）也就可以确定了。如

果我们还是把数据包比作去公交车站排队的人的话，那么可分类的队列规则就相当于公交站有个管理人员，这个管理人员可以根据情况自行确定目前乘客可以排几个队、哪个队排什么样特征的人。自然，我们需要在每个乘客来排队之前，根据确定好的策略对乘客进行过滤，让相关特征的乘客去相应的队伍。这个决定乘客所属分类的人就是过滤器（filter）。

以上是我对这两个概念的描述，希望能够帮助大家理解。相关概念的官方定义在此。

由于相关知识的细节说明在LARTC中已经有了更细节的说明，我们再次不在废话。我们直接来看使用HTB（分层令牌桶）队列规则如何实现上述功能，其实无非就是以下系列命令：

首先，我们需要先讲当前网卡的队列规则换成HTB，保险起见，可以先删除当前队列规则再添加：

```
[root@zorrozou-pc0 zorro]# tc qd del dev enp2s0 root    [root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 root handle 1: htb default 30
```

default参数的含义就是，默认数据包都走标记为30的类（class）。
然后我们开始建立分类，并对各种分类进行限速：

```
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1: classid 1:1 htb rate 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:10 htb rate 6mbit ceil 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:20 htb rate 3mbit ceil 10mbit burst 20k
[root@zorrozou-pc0 zorro]# tc cl add dev enp2s0 parent 1:1 classid 1:30 htb rate 1mbit ceil 10mbit burst 20k
```

这样我们建立好了一个root分类，id为1:1，速率上限为10mbit。然后在这个分类下建立了三个子分类，id分别为1:10、1:20、1:30，这个10、20、30的编号就是针对上面default的参数，你想让默认数据流走哪个分类，就在default参数后面加上它相应的id即可。我们建立了分类并且给分类做了速度限制，并且使用ceil参数指定每个分类都可以在其它分类空闲的时候借用带宽资源最高可以达到10mbit。

之后是给每个分类下再添加相应的过滤器，我们这里分别给三个分类使用了不同的过滤器，以实现不同的Qos保障。当然，每个子分类下还可以继续添加htb过滤器，让整个htb的分层树形结构变的更大，分类更细。一般情况下，两层的结构足以应付绝大多数场景了。

```
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:10 handle 10: fq_codel
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:20 handle 20: sfq
[root@zorrozou-pc0 zorro]# tc qd add dev enp2s0 parent 1:30 handle 30: pie
```

最后，我们使用u32过滤器，对数据包进行过滤，这两条命令分别将源端口为80的数据包放到分类1:10里，源端扣为25的数据包放到分类1:20里。默认其它数据包（包括22），根据default规则走分类1:30。

```
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 u32 match ip dport 80 0xffff flowid 1:10
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 u32
```

```
match ip dport 25 0xffff flowid 1:20
```

至此，htb以及u32过滤器的简单使用介绍完毕。

内核除了实现了u32过滤器来帮我们过滤数据包以外，还有一个常用的过滤器叫fw，就是实用防火墙标记作为数据包分类的区分方法（firewall mark）。我们可以先使用iptable的mangle表对数据包先做mark标记，然后在tc中使用fw过滤器去识别相应的数据包，并进行分类。还是用以上的例子进行说明，此时我们使用fw过滤器的话，最后两条命令将变成这样：

```
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1
handle 1 fw flowid 1:10
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1
handle 2 fw flowid 1:20
```

这两条命令说明，凡是被fwmark标记为1的数据包都走分类1:10，标记为2的走分类1:20。之后，别忘了在iptable里面添加对数据包的标记：

```
[root@zorrozou-pc0 zorro]# iptables -t mangle -A OUTPUT -p tcp --sport 80 -j MARK -
-set-mark 1
[root@zorrozou-pc0 zorro]# iptables -t mangle -A OUTPUT -p tcp --sport 25 -j MARK -
-set-mark 2
```

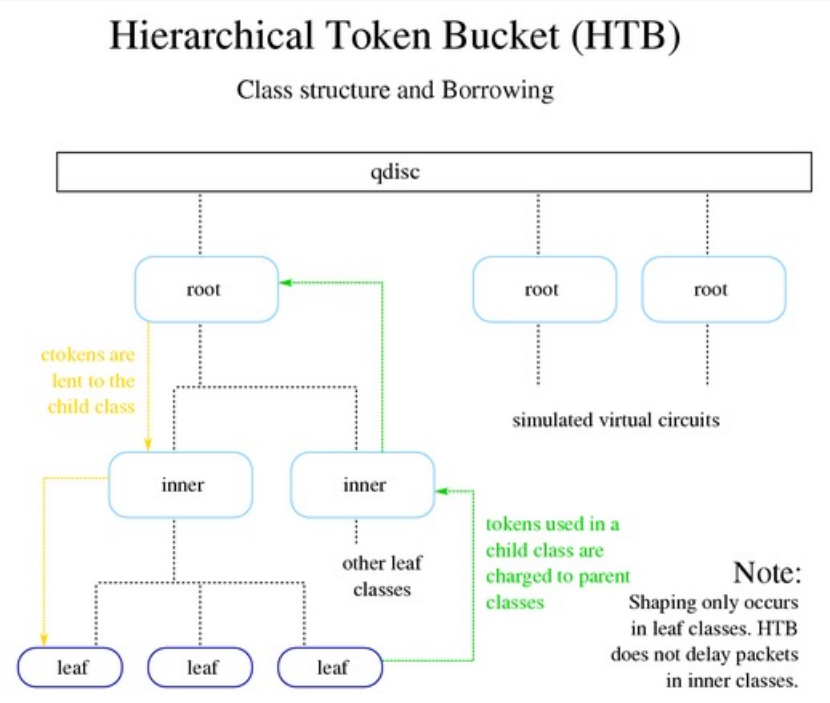
如果你不想学习u32过滤器哪些复杂的语法，那么fwmark是一种很好的替代方式。当然前提是你对iptables和tcp／ip协议有一定了解。

> *思考题：*
> *添加完iptables规则后，我们可以通过以下命令查看目前mangle表的内容：*
>
> ```
> [root@zorrozou-pc0 zorro]# iptables -t mangle -L
> Chain PREROUTING (policy ACCEPT)
> target     prot opt source               destination
> Chain INPUT (policy ACCEPT)
> target     prot opt source               destination
> Chain FORWARD (policy ACCEPT)
> target     prot opt source               destination
> Chain OUTPUT (policy ACCEPT)
> target     prot opt source               destination
> MARK       tcp  --  anywhere             anywhere             tcp spt:http MARK
> set 0x1
> MARK       tcp  --  anywhere             anywhere             tcp spt:smtp MARK
> set 0x2
> Chain POSTROUTING (policy ACCEPT)
> target     prot opt source               destination
> ```
>
> *在本例中，我们使用了其中的OUTPUT链添加了规则。那么问题是：使用不同的链（Chain）的区别是什么？*

因为一些原因，我们推荐使用HTB的方式对比较复杂的网络数据包进行分类并流量整形。当然，可分类的队列规则中，除

了HTB还有PRIO以及非常著名的CBQ。其中CBQ尤其在网络设备的限速方面有着最广泛的使用，但是如果从软件实现的角度来说，令牌桶方式(htb就是分层令牌桶)的流量限制在性能和稳定性上都更具有优势。PRIO由于分类过于简单，并不适合更复杂的场景。

关于这些知识的介绍，大家依然可以在LARTC上找到更详细的讲解。根据上面的命令，我们再参照结构图来理解一下HTB：



###使用cgroup限制网络流量

如果你是从头开始看到这里的，那么真的很佩服你的耐心。我们前面似乎讲了一堆跟cgroup做网络资源隔离没有关系的知识，但是无疑每一个知识点的理解对于我们规划网络的资源隔离都有很重要的作用。毕竟，我么要规划一个架构，必需了解清楚其相关实现以及要解决的问题。但是很不幸，我们依然没有能够讲完目前所有的qdisc实现，比如还有HFSC、ATM、MULTIQ、TEQL、GRED、DSMARK、MQPRIO、QFQ、HHF等，这些还是留着给大家自己去解密吧。相信大家如果真正理解了队列规则要解决的问题和其基础知识，理解这些东西并不难。

最后，我们要来看看如何在cgroup的场景下对网络资源进行隔离了。实际上跟我们上面讲的HTB的例子类似，区别是，上面的例子是通过端口分类，而现在需要通过cgroup进行分类。我们还是通过一个例子来说明一下场景，并实现其功能：我们假定现在有两个cgroup，一个叫jerry，另一个叫zorro。我们现在需要给jerry组中运行的网络程序限制带宽为10mbit，zorro组的网路资源占用为20mbit，总带宽为100mbit，并且不允许借用（ceil）网络资源。那么配置思路是这样：

我们的配置环境是一台centos7的虚拟机，首先，我们在这个服务器上运行一个apache的http服务，并发布了一个1G的数据文件作为测试文件，并在不限速的情况下对齐进行下载速度测试，结果为100MBps，注意这里的速度是byte而不是bit：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   101M      0  0:00:10  0:00:10 --:--:--  100M
```

之后我们在centos7(192.168.139.136)上实现三个分类，一个带宽限制10m给jerry，另一个20m给zorro，还有一个为30m用作default，总带宽100m，剩余的资源给以后可能新加入的cgroup来分配，于是先建立相关的规则和分类：

```
[root@localhost Desktop]# tc qd add dev eno16777736 root handle 1: htb default 100
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1: classid 1:1 htb rate
100mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:10 htb
rate 10mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:20 htb
rate 20mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:100 htb
rate 30mbit burst 20k

[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:10 handle 10: fq_codel
[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:20 handle 20: fq_codel
[root@localhost Desktop]# tc qd add dev eno16777736 parent 1:100 handle 100:
fq_codel
```

建立完分类之后，由于默认情况都要走1:100的分类，所以限速应该是30mbit，验证一下：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 3484k    0     0  3452k      0  0:05:03  0:00:01  0:05:02 3452k
```

当前速度为3452kB左右，大概为30mbit，符合预期。之后将我们的http服务放到zorro组中看看效果，当然是首先建立相关cgroup以及相关配置：

```
[root@localhost Desktop]# ls /sys/fs/cgroup/net_cls/
cgroup.clone_children  cgroup.event_control  cgroup.procs  cgroup.sane_behavior
net_cls.classid  notify_on_release  release_agent  tasks
[root@localhost Desktop]# mkdir /sys/fs/cgroup/net_cls/zorro
[root@localhost Desktop]# mkdir /sys/fs/cgroup/net_cls/jerry
[root@localhost Desktop]# ls /sys/fs/cgroup/net_cls/{zorro,jerry}
/sys/fs/cgroup/net_cls/jerry:
cgroup.clone_children  cgroup.event_control  cgroup.procs  net_cls.classid
notify_on_release  tasks

/sys/fs/cgroup/net_cls/zorro:
cgroup.clone_children  cgroup.event_control  cgroup.procs  net_cls.classid
notify_on_release  tasks
```

建立完毕之后分别配置相关的cgroup，将对应cgroup产生的数据包对应到相应的分类中，配置方法：

```
[root@localhost Desktop]# echo 0x00010100 > /sys/fs/cgroup/net_cls/net_cls.classid
[root@localhost Desktop]# echo 0x00010010 > /sys/fs/cgroup/net_cls／
jerry/net_cls.classid
[root@localhost Desktop]# echo 0x00010020 > /sys/fs/cgroup/net_cls／
zorro/net_cls.classid
[root@localhost Desktop]# tc fi add dev eno16777736 parent 1: protocol ip prio 1
handle 1: cgroup
```

这里的tc命令是对filter进行操作，这里我们使用了cgroup过滤器，来实现将cgroup的数据包送到1:0分类中，细节不再解

释。对于net_cls.classid文件，我们一般echo的是一个0xAAAABBBB的值，AAAA对应class中:前面的数字，而BBBB对应后面的数字，如：0x00010100就表示这个组的数据包将被分类到1:100中，限速为30mbit，以此类推。之后我们把http服务放倒jerry组中看看效果：

```
[root@localhost Desktop]# for i in `ps ax|grep httpd|awk '{ print $1}'`;do echo $i
> /sys/fs/cgroup/net_cls/jerry/tasks;done
bash: echo: write error: No such process
[root@localhost Desktop]# cat /sys/fs/cgroup/net_cls/jerry/tasks
75733
75734
75735
75736
75737
75738
75777
75778
75779
```

测试效果：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 5118k    0     0  1162k      0  0:15:01  0:00:04  0:14:57 1162k
```

确实限速在了10mbitps。成功达到效果，再来看看放倒zorro组下：

```
[root@localhost Desktop]# for i in `ps ax|grep httpd|awk '{ print $1}'`;do echo $i
> /sys/fs/cgroup/net_cls/zorro/tasks;done
bash: echo: write error: No such process
[root@localhost Desktop]# cat /sys/fs/cgroup/net_cls/zorro/tasks
75733
75734
75735
75736
75737
75738
75777
75778
75779
```

再次测试效果：

```
zorrozou-nb:~ zorro$ curl -O http://192.168.139.136/file
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0 1024M    0 5586k    0     0  2334k      0  0:07:29  0:00:02  0:07:27 2334k
```

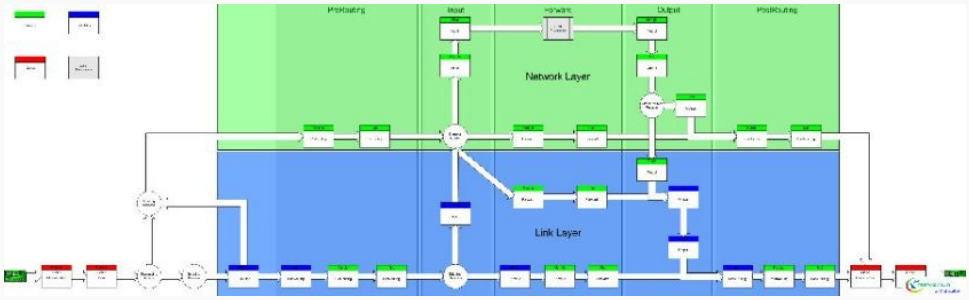
限速20mbps成功。如果想要修改对于一个分类的限速，使用如下命令即可：

```
tc cl change dev eno16777736 parent 1: classid 1:100 htb rate 100mbit
```

关于命令参数的详细解释，这里不做过多说明了。大家可以自行找帮助。

###最后

终于，我的Cgroup系列四部曲算是告一段落了。实际上Linux的Cgroup除了CPU、内存、IO和网络的资源管理以外，还有一些其它的配置，比如针对设备文件的访问控制和freezer机制等功能，但是这些功能都相对比较简单，个人认为没必要过多介绍了，大家要用的时候自己找帮助即可。
最后的最后，还是奉送一张Linux网络相关的数据包处理流程图，从这张图上大家可以清晰的看到qdisc的作用位置和其根iptables的作用关系。原图链接在此。



二月 5, 2016 / Linux / 留下评论

Cgroup – Linux的IO资源隔离

Cgroup – Linux的IO资源隔离

Hi，我是Zorro。这是我的微博地址，我会不定期在这里更新文章，如果你有兴趣，可以来关注我呦。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

本文PDF

今天我们来谈谈：

##Linux的IO隔离

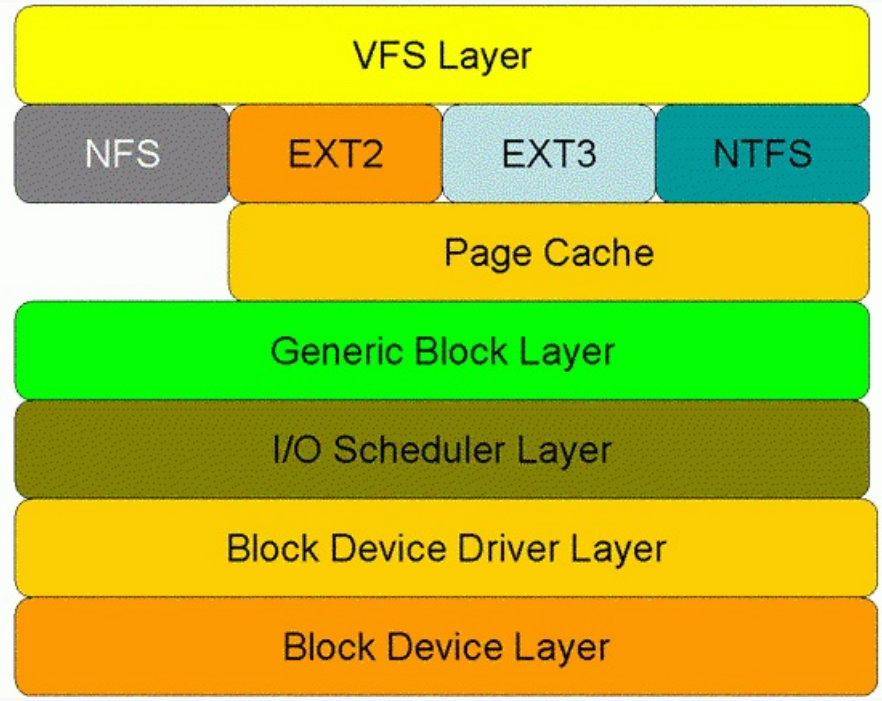
跟内存管理那部分复杂度类似，IO的资源隔离要讲清楚也是比较麻烦的。这部分内容都是这样，配置起来简单，但是要理解清楚确没那么简单。这次是跟Linux内核的IO实现有关系。对于IO的速度限制，实现思路跟CPU和内存都不一样。CPU是针对进程占用时间的比例限制，内存是空间限制，而当我们讨论IO资源隔离的时候，实际上有两个资源需要考虑，一个是空间，另一个是速度。对于空间来说，这个很简单，大不了分区就是了。现实手段中，分区、LVM、磁盘配额、目录配额等等，不同的分区管理方式，不同的文件系统都给出了很多不同的解决方案。所以，空间的限制实际上不是cgroup要解决的问题，那就是说，我们在这里要解决的问题是：如何进行IO数据传输的速度限制。

限速这件事情，现实中有很多模型、算法去解决这个问题。比如，如果我们想控制高速公路上的汽车单位时间通过率，就让收费站每隔固定时间周期只允许通过固定个数的车就好了。这是一种非常有效的控制手段－－漏斗算法。现实中这种算法可能在特定情况下会造成资源浪费以及用户的体验不好，于是又演化出令牌桶算法。这里我们不去详细分析这些算法，但是我们要知道，对io的限速基本是一个漏斗算法的限速效果。无论如何，这种限速都要有个“收费站”这样的设施来执行限速，那么对于Linux的IO体系来说，这个”收费站”建在哪里呢？于是我们就必须先来了解一下：

###Linux的IO体系

Linux的IO体系是个层级还算清晰的结构，它基本上分成了如图示这样几层：

Linux的IO体系层次结构



我们可以通过追踪一个read()系统调用来一窥这些层次的结构，当read()系统调用发生，内核首先会通过汇编指令引发一个软中断，然后根据中断传入的参数查询系统调用影射表，找到read()对应的内核调用方法名，并去执行相关调用，这个系统调用名一般情况下就是sys_read()。从此，便开始了调用在内核中处理的过程的第一步：

1. VFS层：虚拟文件系统层。由于内核要跟多种文件系统打交道，而每一种文件系统所实现的数据结构和相关方法都可能不尽相同，所以，内核抽象了这一层，专门用来适配各种文件系统，并对外提供统一操作接口。
2. 文件系统层：不同的文件系统实现自己的操作过程，提供自己特有的特征，具体不多说了，大家愿意的话自己去看代码即可。
3. 页缓存层：我们的老朋友了，如果不了解缓存是什么的，可以先来看看Linux内存资源管理部分。
4. 通用块层：由于绝大多数情况的io操作是跟块设备打交道，所以Linux在此提供了一个类似vfs层的块设备操作抽象层。下层对接各种不同属性的块设备，对上提供统一的Block IO请求标准。

5. IO调度层：因为绝大多数的块设备都是类似磁盘这样的设备，所以有必要根据这类设备的特点以及应用的不同特点来设置一些不同的调度算法和队列。以便在不同的应用环境下有针对性的提高磁盘的读写效率，这里就是大名鼎鼎的Linux电梯所起作用的地方。针对机械硬盘的各种调度方法就是在这实现的。
6. 块设备驱动层：驱动层对外提供相对比较高级的设备操作接口，往往是C语言的，而下层对接设备本身的操作方法和规范。
7. 块设备层：这层就是具体的物理设备了，定义了各种真对设备操作方法和规范。

根据这几层的特点，如果你是设计者，你会在哪里实现真对块设备的限速策略呢？6、7都是相关具体设备的，如果在这个层次提供，那就不是内核全局的功能，而是某些设备自己的feture。文件系统层也可以实现，但是如果要全局实现也是不可能的，需要每种文件系统中都实现一遍，成本太高。所以，可以实现限速的地方比较合适的是VFS、缓存层、通用块层和IO调度层。而VFS和page cache这样的机制并不是面向块设备设计的，都是做其他事情用的，虽然也在io体系中，但是并不适合用来做block io的限速。所以这几层中，最适合并且成本最低就可以实现的地方就是IO调度层和通用块层。IO调度层本身已经有队列了，我们只要在队列里面实现一个限速机制即可，但是在IO调度层实现的限速会因为不同调度算法的侧重点不一样而有很多局限性，从通用块层实现的限速，原则上就可以对几乎所有的块设备进行带宽和iops的限制。截止目前（4.3.3内核），IO限速主要实现在这两层中。

根据IO调度层和通用块层的特点，这两层分别实现了两种不同策略的IO控制策略，也是目前blkio子系统提供的两种控制策略，一个是权重比例方式的控制，另一个是针对IO带宽和IOPS的控制。

###IO调度层

我们需要先来认识一下IO调度层。这一层要解决的核心问题是，如何提高块设备IO的整体性能？这一层也主要是针对用途最广泛的机械硬盘结构而设计的。众所周知，机械硬盘的存储介质是磁介质，并且是盘状，用磁头在盘片上移动进行数据的寻址，这类似播放一张唱片。这种结构的特点是，顺序的数据读写效率比较理想，但是如果一旦对盘片有随机读写，那么大量的时间都会浪费在磁头的移动上，这时候就会导致每次IO的响应时间很长，极大的降低IO的响应速度。磁头在盘片上寻道的操作，类似电梯调度，如果在寻道的过程中，能把路过的相关磁道的数据请求都“顺便”处理掉，那么就可以在比较小影响响应速度的前提下，提高整体IO的吞吐量。所以，一个好的IO调度算法的需求就此产生。在最开始的阶段，Linux就把这个算法命名为Linux电梯算法。目前在内核中默认开启了三种算法，其实严格算应该是两种，因为第一种叫做noop，就是空操作调度算法，也就是没有任何调度操作，并不对io请求进行排序，仅仅做适当的io合并的一个fifo队列。

目前内核中默认的调度算法应该是cfq，叫做完全公平队列调度。这个调度算法人如其名，它试图给所有进程提供一个完全公平的IO操作环境。它为每个进程创建一个同步IO调度队列，并默认以时间片和请求数限定的方式分配IO资源，以此保证每个进程的IO资源占用是公平的，cfq还实现了针对进程级别的优先级调度，这里我们不去细节解释。我们在此只需要知道，既然时间片分好了，优先级实现了，那么cfq肯定是实现进程级别的权重比例分配的最好方案。内核就是这么做的，cgroup blkio的权重比例限制就是基于cfq调度器实现的。如果你要使用权重比例分配，请先确定对应的块设备的IO调度算法是cfq。

查看和修改的方法是：

```
[zorro@zorrozou-pc0 ~]$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
[zorro@zorrozou-pc0 ~]$ echo cfq > /sys/block/sda/queue/scheduler
```

cfq是通用服务器比较好的IO调度算法选择，对桌面用户也是比较好的选择。但是对于很多IO压力较大的场景就并不是很适应，尤其是IO压力集中在某些进程上的场景。因为这种场景我们需要更多的满足某个或者某几个进程的IO响应速度，而不是让所有的进程公平的使用IO，比如数据库应用。

deadline调度（最终期限调度）就是更适应这样的场景的解决方案。deadline实现了四个队列，其中两个分别处理正常read和write，按扇区号排序，进行正常io的合并处理以提高吞吐量.因为IO请求可能会集中在某些磁盘位置，这样会导致新来的请求一直被合并，于是可能会有其他磁盘位置的io请求被饿死。于是实现了另外两个处理超时read和write的队列，按请求创建时间排序，如果有超时的请求出现，就放进这两个队列，调度算法保证超时（达到最终期限时间）的队列中的请求会优先被处理，防止请求被饿死。由于deadline的特点，无疑在这里无法区分进程，也就不能实现针对进程的io资源控制。

其实不久前，内核还是默认标配四种算法，还有一种叫做as的算法（Anticipatory scheduler），预测调度算法。一个高大上的名字，搞得我一度认为Linux内核都会算命了。结果发现，无非是在基于deadline算法做io调度的之前等一小会时间，如果这段时间内有可以合并的io请求到来，就可以合并处理，提高deadline调度的在顺序读写情况下的数据吞吐量。其实这根本不是啥预测，我觉得不如叫撞大运调度算法。估计结果是不仅没有提高吞吐量，还降低了响应速度，所以内核干脆把它从默认配置里删除了。毕竟Linux的宗旨是实用。

根据以上几种io调度算法的简单分析，我们也能对各种调度算法的使用场景有一些大致的思路了。从原理上看，cfq是一种比较通用的调度算法，是一种以进程为出发点考虑的调度算法，保证大家尽量公平。deadline是一种以提高机械硬盘吞吐量为思考出发点的调度算法，只有当有io请求达到最终期限的时候才进行调度，非常适合业务比较单一并且IO压力比较重的业务，比如数据库。而noop呢？其实如果我们把我们的思考对象拓展到固态硬盘，那么你就会发现，无论cfq还是deadline，都是针对机械硬盘的结构进行的队列算法调整，而这种调整对于固态硬盘来说，完全没有意义。对于固态硬盘来说，IO调度算法越复杂，效率就越低，因为额外要处理的逻辑越多。所以，固态硬盘这种场景下，使用noop是最好的，deadline次之，而cfq由于复杂度的原因，无疑效率最低。但是，如果你想对你的固态硬盘做基于权重比例的IO限速的话，那就没啥办法了，毕竟这时候，效率并不是你的需求，要不你限速干嘛？

###通用块设备层

这层的作用我这里就不再复述了，本节其实主要想解释的是，既然这里实现了对blkio的带宽和iops的速度限制，那么有没有什么需要注意的地方？这自然是有的。首先我们还是要先来搞清楚IO中的几个概念。

一般IO：

一个正常的文件io，需要经过vfs -> buffer\page cache -> 文件系统 -> 通用块设备层 -> IO调度层 -> 块设备驱动 -> 硬件设备这所有几个层次。其实这就是一般IO。当然，不同的状态可能会有变化，比如一个进程正好open并read一个已经存在于page cache中的数据。这样的事情我们排出在外不分析。那么什么是比较特别的io呢？

Direct IO：

中文也可以叫直接IO操作，其特点是，VFS之后跳过buffer\page cache层，直接从文件系统层进行操作。那么就意味着，无论读还是写，都不会进行cache。我们基本上可以理解这样的io看起来效率要低很多，直接看到的速度就是设备的速度，并且缺少了cache层对数据的缓存之后，文件系统和数据块的操作效率直接暴露给了应用程序，块的大小会直接影响io速度。

Sync IO & **write-through**:

中文叫做同步IO操作，如果是写操作的话也叫write-through，这个操作往往容易跟上面的DIO搞混，因为看起来他们速度上差不多，但是是有本质区别的。这种方式写的数据要等待存储写入返回才能成功返回，所以跟DIO效率差不多，但是，写的数据仍然是要在cache中写入的，这样其他一般IO的程度仍然可以使用cache机制加速IO操作。所以，这里的sync的意思就是，在执行write操作的时候，让cache和存储上的数据一致。那么他跟一般IO其实一样，数据是要经过cache层的。

write-back:

既然明白了write-thuough，那么write-back就好理解了，无非就是将目前在cache中还没写回存储的脏数据写回到存储。这个名词一般指的是一个独立的过程，这个过程不是随着应用的写而发生，这往往是内核自己找个时间来单独操作的。说白了就是，应用写文件，感觉自己很快写完了，其实内核都把数据放倒cache里了，然后内核自己找时间再去写回到存储上。实际上write-back只是在一般IO的情况下，保证数据一致性的一种机制而已。

有人将IO过程中，以是否使用缓冲（缓存）的区别，将IO分成了缓存IO（Buffered IO）和直接IO（Direct io）。其实就是名词上的不同而已。这里面的buffer的含义跟内存中buffer cache有概念上的不同。实际上这里Buffered IO的含义，相当于内存中的buffer cache+page cache，就是IO经过缓存的意思。到这我们思考一个问题，如果cgroup针对IO的资源限制实现在了通用块设备层，那么将会对哪些IO操作有影响呢？其实原则上说都有影响，因为绝大多数数据都是要经过通用块设备层写入存储的，但是对于应用程序来说感受可能不一样。在一般IO的情况下，应用程序很可能很快的就写完了数据（在数据量小于缓存空间的情况下），然后去做其他事情了。这时应用程序感受不到自己被限速了，而内核在处理write-back的阶段，由于没有相关page cache中的inode是属于那个cgroup的信息记录，所以所有的page cache的回写只能放到cgroup的root组中进行限制，而不能在其他cgroup中进行限制，因为root组的cgroup一般是不做限制的，所以就相当于目前的cgroup的blkio对buffered IO是没有限速支持的。这个功能将在使用了unified-hierarchy体系的cgroup v2中的部分文件系统（ext系列）已经得到得到支持，目前这个功能还在开发中，据说将在4.5版本的内核中正式发布。

而在Sync IO和Direct IO的情况下，由于应用程序写的数据是不经过缓存层的，所以能直接感受到速度被限制，一定要等到整个数据按限制好的速度写完或者读完，才能返回。这就是当前cgroup的blkio限制所能起作用的环境限制。了解了这个之后，我们就可以来看：

##blkio配置方法

###权重比例分配

我们这次直接使用命令行的方式对cgroup进行操作。在我们的系统上，我们现在想创建两个cgroup组，一个叫test1，一个叫test2。我们想让这两个组的进程在对/dev/sdb，设备号为8:16的这个磁盘进行读写的时候按权重比例进行io资源的分配。具体配置方法如下：

首先确认系统上已经mount了相关的cgroup目录：

```
[root@zorrozou-pc0 ~]# ls /sys/fs/cgroup/blkio/
blkio.io_merged        blkio.io_service_bytes_recursive  blkio.io_wait_time
blkio.sectors            blkio.throttle.read_iops_device   blkio.weight       tasks
blkio.io_merged_recursive  blkio.io_serviced
blkio.io_wait_time_recursive  blkio.sectors_recursive
blkio.throttle.write_bps_device   blkio.weight_device
blkio.io_queued        blkio.io_serviced_recursive       blkio.leaf_weight
blkio.throttle.io_service_bytes  blkio.throttle.write_iops_device
cgroup.clone_children
blkio.io_queued_recursive  blkio.io_service_time         blkio.leaf_weight_device
blkio.throttle.io_serviced       blkio.time                cgroup.procs
blkio.io_service_bytes     blkio.io_service_time_recursive   blkio.reset_stats
blkio.throttle.read_bps_device   blkio.time_recursive          notify_on_release
```

然后创建两个针对blkio的cgroup

```
[root@zorrozou-pc0 ~]# mkdir /sys/fs/cgroup/blkio/test1
[root@zorrozou-pc0 ~]# mkdir /sys/fs/cgroup/blkio/test2
```

相关目录下会自动产生相关配置项：

```
[root@zorrozou-pc0 ~]# ls /sys/fs/cgroup/blkio/test{1,2}
/sys/fs/cgroup/blkio/test1:
blkio.io_merged        blkio.io_service_bytes_recursive  blkio.io_wait_time
blkio.sectors            blkio.throttle.read_iops_device   blkio.weight       tasks
blkio.io_merged_recursive  blkio.io_serviced
blkio.io_wait_time_recursive  blkio.sectors_recursive
blkio.throttle.write_bps_device   blkio.weight_device
blkio.io_queued        blkio.io_serviced_recursive       blkio.leaf_weight
blkio.throttle.io_service_bytes  blkio.throttle.write_iops_device
cgroup.clone_children
blkio.io_queued_recursive  blkio.io_service_time         blkio.leaf_weight_device
blkio.throttle.io_serviced       blkio.time                cgroup.procs
blkio.io_service_bytes     blkio.io_service_time_recursive   blkio.reset_stats
blkio.throttle.read_bps_device   blkio.time_recursive          notify_on_release

/sys/fs/cgroup/blkio/test2:
blkio.io_merged        blkio.io_service_bytes_recursive  blkio.io_wait_time
blkio.sectors            blkio.throttle.read_iops_device   blkio.weight       tasks
blkio.io_merged_recursive  blkio.io_serviced
blkio.io_wait_time_recursive  blkio.sectors_recursive
blkio.throttle.write_bps_device   blkio.weight_device
blkio.io_queued        blkio.io_serviced_recursive       blkio.leaf_weight
blkio.throttle.io_service_bytes  blkio.throttle.write_iops_device
cgroup.clone_children
blkio.io_queued_recursive  blkio.io_service_time         blkio.leaf_weight_device
blkio.throttle.io_serviced       blkio.time                cgroup.procs
blkio.io_service_bytes     blkio.io_service_time_recursive   blkio.reset_stats
blkio.throttle.read_bps_device   blkio.time_recursive          notify_on_release
```

之后我们就可以进行限制了。针对cgroup进行权重限制的配置有**blkio.weight**，是单纯针对cgroup进行权重配置的，还有**blkio.weight_device**可以针对设备单独进行限制，我们都来试试。首先我们想设置test1和test2使用任何设备的io权重比例都是1:2：

```
[root@zorrozou-pc0 zorro]# echo 100 > /sys/fs/cgroup/blkio/test1/blkio.weight
[root@zorrozou-pc0 zorro]# echo 200 > /sys/fs/cgroup/blkio/test2/blkio.weight
```

注意权重设置的取值范围为：10-1000。然后我们来写一个测试脚本：

```
#!/bin/bash

testfile1=/home/test1
testfile2=/home/test2

if [ -e $testfile1 ]
then
    rm -rf $testfile1
fi

if [ -e $testfile2 ]
then
    rm -rf $testfile2
fi
```

```
sync
echo 3 > /proc/sys/vm/drop_caches

cgexec -g blkio:test1 dd if=/dev/zero of=$testfile1 oflag=direct bs=1M count=1023 &

cgexec -g blkio:test2 dd if=/dev/zero of=$testfile2 oflag=direct bs=1M count=1023 &
```

我们dd的时候使用的是direct标记，在这使用sync和不加任何标记的话都达不到效果。因为权重限制是基于cfq实现，cfq要标记进程，而buffered IO都是内核同步，无法标记进程。使用iotop查看限制效果：

```
[root@zorrozou-pc0 zorro]# iotop -b -n1|grep direct
 1519 be/4 root        0.00 B/s  110.00 M/s  0.00 % 99.99 % dd if=/dev/zero
of=/home/test2 oflag=direct bs=1M count=1023
 1518 be/4 root        0.00 B/s   55.00 M/s  0.00 % 99.99 % dd if=/dev/zero
of=/home/test1 oflag=direct bs=1M count=1023
```

却是达到了1:2比例限速的效果。此时对于磁盘读取的限制效果也一样，具体测试用例大家可以自己编写。读取的时候要注意，仍然要保证读取的文件不在page cache中，方法就是：echo 3 > /proc/sys/vm/drop_caches。因为在page cache中的数据已经在内存里了，直接修改是直接改内存中的内容，只有write-back的时候才会经过cfq。

我们再来试一下针对设备的权重分配，请注意设备号的填写格式：

```
[root@zorrozou-pc0 zorro]# echo "8:16 400" >
/sys/fs/cgroup/blkio/test1/blkio.weight_device
[root@zorrozou-pc0 zorro]# echo "8:16 200" >
/sys/fs/cgroup/blkio/test2/blkio.weight_device

[root@zorrozou-pc0 zorro]# iotop -b -n1|grep direct
 1800 be/4 root        0.00 B/s  102.24 M/s  0.00 % 99.99 % dd if=/dev/zero
of=/home/test1 oflag=direct bs=1M count=1023
 1801 be/4 root        0.00 B/s   51.12 M/s  0.00 % 99.99 % dd if=/dev/zero
of=/home/test2 oflag=direct bs=1M count=1023
```

我们会发现，这时权重确实是按照最后一次的设置，test1和test2变成了2:1的比例，而不是1:2了。这里要说明的就是，注意blkio.weight_device的设置会覆盖blkio.weight的设置，因为前者的设置精确到了设备，Linux在这里的策略是，越精确越优先。

###读写带宽和iops限制

针对读写带宽和iops的限制都是绝对值限制，所以我们不用两个cgroup做对比了。我们就设置test1的写带宽速度为1M/s:

```
[root@zorrozou-pc0 zorro]# echo "8:16  1048576" >
/sys/fs/cgroup/blkio/test1/blkio.throttle.write_bps_device

[root@zorrozou-pc0 zorro]# sync
[root@zorrozou-pc0 zorro]# echo 3 > /proc/sys/vm/drop_caches

[root@zorrozou-pc0 zorro]# cgexec -g blkio:test1 dd if=/dev/zero of=/home/test
oflag=direct count=1024 bs=1M
```

```
^C21+0 records in
21+0 records out
22020096 bytes (22 MB) copied, 21.012 s, 1.0 MB/s
```

此时不用dd命令执行完，稍等一下中断执行就能看到速度确实限制在了1M／s。写的同时，iostat显示为：

```
 [zorro@zorrozou-pc0 ~]$ iostat -x 1
Linux 4.3.3-2-ARCH (zorrozou-pc0.tencent.com)   2016年01月15日     _x86_64_    (4
CPU)


avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       0.50    0.00    0.50   25.13    0.00   73.87

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     1.00
1000.00    0.00 1000.00 1000.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
        1.25    0.00    0.50   24.81    0.00   73.43

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     5.00    0.00    6.00     0.00  1060.00   353.33     0.06
9.33    0.00    9.33   9.50   5.70
dm-0              0.00     0.00    0.00   10.00     0.00  1060.00   212.00     1.08
109.00    0.00  109.00 100.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       1.25    0.00    1.00   24.44    0.00   73.32

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     1.00
993.00    0.00  993.00 1000.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
```

```
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       1.50    0.25    0.75   24.50    0.00   73.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    0.00    1.00     0.00  1024.00  2048.00     1.00
1000.00    0.00 1000.00 1000.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
```

可以看到写的速度确实为1024wkB/s左右。我们再来试试读，先创建一个大文件，此处没有限速：

```
[root@zorrozou-pc0 zorro]# dd if=/dev/zero of=/home/test oflag=direct count=1024
bs=1M
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 10.213 s, 105 MB/s
```

然后进行限速设置并确认：

```
[root@zorrozou-pc0 zorro]# sync
[root@zorrozou-pc0 zorro]# echo 3 > /proc/sys/vm/drop_caches
[root@zorrozou-pc0 zorro]# echo "8:16  1048576" >
/sys/fs/cgroup/blkio/test1/blkio.throttle.read_bps_device
[root@zorrozou-pc0 zorro]# cgexec -g blkio:test1 dd if=/home/test of=/dev/null
iflag=direct count=1024 bs=1M
^C15+0 records in
14+0 records out
14680064 bytes (15 MB) copied, 15.0032 s, 978 kB/s
```

iostat结果：

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       0.75    0.00    0.75   24.63    0.00   73.88

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    2.00    0.00  1024.00     0.00  1024.00     0.00
0.00    0.00    0.00   0.00   0.00
```

```
dm-0              0.00     0.00    2.00    0.00  1024.00     0.00  1024.00     1.65
825.00  825.00    0.00 500.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       0.75    0.00    0.50   24.87    0.00   73.87

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     2.00    0.00    2.00     0.00    16.00    16.00     0.02
10.00    0.00   10.00  10.00   2.00
sdb               0.00     0.00    2.00    0.00  1024.00     0.00  1024.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    2.00    0.00  1024.00     0.00  1024.00     1.65
825.00  825.00    0.00 500.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
```

最后是iops的限制，我就不废话了，直接上命令执行结果：

```
 [root@zorrozou-pc0 zorro]# echo "8:16  20" >
/sys/fs/cgroup/blkio/test1/blkio.throttle.write_iops_device
[root@zorrozou-pc0 zorro]# rm /home/test
[root@zorrozou-pc0 zorro]# sync
[root@zorrozou-pc0 zorro]# echo 3 > /proc/sys/vm/drop_caches
[root@zorrozou-pc0 zorro]# cgexec -g blkio:test1 dd if=/dev/zero of=/home/test
oflag=direct count=1024 bs=1M
^C121+0 records in
121+0 records out
126877696 bytes (127 MB) copied, 12.0576 s, 10.5 MB/s

[zorro@zorrozou-pc0 ~]$ iostat -x 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       0.50    0.00    0.25   24.81    0.00   74.44

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    0.00   20.00     0.00 10240.00  1024.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    0.00   20.00     0.00 10240.00  1024.00     2.00
100.00    0.00  100.00  50.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
       0.75    0.00    0.25   24.31    0.00   74.69
```

```
Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdb               0.00     0.00    0.00   20.00     0.00 10240.00  1024.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-0              0.00     0.00    0.00   20.00     0.00 10240.00  1024.00     2.00
100.00    0.00  100.00  50.00 100.00
dm-1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-2              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
dm-3              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
```

iops的读限制我就不在废话了，大家可以自己做实验测试一下。

##其他相关文件

###针对权重比例限制的相关文件

blkio.leaf_weight[_device]

其意义等同于blkio.weight[_device]，主要表示当本cgroup中有子cgroup的时候，本cgroup的进程和子cgroup中的进程所分配的资源比例是怎么样的。举个例子说吧，假设有一组cgroups的关系是这样的：

```
          root
       /    |   \
      A     B    leaf
     400   200   200
```

leaf就表示root组下的进程所占io资源的比例。
此时A组中的进程可以占用的比例为：400/（400+200+200） * 100% = 50%
B为：200/（400+200+200） * 100% = 25%
而root下的进程为：200/（400+200+200） * 100% = 25%

blkio.time

统计相关设备的分配给本组的io处理时间，单位为ms。权重就是依据此时间比例进行分配的。

blkio.sectors

统计本cgroup对设备的读写扇区个数。

blkio.io_service_bytes

统计本cgroup对设备的读写字节个数。

blkio.io_serviced

统计本cgroup对设备的读写操作个数。

blkio.io_service_time

统计本cgroup对设备的各种操作时间。时间单位是ns。

blkio.io_wait_time

统计本cgroup对设备的各种操作的等待时间。时间单位是ns。

blkio.io_merged

统计本cgroup对设备的各种操作的合并处理次数。

blkio.io_queued

统计本cgroup对设备的各种操作的当前正在排队的请求个数。

blkio.*_recursive

这一堆文件是相对应的不带_recursive的文件的递归显示版本，所谓递归的意思就是，它会显示出包括本cgroup在内的衍生cgroup的所有信息的总和。

###针对带宽和iops限制的相关文件

blkio.throttle.io_serviced

统计本cgroup对设备的读写操作个数。

blkio.throttle.io_service_bytes

统计本cgroup对设备的读写字节个数。

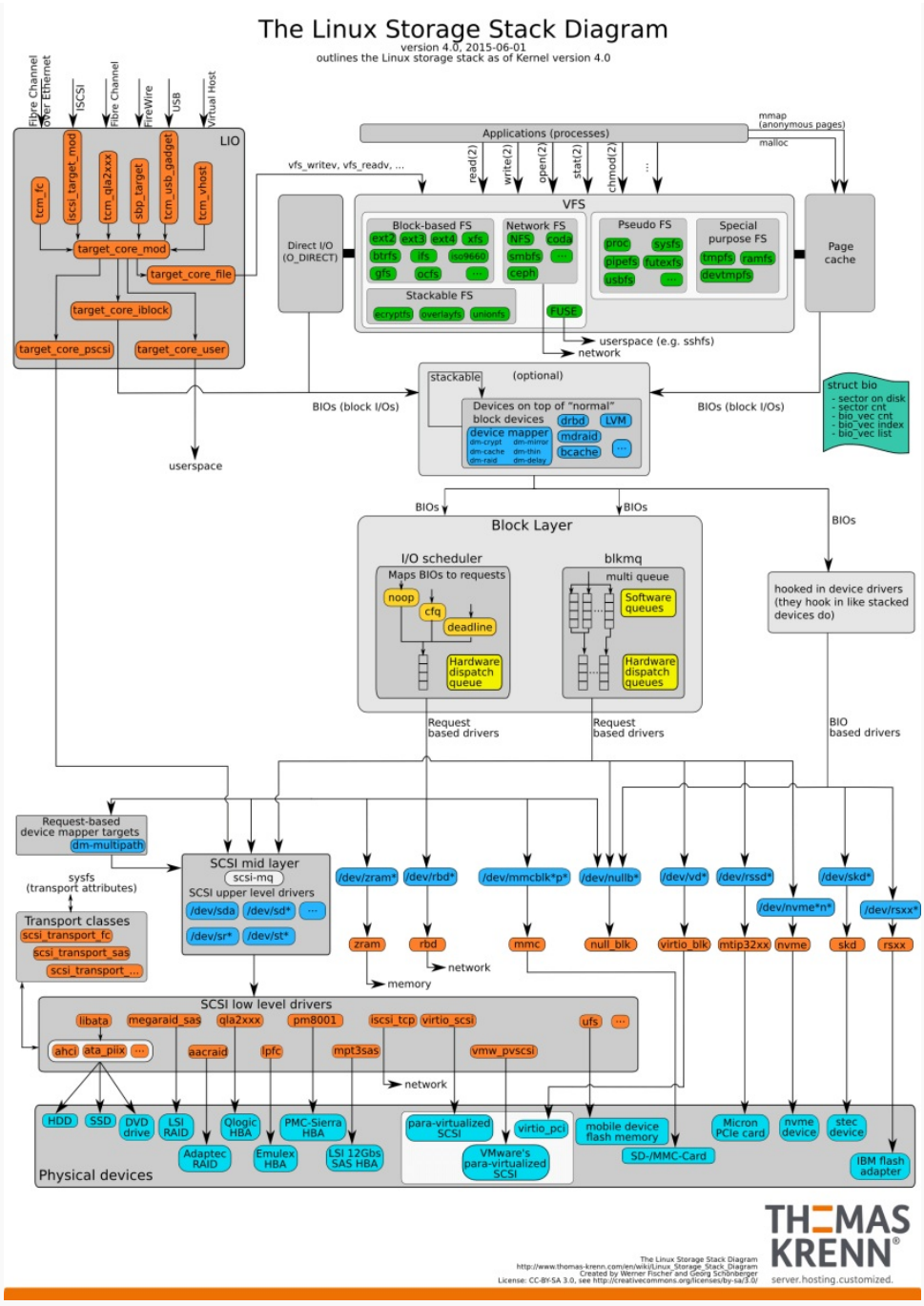
blkio.reset_stats

对本文件写入一个int可以对以上所有文件的值置零，重新开始累计。

##最后

其实一直纠结要不要写这部分IO隔离的文档，因为看上去意义不大。一则因为目前IO隔离似乎工作场景里用的不多，二则因为目前内核中这部分代码还在进行较大变化的调整，还要继续加入其它功能。从内核Linux 3.16版本之后，cgroup调整方向，开始了基于unified hierarchy架构的cgroup v2。IO部分在write-back部分进行了较大调整，加入了对buffered IO的资源限制。我们这次系统环境为ArchLinux，内核版本为Linux 4.3.3，虽然环境中的unified hierarchy的开发版本功能已经部分支持了，但是思考再三还是暂时不加入到此文档中。新架构的cgoup v2预计会跟随Linux 4.5一起推出，到时候我们再做详细分析好了。

附送一张更详细的Linux 4.0 IO协议栈框架图



一月 22, 2016 / Linux / 有1条评论

Cgroup – Linux内存资源管理

#Cgroup – Linux内存资源管理

Hi，我是Zorro。这是我的微博地址，我会不定期在这里更新文章，如果你有兴趣，可以来关注我呦。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

本文PDF

在聊cgroup的内存限制之前，我们有必要先来讲解一下：

##Linux内存管理基础知识

###free命令

无论从任何角度看，Linux的内存管理都是一坨麻烦的事情，当然我们也可以用一堆、一片、一块、一筐来形容这个事情，但是毫无疑问，用一坨来形容它简直恰当无比。在理解它之前，我甚至不会相信精妙的和恶心可以同时形容同一件事情，是的，在我看来它就是这样的。其实我只是做个铺垫，让大家明白，我们下面要讲的内容，绝不是一个成体系的知识，所以，学习起来也确实很麻烦。甚至，我写这个技术文章之前一度考虑了很久该怎么写？从哪里开始写？思考了半天，还是不能免俗，我们无奈，仍然先从free命令说起：

```
[root@zorrozou-pc ~]# free
             total       used       free     shared    buffers     cached
Mem:     131904480    6681612  125222868          0     478428    4965180
-/+ buffers/cache:    1238004  130666476
Swap:      2088956          0    2088956
```

这个命令几乎是每一个使用过Linux的人必会的命令，但越是这样的命令，似乎真正明白的人越少（我是说比例越少）。一般情况下，对此命令的理解可以分这几个阶段：

1. 我擦，内存用了好多，6个多G，可是我什么都没有运行啊？为什么会这样？Linux好占内存。
2. 嗯，根据我专业的眼光看出来，内存才用了1G多点，还有很多剩余内存可用。buffers/cache占用的较多，说明系统中有进程曾经读写过文件，但是不要紧，这部分内存是当空闲来用的。
3. free显示的是这样，好吧我知道了。神马？你问我这些内存够不够，我当然不知道啦！我特么怎么知道你程序怎么写的？

如果你的认识在第一种阶段，那么请你继续补充关于Linux的buffers／cache的知识。如果你处在第二阶段，好吧，你已经是个老手了，但是需要提醒的是，上帝给你关上一扇门的同时，肯定都会给你放一条狗的。是的，Linux的策略是：内存是用来用的，而不是用来看的。但是，只要是用了，就不是没有成本的。有什么成本，凭你对buffer/cache的理解，应该可以想的出来。一般我比较认同第三种情况，一般光凭一个free命令的显示，是无法判断出任何有价值的信息的，我们需要结合业务的场景以及其他输出综合判断目前遇到的问题。当然也可能这种人给人的第一感觉是他很外行，或者他真的是外行。

无论如何，free命令确实给我门透露了一些有用的信息，比如内存总量，剩余多少，多少用在了buffers／cache上，Swap用了多少，如果你用了其它参数还能看到一些其它内容，这里不做一一列举。那么这里又引申出另一些概念，什么是buffer？什么是cache？什么是swap？由此我们就直接引出另一个命令：

```
[root@zorrozou-pc ~]# cat /proc/meminfo
MemTotal:       131904480 kB
```

```
MemFree:        125226660 kB
Buffers:          478504 kB
Cached:          4966796 kB
SwapCached:            0 kB
Active:          1774428 kB
Inactive:        3770380 kB
Active(anon):     116500 kB
Inactive(anon):     3404 kB
Active(file):    1657928 kB
Inactive(file):  3766976 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2088956 kB
SwapFree:        2088956 kB
Dirty:               336 kB
Writeback:             0 kB
AnonPages:         99504 kB
Mapped:            20760 kB
Shmem:             20604 kB
Slab:             301292 kB
SReclaimable:     229852 kB
SUnreclaim:        71440 kB
KernelStack:        3272 kB
PageTables:         3320 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    68041196 kB
Committed_AS:     352412 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      493196 kB
VmallocChunk:   34291062284 kB
HardwareCorrupted:     0 kB
AnonHugePages:     49152 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      194816 kB
DirectMap2M:     3872768 kB
DirectMap1G:    132120576 kB
```

以上显示的内容都是些什么鬼？

其实这个问题的答案也是另一个问题的答案，即：Linux是如何使用内存的？了解清楚这个问题是很有必要的，因为只有先知道了Linux如何使用内存，我们在能知道内存可以如何限制，以及，做了限制之后会有什么问题？我们在此先例举出几个常用概念的意义：

内存，作为一种相对比较有限的资源，内核在考虑其管理时，无非应该主要从以下出发点考虑：

1. 内存够用时怎么办？
2. 内存不够用时怎么办？

在内存够用时，内核的思路是，如何尽量提高资源的利用效率，以加快系统整体响应速度和吞吐量？于是内存作为一个CPU和I／O之间的大buffer的功能就呼之欲出了。为此，内核设计了以下系统来做这个功能：

###Buffers／Cached

buffer和cache是两个在计算机技术中被用滥的名词，放在不通语境下会有不同的意义。在内存管理中，我们需要特别澄清

一下，这里的buffer指Linux内存的：Buffer cache。这里的cache指Linux内存中的：Page cache。翻译成中文可以叫做缓冲区缓存和页面缓存。在历史上，它们一个（buffer）被用来当成对io设备写的缓存，而另一个（cache）被用来当作对io设备的读缓存，这里的io设备，主要指的是块设备文件和文件系统上的普通文件。但是现在，它们的意义已经不一样了。在当前的内核中，page cache顾名思义就是针对内存页的缓存，说白了就是，如果有内存是以page进行分配管理的，都可以使用page cache作为其缓存来使用。当然，不是所有的内存都是以页（page）进行管理的，也有很多是针对块（block）进行管理的，这部分内存使用如果要用到cache功能，则都集中到buffer cache中来使用。（从这个角度出发，是不是buffer cache改名叫做block cache更好？）然而，也不是所有块（block）都有固定长度，系统上块的长度主要是根据所使用的块设备决定的，而页长度在X86上无论是32位还是64位都是4k。

而明白了这两套缓存系统的区别，也就基本可以理解它们究竟都可以用来做什么了。

什么是page cache

Page cache主要用来作为文件系统上的文件数据的缓存来用，尤其是针对当进程对文件有read／write操作的时候。如果你仔细想想的话，作为可以映射文件到内存的系统调用：mmap是不是很自然的也应该用到page cache？如果你再仔细想想的话，malloc会不会用到page cache？

以上提出的问题都请自己思考，本文档不会给出标准答案。

在当前的实现里，page cache也被作为其它文件类型的缓存设备来用，所以事实上page cache也负责了大部分的块设备文件的缓存工作。

什么是buffer cache

Buffer cache则主要是设计用来在系统对块设备进行读写的时候，对块进行数据缓存的系统来使用。但是由于page cache也负责块设备文件读写的缓存工作，于是，当前的buffer cache实际上要负责的工作比较少。这意味着某些对块的操作会使用buffer cache进行缓存，比如我们在格式化文件系统的时候。

一般情况下两个缓存系统是一起配合使用的，比如当我们对一个文件进行写操作的时候，page cache的内容会被改变，而buffer cache则可以用来将page标记为不同的缓冲区，并记录是哪一个缓冲区被修改了。这样，内核在后续执行脏数据的回写（writeback）时，就不用将整个page写回，而只需要写回修改的部分即可。

有搞大型系统经验的人都知道，缓存就像万金油，只要哪里有速度差异产生的瓶颈，就可以在哪里抹。但是其成本之一就是，需要维护数据的一致性。内存缓存也不例外，内核需要维持其一致性，在脏数据产生较快或数据量较大的时候，缓存系统整体的效率一样会下降，因为毕竟脏数据写回也是要消耗IO的。这个现象也会表现在这样一种情况下，就是当你发现free的时候，内存使用量较大，但是去掉了buffer／cache的使用之后剩余确很多。以一般的理解，都会认为此时进程如果申请内存，内核会将buffer／cache占用的内存当成空闲的内存分给进程，这是没错的。但是其成本是，在分配这部分已经被buffer／cache占用的内存的时候，内核会先对其上面的脏数据进行写回操作，保证数据一致后才会清空并分给进程使用。如果此时你的进程是突然申请大量内存，而且你的业务是一直在产生很多脏数据（比如日志），并且系统没有及时写回的时候，此时系统给进程分配内存的效率会很慢，系统IO也会很高。那么此时你还以为buffer／cache可以当空闲内存使用么？

> *思考题：Linux什么时候会将脏数据写回到外部设备上？这个过程如何进行人为干预？*

这足可以证明一点，以内存管理的复杂度，我们必须结合系统上的应用状态来评估系统监控命令所给出的数据，才是做评估的正确途径。如果你不这样做，那么你就可以轻而易举的得出“Linux系统好烂啊！“这样的结论。也许此时，其实是你在这个系统上跑的应用很烂的缘故导致的问题。

接下来，当内存不够用的时候怎么办？

我们好像已经分析了一种内存不够用的状态，就是上述的大量buffer／cache把内存几乎占满的情况。但是基于Linux对内存的使用原则，这不算是不够用，但是这种状态导致IO变高了。我们进一步思考，假设系统已经清理了足够多的buffer／cache分给了内存，而进程还在嚷嚷着要内存咋办？

此时内核就要启动一系列手段来让进程**尽量**在此时能够正常的运行下去。

请注意我在这说的是一种异常状态！我之所以要这样强调是因为，很多人把内存用满了当称一种正常状态。他们认为，当我的业务进程在内存使用到压力边界的情况下，系统仍然需要保证让业务进程有正常的状态！这种想法显然是缘木求鱼了。另外我还要强调一点，系统提供的是内存管理的机制和手段，而内存用的好不好，主要是业务进程的事情，责任不能本末倒置。

###谁该SWAP？

首先是Swap机制。Swap是交换技术，这种技术是指，当内存不够用的时候，我们可以选择性的将一块磁盘、分区或者一个文件当成交换空间，将内存上一些临时用不到的数据放到交换空间上，以释放内存资源给急用的进程。

哪些数据可能会被交换出去呢？从概念上判断，如果一段内存中的数据被经常访问，那么就不应该被交换到外部设备上，因为这样的数据如果交换出去的话会导致系统响应速度严重下降。内存管理需要将内存区分为活跃的（Active）和不活跃的（Inactive），再加上一个进程使用的用户空间内存映射包括文件影射（file）和匿名影射（anon），所以就包括了Active（anon）、Inactive（anon）、Active（file）和Inactive（file）。你说神马？啥是文件影射（file）和匿名影射（anon）？好吧，我们可以这样简单的理解，匿名影射主要是诸如进程使用malloc和mmap的MAP_ANONYMOUS的方式申请的内存，而文件影射就是使用mmap影射的文件系统上的文件，这种文件系统上的文件既包括普通的文件，也包括临时文件系统（tmpfs）。这意味着，Sys V的IPC和POSIX的IPC（IPC是进程间通信机制，在这里主要指共享内存，信号量数组和消息队列）都是通过文件影射方式体现在用户空间内存中的。这两种影射的内存都会被算成进程的RSS，但是也一样会被显示在cache的内存计数中，在相关cgroup的另一项统计中，共享内存的使用和文件缓存（file cache）也都会被算成是cgroup中的cache使用的总量。这个统计显示的方法是：

```
[root@zorrozou-pc ~]# cat /cgroup/memory/memory.stat
cache 94429184
rss 102973440
rss_huge 50331648
mapped_file 21512192
swap 0
pgpgin 656572990
pgpgout 663474908
pgfault 2871515381
pgmajfault 1187
inactive_anon 3497984
active_anon 120524800
inactive_file 39059456
active_file 34484224
unevictable 0
hierarchical_memory_limit 9223372036854775807
hierarchical_memsw_limit 9223372036854775807
total_cache 94429184
total_rss 102969344
total_rss_huge 50331648
total_mapped_file 21520384
total_swap 0
total_pgpgin 656572990
total_pgpgout 663474908
total_pgfault 2871515388
```

```
total_pgmajfault 1187
total_inactive_anon 3497984
total_active_anon 120524800
total_inactive_file 39059456
total_active_file 34484224
total_unevictable 0
```

好吧，说了这么半天终于联系到一个cgroup的内存限制相关的文件了。在这需要说明的是，你之所以看见我废话这么多，是因为我们必须先基本理清楚Linux系统的内存管理方式，才能进一步对cgroup中的内存限制做规划使用，否则同样的名词会有很多的歧义。就比如我们在观察某一个cgroup中的cache占用数据的时候，我们究竟该怎么理解它？真的把它当成空闲空间来看么？

我们撒的有点远，回过头来说说这些跟Swap有什么关系？还是刚才的问题，什么内容该被从内存中交换出去呢？文件cache是一定不需要的，因为既然是cache，就意味着它本身就是硬盘上的文件（当然你现在应该知道了，它也不仅仅只有文件），那么如果是硬盘上的文件，就不用swap交换出去，只要写回脏数据，保持数据一致之后清除就可以了，这就是刚才说过的缓存清楚机制。但是我们同时也要知道，并不是所有被标记为cache的空间都能被写回硬盘的(是的，比如共享内存)。那么能交换出去内存应该主要包括有Inactive（anon）这部分内存。主要注意的是，内核也将共享内存作为计数统计近了Inactive（anon）中去了（是的，共享内存也可以被Swap）。还要补充一点，如果内存被mlock标记加锁了，则也不会交换，这是对内存加mlock锁的唯一作用。刚才我们讨论的这些计数，很可能会随着Linux内核的版本改变而产生变化，但是在比较长的一段时间内，我们可以这样理解。

我们基本搞清了swap这个机制的作用效果，那么既然swap是内部设备和外部设备的数据拷贝，那么加一个缓存就显得很有必要，这个缓存就是swapcache，在memory.stat文件中，swapcache是跟anon page被一起记录到rss中的，但是并不包含共享内存。另外再说明一下，HugePages也是不会交换的。显然，当前的swap空间用了多少，总共多少，这些我们也可以在相关的数据中找到答案。

> *以上概念中还有一些名词大家可能并不清楚其含义，比如RSS或HugePages。请自行查资料补上这些知识。为了让大家真的理解什么是RSS，请思考ps aux命令中显示的VSZ，RSS和cat /proc/pid/smaps中显示的：PSS这三个进程占用内存指标的差别？*

###何时SWAP？
搞清楚了谁该swap，那么还要知道什么时候该swap。这看起来比较简单，内存耗尽而且cache也没什么可以回收的时候就应该触发swap。其实现实情况也没这么简单，实际上系统在内存压力可能不大的情况下也会swap，这种情况并不是我们今天要讨论的范围。

> *思考题：除了内存被耗尽的时候要swap，还有什么时候会swap？如何调整内核swap的行为？如何查看当前系统的swap空间有哪些？都是什么类型？什么是swap权重？swap权重有什么意义？*

其实绝大多数场景下，什么时候swap并不重要，而swap之后的事情相对却更重要。大多数的内存不够用，只是临时不够用，比如并发突增等突发情况，这种情况的特点是时间持续短，此时swap机制作为一种临时的中转措施，可以起到对业务进程的保护作用。因为如果没有swap，内存耗尽的结果一般都是触发oom killer，会杀掉此时积分比较高的进程。如果更严重的话，内存不够用还会触发进程D状态死锁，这一般发生在多个进程同时要申请内存的时候，此时oom killer机制也可能会失效，因为需要被干掉的积分比较高的进程很可能就是需要申请内存的进程，而这个进程本身因为正在争抢内存而导致陷入D状态，那么此时kill就可能是对它无效的。

但是swap也不是任何时候都有很好的保护效果。如果内存申请是长期并大量的，那么交换出去的数据就会因为长时间驻留在外部设备上，导致进程调用这段内存的几率大大增加，当进程很频繁的使用它已经被交换出去的内存时，就会让整个系统处在io繁忙的状态，此时进程的响应速度会严重下降，导致整个系统夯死。对于系统管理员来说，这种情况是完全不能接受的，因为故障之后的第一要务是赶紧恢复服务，但是swap频繁使用的IO繁忙状态会导致系统除了断电重启之外，没有其

它可靠手段可以让系统从这种状态中恢复回来，所以这种情况是要尽力避免的。此时，如果有必要，我们甚至可以考虑不用swap，哪怕内存过量使用被oom，或者进程D状态都是比swap导致系统卡死的情况更好处理的状态。如果你的环境需求是这样的，那么可以考虑关闭swap。

###进程申请内存的时候究竟会发生什么？

刚才我们从系统宏观的角度简要说明了一下什么是buffer／cache以及swap。下面我们从一个更加微观的角度来把一个内存申请的过程以及相关机制什么时候触发给串联起来。本文描述的过程是基于Linux 3.10内核版本的，Linux 4.1基本过程变化不大。如果你想确认在你的系统上究竟是什么样子，请自行翻阅相关内核代码。

进程申请内存可能用到很多种方法，最常见的就是malloc和mmap。但是这对于我们并不重要，因为无论是malloc还是mmap，或是其他的申请内存的方法，都不会真正的让内核去给进程分配一个实际的物理内存空间。真正会触发分配物理内存的行为是**缺页异常**。

缺页异常就是我们可以在memory.stat中看到的total_pgfault，这种异常一般分两种，一种叫major fault，另一种叫minor fault。这两种异常的主要区别是，进程所请求的内存数据是否会引发磁盘io？如果会引发，就是一个majfault，如果不引发，那就是minfault。就是说如果产生了major fault，这个数据基本上就意味着已经被交换到了swap空间上。

缺页异常的处理过程大概可以整理为以下几个路径：

首先检查要访问的虚拟地址是否合法，如果合法则继续查找和分配一个物理页，步骤如下：

1. 检查发生异常的虚拟地址是不是在物理页表中不存在？如果是，并且是匿名影射，则申请置0的匿名影射内存，此时也有可能是影射了某种虚拟文件系统，比如共享内存，那么就去影射相关的内存区，或者发生COW写时复制申请新内存。如果是文件影射，则有两种可能，一种是这个影射区是一个page cache，直接将相关page cache区影射过来即可，或者COW新内存存放需要影射的文件内容。如果page cache中不存在，则说明这个区域已经被交换到swap空间上，应该去处理swap。
2. 如果页表中已经存在需要影射的内存，则检查是否要对内存进行写操作，如果不写，那就直接复用，如果要写，就发生COW写时复制，此时的COW跟上面的处理过程不完全相同，在内核中，这里主要是通过do_wp_page方法实现的。

如果需要申请新内存，则都会通过alloc_page_vma申请新内存，而这个函数的核心方法是__alloc_pages_nodemask，也就是Linux内核著名的内存管理系统**伙伴系统**的实现。

分配过程先会检查空闲页表中有没有页可以申请，实现方法是：get_page_from_freelist，我们并不关心正常情况，分到了当然一切ok。更重要的是异常处理，如果空闲中没有，则会进入__alloc_pages_slowpath方法进行处理。这个处理过程的主逻辑大概这样：

1. 唤醒kswapd进程，把能换出的内存换出，让系统有内存可用。
2. 继续检查看看空闲中是否有内存。有了就ok，没有继续下一步：
3. 尝试清理page cache，清理的时候会将进程置为D状态。如果还申请不到内存则：
4. 启动oom killer干掉一些进程释放内存，如果这样还不行则：
5. 回到步骤1再来一次！

当然以上逻辑要符合一些条件，但是这一般都是系统默认的状态，比如，你必须启用oom killer机制等。另外这个逻辑中有很多其它状态与本文无关，比如检查内存水印、检查是否是高优先级内存申请等等，当然还有关于numa节点状态的判断处理，我没有一一列出。另外，以上逻辑中，不仅仅只有清理cache的时候会使进程进入D状态，还有其它逻辑也会这样做。这就是为什么在内存不够用的情况下，oom killer有时也不生效，因为可能要干掉的进程正好陷入这个逻辑中的D状态了。

以上就是内存申请中，大概会发生什么的过程。当然，我们这次主要是真对本文的重点cgroup内存限制进行说明，当我们处理限制的时候，更多需要关心的是当内存超限了会发生什么？对边界条件的处理才是我们这次的主题，所以我并没有对正常申请到的情况做细节说明，也没有对用户态使用malloc什么时候使用sbrk还是mmap来申请内存做出细节说明，毕竟那是程序正常状态的时候的事情，后续可以另写一个内存优化的文章主要讲解那部分。

下面我们该进入正题了：

##Cgroup内存限制的配置

当限制内存时，我们最好先想清楚如果内存超限了会发生什么？该怎么处理？业务是否可以接受这样的状态？这就是为什么我们在讲如何限制之前说了这么多基础知识的“废话”。其实最简单的莫过于如何进行限制了，我们的系统环境还是沿用上一次讲解CPU内存隔离的环境，使用cgconfig和cgred服务进行cgroup的配置管理。还是创建一个zorro用户，对这个用户产生的进程进行内存限制。基础配置方法不再多说，如果不知道的请参考这个文档。

环境配置好之后，我们就可以来检查相关文件了。内存限制的相关目录根据cgconfig.config的配置放在了/cgroup/memory目录中，如果你跟我做了一样的配置，那么这个目录下的内容应该是这样的：

```
[root@zorrozou-pc ~]# ls /cgroup/memory/
cgroup.clone_children  memory.failcnt                   memory.kmem.slabinfo
memory.kmem.usage_in_bytes  memory.memsw.limit_in_bytes      memory.oom_control
memory.usage_in_bytes  shrek
cgroup.event_control   memory.force_empty               memory.kmem.tcp.failcnt
memory.limit_in_bytes       memory.memsw.max_usage_in_bytes  memory.pressure_level
memory.use_hierarchy   tasks
cgroup.procs           memory.kmem.failcnt
memory.kmem.tcp.limit_in_bytes      memory.max_usage_in_bytes
memory.memsw.usage_in_bytes      memory.soft_limit_in_bytes           zorro
cgroup.sane_behavior   memory.kmem.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes  memory.meminfo
memory.move_charge_at_immigrate  memory.stat                  notify_on_release
jerry                  memory.kmem.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes      memory.memsw.failcnt        memory.numa_stat
memory.swappiness           release_agent
```

其中，zorro、jerry、shrek都是目录概念跟cpu隔离的目录树结构类似。相关配置文件内容：

```
[root@zorrozou-pc ~]# cat /etc/cgconfig.conf    mount {
    cpu = /cgroup/cpu;
    cpuset  = /cgroup/cpuset;
    cpuacct = /cgroup/cpuacct;
    memory  = /cgroup/memory;
    devices = /cgroup/devices;
    freezer = /cgroup/freezer;
    net_cls = /cgroup/net_cls;
    blkio   = /cgroup/blkio;
}

group zorro {
    cpu {
        cpu.shares = 6000;
#       cpu.cfs_quota_us = "600000";
    }
    cpuset {
#       cpuset.cpus = "0-7,12-19";
```

```
#       cpuset.mems = "0-1";
    }
    memory {
    }
}
```

配置中添加了一个真对memory的空配置项，我们稍等下再给里面添加配置。

```
[root@zorrozou-pc ~]# cat /etc/cgrules.conf
zorro       cpu,cpuset,cpuacct,memory   zorro
jerry       cpu,cpuset,cpuacct,memory   jerry
shrek       cpu,cpuset,cpuacct,memory   shrek
```

文件修改完之后记得重启相关服务：

```
[root@zorrozou-pc ~]# service cgconfig restart
[root@zorrozou-pc ~]# service cgred restart
```

让我们继续来看看真对内存都有哪些配置参数：

```
[root@zorrozou-pc ~]# ls /cgroup/memory/zorro/
cgroup.clone_children  memory.kmem.failcnt
memory.kmem.tcp.limit_in_bytes      memory.max_usage_in_bytes
memory.memsw.usage_in_bytes      memory.soft_limit_in_bytes
cgroup.event_control   memory.kmem.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes  memory.meminfo
memory.move_charge_at_immigrate  memory.stat                 notify_on_release
cgroup.procs           memory.kmem.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes      memory.memsw.failcnt
memory.numa_stat                 memory.swappiness           tasks
memory.failcnt         memory.kmem.slabinfo           memory.kmem.usage_in_bytes
memory.memsw.limit_in_bytes      memory.oom_control
memory.usage_in_bytes
memory.force_empty     memory.kmem.tcp.failcnt         memory.limit_in_bytes
memory.memsw.max_usage_in_bytes  memory.pressure_level
memory.use_hierarchy
```

首先我们已经认识了memory.stat文件了，这个文件内容不能修改，它实际上是输出当前cgroup相关内存使用信息的。常见的数据及其含义我们刚才也已经说过了，在此不再复述。

###cgroup内存限制

memory.memsw.limit_in_bytes:内存+swap空间使用的总量限制。

memory.limit_in_bytes：内存使用量限制。

这两项的意义很清楚了，如果你决定在你的cgroup中关闭swap功能，可以把两个文件的内容设置为同样的值即可。至于为什么相信大家都能想清楚。

###OOM控制

memory.oom_control:内存超限之后的oom行为控制。
这个文件中有两个值：

oom_kill_disable 0

默认为0表示打开oom killer，就是说当内存超限时会触发干掉进程。如果设置为1表示关闭oom killer，此时内存超限不会触发内核杀掉进程。而是将进程夯住（hang／sleep），实际上内核中就是将进程设置为D状态，并且将相关进程放到一个叫做OOM-waitqueue的队列中。这时的进程可以kill杀掉。如果你想继续让这些进程执行，可以选择这样几个方法：

1. 增加内存，让进程有内存可以继续申请。
2. 杀掉一些进程，让本组内有内存可用。
3. 把一些进程移到别的cgroup中，让本cgroup内有内存可用。
4. 删除一些tmpfs的文件，就是占用内存的文件，比如共享内存或者其它会占用内存的文件。

说白了就是，此时只有当cgroup中有更多内存可以用了，在OOM-waitqueue队列中被挂起的进程就可以继续运行了。

under_oom 0

这个值只是用来看的，它表示当前的cgroup的状态是不是已经oom了，如果是，这个值将显示为1。我们就是通过设置和监测这个文件中的这两个值来管理cgroup内存超限之后的行为的。在默认场景下，如果你使用了swap，那么你的cgroup限制内存之后最常见的异常效果是IO变高，如果业务不能接受，我们一般的做法是关闭swap，那么cgroup内存oom之后都会触发kill掉进程，如果我们用的是LXC或者Docker这样的容器，那么还可能干掉整个容器。当然也经常会因为kill进程的时候因为进程处在D状态，而导致整个Docker或者LXC容器根本无法被杀掉。至于原因，在前面已经说的很清楚了。当我们遇到这样的困境时该怎么办？一个好的办法是，关闭oom killer，让内存超限之后，进程挂起，毕竟这样的方式相对可控。此时我们可以检查under_oom的值，去看容器是否处在超限状态，然后根据业务的特点决定如何处理业务。我推荐的方法是关闭部分进程或者重启掉整个容器，因为可以想像，容器技术所承载的服务应该是在整体软件架构上有容错的业务，典型的场景是web服务。容器技术的特点就是生存周期短，在这样的场景下，杀掉几个进程或者几个容器，都应该对整体服务的稳定性影响不大，而且容器的启动速度是很快的，实际上我们应该认为，容器的启动速度应该是跟进程启动速度可以相媲美的。你的业务会因为死掉几个进程而表现不稳定么？如果不会，请放心的干掉它们吧，大不了很快再启动起来就是了。但是如果你的业务不是这样，那么请根据自己的情况来制定后续处理的策略。

当我们进行了内存限制之后，内存超限的发生频率要比使用实体机更多了，因为限制的内存量一般都是小于实际物理内存的。所以，使用基于内存限制的容器技术的服务应该多考虑自己内存使用的情况，尤其是内存超限之后的业务异常处理应该如何让服务受影响的程度降到更低。在系统层次和应用层次一起努力，才能使内存隔离的效果达到最好。

###内存资源审计

memory.memsw.usage_in_bytes:当前cgroup的内存＋swap的使用量。

memory.usage_in_bytes:当前cgroup的内存使用量。

memory.max_usage_in_bytes:cgroup的最大内存使用量。

memory.memsw.max_usage_in_bytes:cgroup最大的内存＋swap的使用量。

这些文件都是只读的，用来查看相关状态信息，只能看不能改。

如果你的内核配置打开了CONFIG_MEMCG_KMEM选项的话，那么可以看到当前cgroup的内核内存使用的限制和状态统计信息，他们都是以memory.kmem开头的文件。你可以通过memory.kmem.limit_in_bytes来限制内核使用的内存大小，通过memory.kmem.slabinfo来查看内核slab分配器的状态。现在还能通过memory.kmem.tcp开头的文件来限制cgroup中使用tcp协议的内存资源使用和状态查看。

所有名字中有failcnt的文件里面的值都是相关资源超限的次数的计数，可以通过echo 0将这些计数重置。如果你的服务器是NUMA架构的话，可以通过memory.numa_stat这个文件来查看cgroup中的NUMA相关状态。memory.swappiness跟/proc/sys/vm/swappiness的概念一致，用来调整cgroup使用swap的状态，如果大家认真做了本文前面的思考题的话，应该知道这个文件是干嘛的，本文不会详细解释关于swappiness的细节算法，以后将在性能调整系列文章中详细解释相关参数。

###内存软限制以及内存超卖

memory.soft_limit_in_bytes:内存软限制。

如果超过了memory.limit_in_bytes所定义的限制，那么进程会被oom killer干掉或者被暂停，这相当于硬限制，因为进程无法申请超过自身cgroup限制的内存，但是软限制确是可以突破的。我们假定一个场景，如果你的实体机上有四个cgroup，实体机的内存总量是64G，那么一般情况我们会考虑给每个cgroup限制到16G内存。但是现实情况并不会这么理想，首先实体机上其他进程和内核会占用部分内存，这将导致实际上每个cgroup都不会真的有16G内存可用，如果四个cgroup都尽量占用内存的话，他们可能谁都不会到达内存的上限触发超限的行为，这可能将导致进程都抢不到内存而被饿死。类似的情况还可能发上在内存超卖的环境中，比如，我们仍然只有64G内存，但是确开了8个cgroup，每个都限制了16G内存。这样每个cgroup分配的内存之和达到了128G，但是实际内存量只有64G。这种情况是出于绝大多数应用可能不会占用满所有的内存来考虑的，这样就可以把本来属于它的那份内存“借用”给其它cgroup。以上这样的情况都会出现类似的问题，就是，如果全局内存已经耗尽了，但是某些cgroup还没达到他的内存使用上限，而它们此时如果要申请内存的话，此时该从哪里回收内存？如果我们配置了memory.soft_limit_in_bytes，那么内核将去回收那些内存超过了这个软限制的cgroup的内存，尽量缩减它们的内存占用达到软限制的量以下，以便让没有达到软限制的cgroup有内存可以用。当然，在没有这样的内存竞争以及没有达到硬限制的情况下，软限制是不会生效的。还有就是，软限制的起作用时间可能会比较长，毕竟内核要平衡多个cgroup的内存使用。

根据软限制的这些特点，我们应该明白如果想要软限制生效，应该把它的值设置成小于硬限制。

###进程迁移时的内存charge

memory.move_charge_at_immigrate:打开或者关闭进程迁移时的内存记账信息。

进程可以在多个cgroup之间切换，所以内存限制必须考虑当发生这样的切换时，进程进入的新cgroup中记录的内存使用量是重新从0累计还是把原来cgroup中的信息迁移过来？当这个开关设置为0的时候是关闭这个功能，相当于不累计之前的信息，默认是1，迁移的时候要在新的cgroup中累积（charge）原来信息，并把旧group中的信息给uncharge掉。如果新cgroup中没有足够的空间容纳新来的进程，首先内核会在cgroup内部回收内存，如果还是不够，就会迁移失败。

###内存压力通知机制

最后，内存的资源隔离还提供了一种压力通知机制。当cgoup内的内存使用量达到某种压力状态的时候，内核可以通过eventfd的机制来通知用户程序，这个通知是通过**cgroup.event_control**和**memory.pressure_level**来实现的。使用方法是：

使用eventfd()创建一个eventfd，假设叫做efd，然后open()打开memory.pressure_level的文件路径，产生一个另一个fd，我们暂且叫它cfd，然后将这两个fd和我们要关注的内存压力级别告诉内核，让内核帮我们关注条件是否成立，通知方式就

是把以上信息按这样的格式:”<event_fd：efd> “写入cgroup.event_control。然后就可以去等着efd是否可读了，如果能读出信息，则代表内存使用已经触发相关压力条件。

压力级别的level有三个：

“low”：表示内存使用已经达到触发内存回收的压力级别。

“medium”：表示内存使用压力更大了，已经开始触发swap以及将活跃的cache写回文件等操作了。

“critical”：到这个级别，就意味着内存已经达到上限，内核已经触发oom killer了。

程序从efd读出的消息内容就是这三个级别的关键字。我们可以通过这个机制，建立一个内存压力管理系统，在内存达到相应级别的时候，触发响应的管理策略，来达到各种自动化管理的目的。

下面给出一个监控程序的例子：

```
#include <assert.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <sys/eventfd.h>

#define USAGE_STR "Usage: cgroup_event_listener <path-to-control-file> <args>"

int main(int argc, char **argv)
{
    int efd = -1;
    int cfd = -1;
    int event_control = -1;
    char event_control_path[PATH_MAX];
    char line[LINE_MAX];
    int ret;

    if (argc != 3)
        errx(1, "%s", USAGE_STR);

    cfd = open(argv[1], O_RDONLY);
    if (cfd == -1)
        err(1, "Cannot open %s", argv[1]);

    ret = snprintf(event_control_path, PATH_MAX, "%s/cgroup.event_control",
            dirname(argv[1]));
    if (ret >= PATH_MAX)
        errx(1, "Path to cgroup.event_control is too long");

    event_control = open(event_control_path, O_WRONLY);
    if (event_control == -1)
        err(1, "Cannot open %s", event_control_path);

    efd = eventfd(0, 0);
    if (efd == -1)
        err(1, "eventfd() failed");
```

```
    ret = snprintf(line, LINE_MAX, "%d %d %s", efd, cfd, argv[2]);
    if (ret >= LINE_MAX)
        errx(1, "Arguments string is too long");

    ret = write(event_control, line, strlen(line) + 1);
    if (ret == -1)
        err(1, "Cannot write to cgroup.event_control");

    while (1) {
        uint64_t result;

        ret = read(efd, &result, sizeof(result));
        if (ret == -1) {
            if (errno == EINTR)
                continue;
            err(1, "Cannot read from eventfd");
        }
        assert(ret == sizeof(result));

        ret = access(event_control_path, W_OK);
        if ((ret == -1) && (errno == ENOENT)) {
            puts("The cgroup seems to have removed.");
            break;
        }

        if (ret == -1)
            err(1, "cgroup.event_control is not accessible any more");

        printf("%s %s: crossed\n", argv[1], argv[2]);
    }

    return 0;
}
```

##最后

Linux的内存限制要说的就是这么多了，当我们限制了内存之后，相对于使用实体机，实际上对于应用来说可用内存更少了，所以业务会相对更经常地暴露在内存资源紧张的状态下。相对于虚拟机（kvm，xen），多个cgroup之间是共享内核的，我们可以从内存限制的角度思考一些关于“容器”技术相对于虚拟机和实体机的很多特点：

1. 内存更紧张，应用的内存泄漏会导致相对更严重的问题。
2. 容器的生存周期时间更短，如果实体机的开机运行时间是以年计算的，那么虚拟机则是以月计算的，而容器应该跟进程的生存周期差不多，顶多以天为单位。所以，容器里面要跑的应用应该可以被经常重启。
3. 当有多个cgroup（容器）同时运行时，我们不能再以实体机或者虚拟机对资源的使用的理解来规划整体运营方式，我们需要更细节的理解什么是cache，什么是swap，什么是共享内存，它们会被统计到哪些资源计数中？在内核并不冲突的环境，这些资源都是独立给某一个业务使用的，在理解上即使不是很清晰，也不会造成歧义。但是在cgroup中，我们需要彻底理解这些细节，才能对遇到的情况进行预判，并规划不同的处理策略。

也许我们还可以从中得到更多的理解，大家一起来想喽？

一月 22, 2016 / Linux / 留下评论

Cgroup – 从CPU资源隔离说起

Cgroup – 从CPU资源隔离说起

Hi，我是Zorro。这是我的微博地址，我会不定期在这里更新文章，如果你有兴趣，可以来关注我呦。

本文有配套视频演示，一起服用效果更佳。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

今天我们来谈谈：

##什么是Cgroup？

> *cgroups，其名称源自控制组群（control groups）的简写，是Linux内核的一个功能，用来限制，控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）。*
>
> *–引自维基百科:cgroup*

引用官方说法总是那么冰冷的让人不好理解，所以我还是稍微解释一下：

一个正在运行着服务的计算机系统，跟我们高中上课的情景还是很相似的。如果把系统中的每个进程理解为一个同学的话，那么班主任就是操作系统的核心（kernel），负责管理班里的同学。而cgroup，就是班主任控制学生行为的一种手段，所以，它起名叫control groups。

既然是一种控制手段，那么cgroup能控制什么呢？当然是资源啦！对于计算机来说，资源大概可以分成以下几个部分：

- 计算资源
- 内存资源
- io资源
- 网络资源

这就是我们常说的内核四大子系统。当我们学习内核的时候，我们也基本上是围绕这四大子系统进行研究。
我们今天要讨论的，主要是cgroup是如何对系统中的CPU资源进行隔离和分配的。其他资源的控制，我们以后有空再说喽。

##如何看待CPU资源？

*由于进程和线程在Linux的CPU调度看来没啥区别，所以本文后续都会用**进程**这个名词来代表内核的调度对象，一般来讲也*

包括线程

如果要分配资源，我们必须先搞清楚这个资源是如何存在的，或者说是如何组织的。我想CPU大家都不陌生，我们都在系统中用过各种工具查看过CPU的使用率，比如说以下这个命令和它的输出：

```
[zorro@zorrozou-pc0 ~]$ mpstat -P ALL 1 1
Linux 4.2.5-1-ARCH (zorrozou-pc0)   2015年12月22日 _x86_64_     (4 CPU)mt

16时01分08秒  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
16时01分09秒  all    0.25    0.00    0.25    0.00    0.00    0.00    0.00    0.00
0.00   99.50
16时01分09秒    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
16时01分09秒    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
16时01分09秒    2    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
16时01分09秒    3    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00   99.00

Average:     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
Average:     all    0.25    0.00    0.25    0.00    0.00    0.00    0.00    0.00
0.00   99.50
Average:       0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
Average:       1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
Average:       2    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
Average:       3    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00   99.00
```

显示的内容具体什么意思，希望大家都能了解，我就不在这细解释了。根据显示内容我们知道，这个计算机有4个cpu核心，目前的cpu利用率几乎是0，就是说系统整体比较闲。

从这个例子大概可以看出，我们对cpu资源的评估一般有两个观察角度：

- 核心个数
- 百分比

目前的计算机基本都是多核甚至多cpu系统，一个服务器上存在几个到几十个cpu核心的情况都很常见。所以，从这个角度看，cgroup应该提供一种手段，可以给进程们指定它们可以占用的cpu核心，以此来做到cpu计算资源的隔离。
百分比这个概念我们需要多解释一下：这个百分比究竟是怎么来的呢？难道每个cpu核心的计算能力就像一个带刻度表的水杯一样？一个进程要占用就会占用到它的一定刻度么？

当然不是啦！这个cpu的百分比是按时间比率计算的。基本思路是：一个CPU一般就只有两种状态，要么被占用，要么不被占用。当有多个进程要占用cpu的时候，那么操作系统在一个cpu核心上是进行分时处理的。比如说，我们把一秒钟分成1000份，那么每一份就是1毫秒，假设现在有5个进程都要用cpu，那么我们就让它们5个轮着使用，比如一人一毫秒，那么1秒过后，每个进程只占用了这个CPU的200ms，使用率为20%。整体cpu使用比率为100%。
同理，如果只有一个进程占用，而且它只用了300ms，那么在这一秒的尺度看来，cpu的占用时间是30％。于是显示出来的状态就是占用30%的CPU时间。

这就是内核是如何看待和分配计算资源的。当然实际情况要比这复杂的多，但是基本思路就是这样。Linux内核是通过CPU调度器CFS－－完全公平调度器对CPU的时间进行调度的，由于本文的侧重点是cgroup而不是CFS，对这个题目感兴趣的同学可以到这里进一步学习。CFS是内核可以实现真对CPU资源隔离的核心手段，因此，理解清楚CFS对理解清楚CPU资源隔离会有很大的帮助。

##如何隔离CPU资源？

根据CPU资源的组织形式，我们就可以理解cgroup是如何对CPU资源进行隔离的了。

无非也是两个思路，一个是分配核心进行隔离，另一个是分配CPU使用时间进行隔离。

再介绍如何做隔离之前，我们先来介绍一下我们的实验系统环境：没有特殊情况，我们的实验环境都是一台24核心、128G内存的服务器，上面安装的系统可以认为是Centos7.

###搭建测试环境

我们将使用cgconfig服务和cgred服务对cgroup进行配置和使用。我们将配置两个group，一个叫zorro，另一个叫jerry。它们分别也是系统上的两个账户，其中zorro用户所运行的进程都默认在zorro group中进行限制，jerry用户所运行的进程都放到jerry group中进行限制。配置文件内容和配置方法如下：

本文并不对以下配置方法的具体含义做解释，大家只要知道如此配置可以达到相关试验环境要求即可。如果大家对配置的细节感兴趣，可以自行查找相关资料进行学习。

首先添加两个用户，zorro和jerry：

```
[root@zorrozou-pc ~]# useradd zorro
[root@zorrozou-pc ~]# useradd jerry
```

修改/etc/cgrules.conf，添加两行内容：

```
[root@zorrozou-pc ~]# cat /etc/cgrules.conf
zorro       cpu,cpuacct zorro
jerry       cpu,cpuacct jerry
```

修改/etc/cgconfig.conf，添加以下内容：

```
[root@zorrozou-pc ~]# cat /etc/cgconfig.conf
mount {
    cpuset  = /cgroup/cpuset;
    cpu = /cgroup/cpu;
    cpuacct = /cgroup/cpuacct;
    memory  = /cgroup/memory;
    devices = /cgroup/devices;
    freezer = /cgroup/freezer;
    net_cls = /cgroup/net_cls;
    blkio   = /cgroup/blkio;
}
```

```
group zorro {
    cpuset {
        cpuset.cpus = "1,2";
    }
}

group jerry {
    cpuset {
        cpuset.cpus = "3,4";
    }
}
```

重启cgconfig服务和cgred服务：

```
[root@zorrozou-pc ~]# service cgconfig restart
[root@zorrozou-pc ~]# service cgred restart
```

根据上面的配置，我们给zorro组合jerry组分别配置了cpuset的隔离设置，那么在cgroup的相关目录下应该出现相关组的配置文件：
本文中所出现的组的含义，如无特殊说明都是对应cgroup的控制组，而非用户组身份。
我们可以通过检查相关目录内容来检查一下环境是否配置完成：

```
[root@zorrozou-pc ~]# ls /cgroup/cpuset/{zorro,jerry}
/cgroup/cpuset/jerry:
cgroup.clone_children  cpuset.cpu_exclusive  cpuset.mem_exclusive
cpuset.memory_pressure     cpuset.mems                      cpuset.stat
cgroup.event_control   cpuset.cpuinfo        cpuset.mem_hardwall
cpuset.memory_spread_page  cpuset.sched_load_balance        notify_on_release
cgroup.procs           cpuset.cpus           cpuset.memory_migrate
cpuset.memory_spread_slab  cpuset.sched_relax_domain_level  tasks

/cgroup/cpuset/zorro:
cgroup.clone_children  cpuset.cpu_exclusive  cpuset.mem_exclusive
cpuset.memory_pressure     cpuset.mems                      cpuset.stat
cgroup.event_control   cpuset.cpuinfo        cpuset.mem_hardwall
cpuset.memory_spread_page  cpuset.sched_load_balance        notify_on_release
cgroup.procs           cpuset.cpus           cpuset.memory_migrate
cpuset.memory_spread_slab  cpuset.sched_relax_domain_level  tasks
```

至此，我们的实验环境已经搭建完成。

###测试用例设计
无论是针对CPU核心的隔离还是针对CPU时间的隔离，我们都需要一个可以消耗大量的CPU运算资源的程序来进行测试，考虑到我们是一个多CPU核心的环境，所以我们的测试用例一定也是一个可以并发使用多个CPU核心的计算型测试用例。针对这个需求，我们首先设计了一个使用多线程并发进行筛质数的简单程序。这个程序可以打印出从100010001到100020000数字范围内的质数有哪些。并发48个工作线程从一个共享的count整型变量中取数进行计算。程序源代码如下：

```
#include <pthread.h>
```

```
#include <stdio.h>
#include <stdlib.h>

#define NUM 48
#define START 100010001
#define END 100020000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int count = 0;

void *prime(void *p)
{
    int n, i, flag;

    while (1) {
            if (pthread_mutex_lock(&mutex) != 0) {
                    perror("pthread_mutex_lock()");
                    pthread_exit(NULL);
            }
            while (count == 0) {
                    if (pthread_cond_wait(&cond, &mutex) != 0) {
                            perror("pthread_cond_wait()");
                            pthread_exit(NULL);
                    }
            }
            if (count == -1) {
                    if (pthread_mutex_unlock(&mutex) != 0) {
                            perror("pthread_mutex_unlock()");
                            pthread_exit(NULL);
                    }
                    break;
            }
            n = count;
            count = 0;
            if (pthread_cond_broadcast(&cond) != 0) {
                    perror("pthread_cond_broadcast()");
                    pthread_exit(NULL);
            }
            if (pthread_mutex_unlock(&mutex) != 0) {
                    perror("pthread_mutex_unlock()");
                    pthread_exit(NULL);
            }
            flag = 1;
            for (i=2;i<n/2;i++) {
                    if (n%i == 0) {
                            flag = 0;
                            break;
                    }
            }
            if (flag == 1) {
                    printf("%d is a prime form %d!\n", n, pthread_self());
            }
    }
    pthread_exit(NULL);
}


int main(void)
{
    pthread_t tid[NUM];
    int ret, i;

    for (i=0;i<NUM;i++) {
            ret = pthread_create(&tid[i], NULL, prime, NULL);
```

```
            if (ret != 0) {
                    perror("pthread_create()");
                    exit(1);
            }
    }

    for (i=START;i<END;i+=2) {
            if (pthread_mutex_lock(&mutex) != 0) {
                    perror("pthread_mutex_lock()");
                    pthread_exit(NULL);
            }
            while (count != 0) {
                    if (pthread_cond_wait(&cond, &mutex) != 0) {
                            perror("pthread_cond_wait()");
                            pthread_exit(NULL);
                    }
            }
            count = i;
            if (pthread_cond_broadcast(&cond) != 0) {
                    perror("pthread_cond_broadcast()");
                    pthread_exit(NULL);
            }
            if (pthread_mutex_unlock(&mutex) != 0) {
                    perror("pthread_mutex_unlock()");
                    pthread_exit(NULL);
            }
    }

    if (pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock()");
            pthread_exit(NULL);
    }
    while (count != 0) {
            if (pthread_cond_wait(&cond, &mutex) != 0) {
                    perror("pthread_cond_wait()");
                    pthread_exit(NULL);
            }
    }
    count = -1;
    if (pthread_cond_broadcast(&cond) != 0) {
            perror("pthread_cond_broadcast()");
            pthread_exit(NULL);
    }
    if (pthread_mutex_unlock(&mutex) != 0) {
            perror("pthread_mutex_unlock()");
            pthread_exit(NULL);
    }

    for (i=0;i<NUM;i++) {
            ret = pthread_join(tid[i], NULL);
            if (ret != 0) {

                    perror("pthread_join()");
                    exit(1);
            }
    }

    exit(0);
}
```

我们先来看一下这个程序在不做限制的情况下的执行效果和执行时间：

```
[root@zorrozou-pc ~/test]# time ./prime_thread
......

100019603 is a prime form 2068363008!
100019471 is a prime form 1866938112!
100019681 is a prime form 1934079744!
100019597 is a prime form 1875330816!
100019701 is a prime form 2059970304!
100019657 is a prime form 1799796480!
100019761 is a prime form 1808189184!
100019587 is a prime form 1824974592!
100019659 is a prime form 2076755712!
100019837 is a prime form 1959257856!
100019923 is a prime form 2034792192!
100019921 is a prime form 1908901632!
100019729 is a prime form 1850152704!
100019863 is a prime form -2109106432!
100019911 is a prime form -2125891840!
100019749 is a prime form 2101933824!
100019879 is a prime form 2026399488!
100019947 is a prime form 1942472448!
100019693 is a prime form 1917294336!
100019683 is a prime form 2051577600!
100019873 is a prime form 2110326528!
100019929 is a prime form -2134284544!
100019977 is a prime form 1892116224!

real    0m8.945s
user    3m32.095s
sys 0m0.235s



[root@zorrozou-pc ~]# mpstat -P ALL 1
11:21:51     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
11:21:52     all   99.92    0.00    0.08    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       0  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       2  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       3  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       4   99.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       5  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       6  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       7  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       8  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52       9  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      10  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      11   99.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      12  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
```

```
11:21:52      13  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      14  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      15  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      16   99.01    0.00    0.00    0.00    0.99    0.00    0.00    0.00
0.00    0.00
11:21:52      17  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      18  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      19  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      20  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      21  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      22  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
11:21:52      23  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
```

经过多次测试，程序执行时间基本稳定：

```
[root@zorrozou-pc ~/test]# time ./prime_thread &> /dev/null

real    0m8.953s
user    3m31.950s
sys 0m0.227s
[root@zorrozou-pc ~/test]# time ./prime_thread &> /dev/null

real    0m8.932s
user    3m31.984s
sys 0m0.231s
[root@zorrozou-pc ~/test]# time ./prime_thread &> /dev/null

real    0m8.954s
user    3m31.794s
sys 0m0.224s
```

所有相关环境都准备就绪，后续我们将在此程序的基础上进行各种隔离的测试。

###针对CPU核心进行资源隔离
针对CPU核心进行隔离，其实就是把要运行的进程绑定到指定的核心上运行，通过让不同的进程占用不同的核心，以达到运算资源隔离的目的。其实对于Linux来说，这种手段并不新鲜，也并不是在引入cgroup之后实现的，早在内核使用O1调度算法的时候，就已经支持通过taskset命令来绑定进程的cpu核心了。

好的，废话少说，我们来看看这在cgroup中是怎么配置的。

其实通过刚才的/etc/cgconfig.conf配置文件的内容，我们已经配置好了针对不同的组占用核心的设置，来回顾一下：

```
group zorro {
    cpuset {
```

```
        cpuset.cpus = "1,2";
    }
}
```

这段配置内容就是说，将zorro组中的进程都放在编号为1，2的cpu核心上运行。这里要说明的是，cpu核心的编号一般是从0号开始的。24个核心的服务器编号范围是从0-23.我们可以通过查看/proc/cpuinfo的内容来确定相关物理cpu的个数和核心的个数。我们截取一段来看一下：

```
[root@zorrozou-pc ~/test]# cat /proc/cpuinfo
processor   : 23
vendor_id   : GenuineIntel
cpu family  : 6
model       : 63
model name  : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping    : 2
microcode   : 0x2b
cpu MHz     : 2599.968
cache size  : 15360 KB
physical id : 1
siblings    : 12
core id     : 5
cpu cores   : 6
apicid      : 27
initial apicid  : 27
fpu     : yes
fpu_exception   : yes
cpuid level : 15
wp      : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr
pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips    : 4796.38
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

其中*processor : 23*就是核心编号，说明我们当前显示的是这个服务器上的第24个核心，*physical id : 1*表示的是这个核心所在的物理cpu是哪个。这个编号也是从0开始，表示这个核心在第二个物理cpu上。那就意味着，我这个服务器是一个双物理cpu的服务器，那就可能意味着我们的系统时NUMA架构。另外还有一个要注意的是*core id : 5*这个子段，这里面隐含着一个可能的含义：你的服务器是否开启了超线程。众所周知，开启了超线程的服务器，在系统看来，一个核心会编程两个核心来看待。那么我们再确定一下是否开了超线程，可以grep一下：

```
[root@zorrozou-pc ~/test]# cat /proc/cpuinfo |grep -e "core id" -e "physical id"
physical id : 0
core id     : 0
physical id : 0
core id     : 1
physical id : 0
core id     : 2
physical id : 0
```

```
core id     : 3
physical id : 0
core id     : 4
physical id : 0
core id     : 5
physical id : 1
core id     : 0
physical id : 1
core id     : 1
physical id : 1
core id     : 2
physical id : 1
core id     : 3
physical id : 1
core id     : 4
physical id : 1
core id     : 5
physical id : 0
core id     : 0
physical id : 0
core id     : 1
physical id : 0
core id     : 2
physical id : 0
core id     : 3
physical id : 0
core id     : 4
physical id : 0
core id     : 5
physical id : 1
core id     : 0
physical id : 1
core id     : 1
physical id : 1
core id     : 2
physical id : 1
core id     : 3
physical id : 1
core id     : 4
physical id : 1
core id     : 5
```

这个内容显示出我的服务器是开启了超线程的，因为有同一个*physical id : 1*的*core id : 5*可能出现两次，那么就说明这个物理cpu上的5号核心在系统看来出现了2个，那么肯定意味着开了超线程。

> *我在此要强调超线程这个事情，因为在一个开启了超线程的服务器上运行我们当前的测试用例是很可能得不到预想的结果的。因为从原理上看，超线程技术虽然使cpu核心变多了，但是在本测试中并不能反映出相应的性能提高。我们后续会通过cpuset的资源隔离先来说明一下这个问题，然后在后续的测试中，我们将采用一些手段规避这个问题。*

我们先通过一个cpuset的配置来反映一下超线程对本测试的影响，顺便学习一下cgroup的cpuset配置方法。

1. 不绑定核心测试：

将/etc/cgconfig.conf文件中zorro组相关配置修改为以下状态，之后重启cgconfig服务：

```
group zorro {
```

```
    cpuset {
        cpuset.cpus = "0-23";
        cpuset.mems = "0-1";
    }
}

[root@zorrozou-pc ~]# service cgconfig restart
```

切换用户身份到zorro，并察看zorro组的配置：

```
[root@zorrozou-pc ~]# su - zorro
[zorro@zorrozou-pc ~]$ cat /cgroup/cpuset/zorro/cpuset.cpus
0-23
```

zorro用户对应的进程已经绑定在0-23核心上执行，我们看一下执行结果：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m8.956s
user    3m31.990s
sys 0m0.246s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m8.944s
user    3m31.956s
sys 0m0.247s
```

执行速度跟刚才一样，这相当于没绑定的情况。下面，我们对zorro组的进程绑定一半的cpu核心进行测试，先测试绑定0-11号核心，将*cpuset.cpus = "0-23"*改为*cpuset.cpus = "0-11"*。

> *请注意每次修改完/etc/cgconfig.conf文件内容都应该重启cgconfig服务，并重新登陆zorro账户。过程不再复述。*

将核心绑定到0-11之后的测试结果如下：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m9.457s
user    1m52.773s
sys 0m0.155s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m9.460s
user    1m52.589s
sys 0m0.153s

14:52:02     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
14:52:03     all   49.92    0.00    0.08    0.00    0.08    0.00    0.00    0.00
0.00   49.92
14:52:03       0  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

```
0.00    0.00
14:52:03       1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       2  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       3  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       4  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       5  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       6   99.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       7  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       8  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03       9  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03      10  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03      11  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
14:52:03      12    0.00    0.00    0.00    0.00    2.00    0.00    0.00    0.00
0.00   98.00
14:52:03      13    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      14    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      15    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      16    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      17    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      18    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      19    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      20    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      21    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      22    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
14:52:03      23    0.00    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00   99.01
```

此时会发现一个现象，执行的总体时间变化不大，大概慢了0.5秒，但是user时间下降了将近一半。

我们再降核心绑定成0-5,12-17测试一下，就是*cpuset.cpus = "0-5,12-17"*，测试结果如下：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m17.821s
user    3m32.425s
sys 0m0.223s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null
real    0m17.839s
user    3m32.375s
```

```
sys 0m0.223s

15:03:03     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
15:03:04     all   49.94    0.00    0.04    0.00    0.04    0.00    0.00    0.00
0.00   49.98
15:03:04       0  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       2  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       3  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       4  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       5  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04       6    0.00    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00   99.01
15:03:04       7    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04       8    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04       9    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      10    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      11    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      12  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      13  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      14  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      15   99.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      16   99.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      17  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
15:03:04      18    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      19    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      20    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      21    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      22    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
15:03:04      23    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
```

这次测试的结果就比较符合我们的常识，看上去cpu核心少了一半，于是执行时间增加了几乎一倍。那么是什么原因导致我们绑定到0-11核心的时候看上去性能没有下降呢？

在此我们不去过多讨论超线程的技术细节，简单来说：0-5核心是属于物理cpu0的6个实际核心，6-11是属于物理cpu1的6个实际核心，当我们使用这12个核心的时候，运算覆盖了两个物理cpu的所有真实核心。而12-17核心是对应0-5核心超线程出来的6个核心，18-23则是对应6-11核心超线程出来的6个。我们的测试应用并不能充分利用超线程之后的运算资源，所以，

从我们的测试用例角度看来，只要选择了合适核心，12核跟24核的效果几本差别不大。了解了超线程的这个问题，我们后续的测试过程就要注意对比的环境。从本轮测试看来，我们应该用绑定0-5，12-17的测试结果来参考绑定一半cpu核心的效果，而不是绑定到“0-11”上的结果。从测试结果看，减少一半核心之后，确实让运算时间增加了一倍。

> *出个两个思考题吧：*
>
> 1. *我们发现第二轮绑定0-11核心测试的user时间和绑定0-23的测试时间减少一倍，而real时间几乎没变，这是为什么？*
> 2. *我们发现第三轮绑定0-5，12-17核心测试的user时间和绑定0-23的测试时间几乎一样，而real时间增加了一倍，这是为什么？*

至此，如何使用cgroup的cpuset对cpu核心进行资源分配的方法大家应该学会了，这里需要强调一点：

配置中*cpuset.mems = “0-1”*这段配置非常重要，它相当于打开cpuset功能的开关，本身的意义是用来配置cpu使用的内存节点的，不配置这个字段的结果将是cpuset.cpus设置无效。字段具体含义，请大家自行补脑。

###针对CPU时间进行资源隔离

再回顾一下系统对cpu资源的使用方式 – –分时使用。分时使用要有一个基本本的时间调度单元，这个单元的意思是说，在这样一段时间范围内，我们将多少比例分配给某个进程组。我们刚才举的例子是说1秒钟，但是实际情况是1秒钟这个时间周期对计算机来说有点长。Linux内核将这个时间周期定义放在cgroup相关目录下的一个文件里，这个文件在我们服务器上是：

```
[root@zorrozou-pc ~]# cat /cgroup/cpu/zorro/cpu.cfs_period_us
100000
```

这个数字的单位是微秒，就是说，我们的cpu时间周期是100ms。还有一点需要注意的是，这个时间是针对单核来说的。

那么针对cgroup的限制放在哪里呢？

```
[root@zorrozou-pc ~]# cat /cgroup/cpu/zorro/cpu.cfs_quota_us
-1
```

就是这个cpu.cfs_quota_us文件。这里的cfs就是完全公平调度器，我们的资源隔离就是靠cfs来实现的。-1表示目前无限制。

限制方法很简单，就是设置cpu.cfs_quota_us这个文件的值，调度器会根据这个值的大小决定进程组在一个时间周期内（即100ms）使用cpu时间的比率。比如这个值我们设置成50000，那么就是时间周期的50%，于是这个进程组只能在一个cpu上占用50%的cpu时间。理解了这个概念，我们就可以思考一下，如果想让我们的进程在24核的服务器上不绑定核心的情况下占用所有核心的50%的cpu时间，该如何设置？计算公式为：

> *（50% * 100000 * cpu核心数）*

在此设置为1200000，我们来试一下。修改cgconfig.conf内容，然后重启cgconfig：

```
group zorro {
    cpu {
            cpu.cfs_quota_us = "1200000";
    }
}

[root@zorrozou-pc ~]# service cgconfig restart
```

测试结果如下：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m17.322s
user    3m27.116s
sys 0m0.266s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m17.347s
user    3m27.208s
sys 0m0.260s

16:15:12     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
16:15:13     all   49.92    0.00    0.08    0.00    0.04    0.00    0.00    0.00
0.00   49.96
16:15:13       0   51.49    0.00    0.00    0.00    0.99    0.00    0.00    0.00
0.00   47.52
16:15:13       1   51.49    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00   47.52
16:15:13       2   54.46    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   45.54
16:15:13       3   51.52    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   48.48
16:15:13       4   48.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.49
16:15:13       5   48.04    0.00    0.98    0.00    0.00    0.00    0.00    0.00
0.00   50.98
16:15:13       6   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13       7   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13       8   49.49    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.51
16:15:13       9   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      10   48.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      11   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      12   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      13   49.49    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.51
16:15:13      14   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      15   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
```

```
16:15:13      16   50.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   49.49
16:15:13      17   49.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   51.00
16:15:13      18   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
16:15:13      19   50.50    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   49.50
16:15:13      20   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
16:15:13      21   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
16:15:13      22   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
16:15:13      23   50.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00   50.00
```

我们可以看到，基本跟绑定一半的cpu核心数的效果一样，从这个简单的对比来看，使用cpu核心数绑定的方法和使用cpu分配时间的方法，在隔离上效果几乎是相同的。但是考虑到超线程的影响，我们使用cpu时间比率的方式很可能根cpuset的方式有些差别，为了看到这个差别，我们将针对cpuset和cpuquota进行一个对比测试，测试结果如下表：

cpu比率（核心数）	**cpuset realtime**	**cpuquota realtime**
8.3%(2)	1m46.557s	1m36.786s
16.7%(4)	0m53.271s	0m51.067s
25%(6)	0m35.528s	0m34.539s
33.3%(8)	0m26.643s	0m25.923s
50%(12)	0m17.839s	0m17.347s
66.7%(16)	0m13.384s	0m13.015s
100%(24)	0m8.972s	0m8.932s

> *思考题时间又到了：请解释这个表格测试得到的数字的差异。*

我们现在已经学会了如何使用cpuset和cpuquota两种方式对cpu资源进行分配，但是这两种分配的缺点也是显而易见的－－就是分配完之后，进程都最多只能占用相关比例的cpu资源。即使服务器上还有空闲资源，这两种方式都无法将资源“**借来使用**”。

那么有没有一种方法，既可以保证在系统忙的情况下让cgroup进程组只占用相关比例的资源，而在系统闲的情况下，又可以借用别人的资源，以达到资源利用率最大话的程度呢？当然有！那就是－－

###权重CPU资源隔离

这里的权重其实是shares。我把它叫做权重是因为这个值可以理解为对资源占用的权重。这种资源隔离方式事实上也是对cpu时间的进行分配。区别是作用在cfs调度器的权重值上。从用户的角度看，无非就是给每个cgroup配置一个share值，cpu在进行时间分配的时候，按照share的大小比率来确定cpu时间的百分比。它对比cpuquota的优势是，当进程不在cfs可执行调度队列中的时候，这个权重是不起作用的。就是说，一旦其他cgroup的进程释放cpu的时候，正在占用cpu的进程可以全占所有计算资源。而当有多个cgroup进程都要占用cpu的时候，大家按比例分配。

我们照例通过实验来说明这个情况，配置方法也很简单，修改cgconfig.conf，添加字段，并重启服务：

```
group zorro {
    cpu {
            cpu.shares = 1000;
    }
}

[root@zorrozou-pc ~]# service cgconfig restart
```

配置完之后，我们就给zorro组配置了一个shares值为1000，但是实际上如果系统中只有这一个组的话，cpu看起来对他是没有限制的。现在的执行效果是这样：

```
[root@zorrozou-pc ~]# mpstat -P ALL 1

17:17:29     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
17:17:30     all   99.88    0.00    0.12    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       0  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       2  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       3   99.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       4   99.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       5  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       6  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       7  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       8  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30       9  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      10  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      11  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      12  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      13  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      14  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      15  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      16  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      17   99.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      18  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      19  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

```
0.00    0.00
17:17:30      20  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      21  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      22  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:17:30      23  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00

[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m8.937s
user    3m32.190s
sys 0m0.225s
```

如显示，cpu我们是独占的。那么什么时候有隔离效果呢？是系统中有别的cgroup也要占用cpu的时候，就能看出效果了。比如此时我们再添加一个jerry，shares值也配置为1000，并且让jerry组一直有占用cpu的进程在运行。

```
group jerry {
    cpu {
            cpu.shares = "1000";
    }
}


top - 17:24:26 up 1 day, 5 min,  2 users,  load average: 41.34, 16.17, 8.17
Tasks: 350 total,   2 running, 348 sleeping,   0 stopped,   0 zombie
Cpu0  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu8  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu9  : 99.7%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.3%hi,  0.0%si,  0.0%st
Cpu10 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu11 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu12 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu13 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu14 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu15 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu16 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu17 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu18 : 99.3%us,  0.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu19 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu20 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu21 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu22 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu23 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  131904480k total,  4938020k used, 126966460k free,   136140k buffers
Swap:  2088956k total,        0k used,  2088956k free,  3700480k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
13945 jerry     20   0  390m  872  392 S 2397.2  0.0  48:42.54 jerry
```

我们以jerry用户身份执行了一个进程一直100%占用cpu，从上面的显示可以看到，这个进程占用了2400%的cpu，是因为每个cpu核心算100%，24个核心就是2400%。此时我们再以zorro身份执行筛质数的程序，并察看这个程序占用cpu的百分比：

```
top - 19:44:11 up 1 day,  2:25,  3 users,  load average: 60.91, 50.92, 48.85
Tasks: 336 total,   3 running, 333 sleeping,   0 stopped,   0 zombie
Cpu0  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu8  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu9  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu10 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu11 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu12 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu13 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu14 : 99.7%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.3%hi,  0.0%si,  0.0%st
Cpu15 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu16 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu17 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu18 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu19 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu20 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu21 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu22 : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu23 :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  131904480k total,  1471772k used, 130432708k free,   144216k buffers
Swap:  2088956k total,        0k used,  2088956k free,   322404k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
13945 jerry     20   0  390m  872  392 S 1200.3  0.0   3383:04 jerry
 9311 zorro     20   0  390m  872  392 R 1197.0  0.0   0:51.56 prime_thread_zo
```

通过top我们可以看到，以zorro用户身份执行的进程和jerry进程平分了cpu，每人50%。zorro筛质数执行的时间为：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m15.152s
user    2m58.637s
sys 0m0.220s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m15.465s
user    3m0.706s
sys 0m0.221s
```

根据这个时间看起来，基本与通过cpuquota方式分配50%的cpu时间以及通过cpuset方式分配12个核心的情况相当，而且效率还稍微高一些。当然我要说明的是，这里几乎两秒左右的效率的提高并不具备很大的参考性，它与jerry进程执行的运算是有很大相关性的。此时jerry进程执行的是一个多线程的while死循环，占满所有cpu跑。当我们把jerry进程执行的内容同样变成筛质数的时候，zorro用户的进程执行效率的参考值就比较标准了：

```
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m17.521s
user    3m32.684s
sys 0m0.254s
[zorro@zorrozou-pc ~/test]$ time ./prime_thread_zorro &> /dev/null

real    0m17.597s
user    3m32.682s
sys 0m0.253s
```

如程序执行显示，执行效率基本与cpuset和cpuquota相当。

> *这又引发了另一个问题请大家思考：为什么jerry用户执行的运算的逻辑不同会影响zorro用户的运算效率？*

我们可以将刚才cpuset和cpuquota的对比列表加入cpushare一列来一起对比了，为了方便参考，我们都以cpuset为基准进行比较：

shares zorro/shares jerry（核心数）	**cpuset realtime**	**cpushare realtime**	**cpuquota realtime**
2000/22000(2)	1m46.557s	**1m41.691s**	1m36.786s
4000/20000(4)	0m53.271s	**0m51.801s**	0m51.067s
6000/18000(6)	0m35.528s	**0m35.152s**	0m34.539s
8000/16000(8)	0m26.643s	**0m26.372s**	0m25.923s
12000/12000(12)	0m17.839s	**0m17.694s**	0m17.347s
16000/8000(16)	0m13.384s	**0m13.388s**	0m13.015s
24000/0(24)	0m8.972s	**0m8.943s**	0m8.932s

请注意一个问题，由于cpushares无法像cpuquota或者cpuset那样只执行zorro用户的进程，所以在进行cpushares测试的时候，必须让jerry用户同时执行相同的筛质数程序，才能使两个用户分别分到相应比例的cpu时间。这样可能造成本轮测试结果的不准确。通过对比看到，当比率分别都配置了相当于两个核心的计算能力的情况下，本轮测试是cpuquota方式消耗了1m36.786s稍快一些。为了保证相对公平的环境作为参照，我们将重新对这轮测试进行数据采集，这次在cpuset和cpuquota的压测时，都用jerry用户执行一个干扰程序作为参照，重新分析数据。当然，cpushares的测试数据就不必重新测试了：

shares zorro/shares jerry（核心数）	**cpuset realtime**	**cpushare realtime**	**cpuquota realtime**
2000/22000(2)	1m46.758s	1m41.691s	1m42.341s
4000/20000(4)	0m53.340s	0m51.801s	0m51.512s
6000/18000(6)	0m35.525s	0m35.152s	0m34.392s
8000/16000(8)	0m26.738s	0m26.372s	0m25.772s

12000/12000(12)	0m17.793s	0m17.694s	0m17.256s
16000/8000(16)	0m13.366s	0m13.388s	0m13.155s
24000/0(24)	0m8.930s	0m8.943s	0m8.939s

至此，cgroup中针对cpu的三种资源隔离都介绍完了，分析我们的测试数据可以得出一些结论：

1. 三种cpu资源隔离的效果基本相同，在资源分配比率相同的情况下，它们都提供了差不多相同的计算能力。
2. cpuset隔离方式是以分配核心的方式进行资源隔离，可以提供的资源分配最小粒度是核心，不能提供更细粒度的资源隔离，但是隔离之后运算的相互影响最低。需要注意的是在服务器开启了超线程的情况下，要小心选择分配的核心，否则不同cgroup间的性能差距会比较大。
3. cpuquota给我们提供了一种比cpuset可以更细粒度的分配资源的方式，并且保证了cgroup使用cpu比率的上限，相当于对cpu资源的硬限制。
4. cpushares给我们提供了一种可以按权重比率弹性分配cpu时间资源的手段：当cpu空闲的时候，某一个要占用cpu的cgroup可以完全占用剩余cpu时间，充分利用资源。而当其他cgroup需要占用的时候，每个cgroup都能保证其最低占用时间比率，达到资源隔离的效果。

大家可以根据这三种不同隔离手段特点，针对自己的环境来选择不同的方式进行cpu资源的隔离。当然，这些手段也可以混合使用，以达到更好的QOS效果。

但是可是but，这就完了么？
显然并没有。。。。。。

以上测试只针对了一种计算场景，这种场景在如此的简单的情况下，影响测试结果的条件已经很复杂了。如果是其他情况呢？我们线上真正跑业务的环境会这么单纯么？显然不会。我们不可能针对所有场景得出结论，想要找到适用于自己场景的隔离方式，还是需要在自己的环境中进行充分测试。在此只能介绍方法，以及针对一个场景的参考数据，仅此而已。单就这一个测试来说，它仍然不够全面，无法体现出内核cpu资源隔离的真正面目。众所周知，cpu使用主要分两个部分，user和sys。上面这个测试，由于测试用例的选择，只关注了user的使用。那么如果我们的sys占用较多会变成什么样呢？

##CPU资源隔离在sys较高的情况下是什么表现？

###内核资源不冲突的情况

首先我们简单说一下什么叫sys较高。先看mpstat命令的输出：

```
[root@zorrozou-pc ~]# mpstat 1
Linux 3.10.90-1-linux (zorrozou-pc)         12/24/15    _x86_64_    (24 CPU)

16:08:52     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
16:08:53     all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
16:08:54     all    0.00    0.00    0.04    0.00    0.04    0.00    0.00    0.00
0.00   99.92
16:08:55     all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00  100.00
16:08:56     all    0.04    0.00    0.04    0.00    0.00    0.00    0.00    0.00
0.00   99.92
16:08:57     all    0.04    0.00    0.04    0.00    0.00    0.00    0.00    0.00
0.00   99.92
16:08:58     all    0.00    0.00    0.04    0.00    0.00    0.00    0.00    0.00
```

```
0.00   99.96

Average:     all    0.01    0.00    0.03    0.00    0.01    0.00    0.00    0.00
0.00   99.95
```

这里面我们看到cpu的使用比率分了很多栏目，我们一般评估进程占用CPU的时候，最重要的是%user和%sys。%sys一般是指，进程陷入内核执行时所占用的时间，这些时间是内核在工作。常见的情况时，进程执行过程中之行了某个系统调用，而陷入内核态执行所产生的cpu占用。

所以在这一部分，我们需要重新提供一个测试用例，让sys部分的cpu占用变高。基于筛质数进行改造即可，我们这次让每个筛质数的线程，在做运算之前都用非阻塞方式open()打开一个文件，每次拿到一个数运算的时候，循环中都用系统调用read()读一下文件。以此来增加sys占用时间的比率。先来改程序：

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

#define NUM 48
#define START 1010001
#define END 1020000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int count = 0;

void *prime(void *p)
{
    int n, i, flag;
    int num, fd, ret;
    char name[BUFSIZ];
    char buf[BUFSIZ];

    bzero(name, BUFSIZ);

    num = (int *)p;
    sprintf(name, "/tmp/tmpfilezorro%d", num);

    fd = open(name, O_RDWR|O_CREAT|O_TRUNC|O_NONBLOCK , 0644);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }

    while (1) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock()");
            pthread_exit(NULL);
        }
        while (count == 0) {
            if (pthread_cond_wait(&cond, &mutex) != 0) {
                perror("pthread_cond_wait()");
                pthread_exit(NULL);
            }
        }
```

```
        if (count == -1) {
            if (pthread_mutex_unlock(&mutex) != 0) {
                perror("pthread_mutex_unlock()");
                pthread_exit(NULL);
            }
            break;
        }
        n = count;
        count = 0;
        if (pthread_cond_broadcast(&cond) != 0) {
            perror("pthread_cond_broadcast()");
            pthread_exit(NULL);
        }
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("pthread_mutex_unlock()");
            pthread_exit(NULL);
        }
        flag = 1;
        for (i=2;i<n/2;i++) {
            ret = read(fd, buf, BUFSIZ);
            if (ret < 0) {
                perror("read()");
            }
            if (n%i == 0) {
                flag = 0;
                break;
            }
        }
        if (flag == 1) {
            printf("%d is a prime form %d!\n", n, pthread_self());
        }
    }

    close(fd);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid[NUM];
    int ret, i, num;

    for (i=0;i<NUM;i++) {
        ret = pthread_create(&tid[i], NULL, prime, (void *)i);
        if (ret != 0) {
            perror("pthread_create()");
            exit(1);
        }
    }

    for (i=START;i<END;i+=2) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock()");
            pthread_exit(NULL);
        }
        while (count != 0) {
            if (pthread_cond_wait(&cond, &mutex) != 0) {
                perror("pthread_cond_wait()");
                pthread_exit(NULL);
            }
        }
        count = i;
        if (pthread_cond_broadcast(&cond) != 0) {
            perror("pthread_cond_broadcast()");
            pthread_exit(NULL);
```

```
        }
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("pthread_mutex_unlock()");
            pthread_exit(NULL);
        }
    }
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock()");
        pthread_exit(NULL);
    }
    while (count != 0) {
        if (pthread_cond_wait(&cond, &mutex) != 0) {
            perror("pthread_cond_wait()");
            pthread_exit(NULL);
        }
    }
    count = -1;
    if (pthread_cond_broadcast(&cond) != 0) {
        perror("pthread_cond_broadcast()");
        pthread_exit(NULL);
    }
    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("pthread_mutex_unlock()");
        pthread_exit(NULL);
    }

    for (i=0;i<NUM;i++) {
        ret = pthread_join(tid[i], NULL);
        if (ret != 0) {
            perror("pthread_join()");
            exit(1);
        }
    }

    exit(0);
}
```

我们将筛质数的范围缩小了两个数量级，并且每个线程都打开一个文件，每次计算的循环中都read一遍。此时这个进程执行的时候的cpu使用状态是这样的：

```
 17:20:46     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice   %idle
17:20:47     all   53.04    0.00   46.96    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       0   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       1   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       2   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       3   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       4   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       5   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       6   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       7   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
```

```
17:20:47       8   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47       9   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      10   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      11   53.47    0.00   46.53    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      12   52.00    0.00   48.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      13   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      14   53.47    0.00   46.53    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      15   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      16   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      17   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      18   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      19   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      20   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      21   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      22   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00
17:20:47      23   53.00    0.00   47.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00

[zorro@zorrozou-pc ~/test]$ time ./prime_sys &> /dev/null

real    0m12.227s
user    2m34.869s
sys 2m17.239s
```

测试用例已经基本符合我们的测试条件，可以达到近50%的sys占用，下面开始进行对比测试。测试方法根上一轮一样，仍然用jerry账户运行一个相同的程序在另一个cgroup不断的循环，然后分别看在不同资源分配比率下的zorro用户筛质数程序运行的时间。以下是测试结果：

shares zorro/shares jerry（核心数）	**cpuset realtime**	**cpushare realtime**	**cpuquota realtime**
2000/22000(2)	2m27.666s	2m27.599s	2m27.918s
4000/20000(4)	1m12.621s	1m14.345s	1m13.581s
6000/18000(6)	0m48.612s	0m49.474s	0m48.730s
8000/16000(8)	0m36.412s	0m37.269s	0m36.784s
12000/12000(12)	0m24.611s	0m24.624s	0m24.628s
16000/8000(16)	0m18.401s	0m18.688s	0m18.480s
24000/0(24)	0m12.188s	0m12.487s	0m12.147s

shares zorro/shares jerry（核心数）	cpuset systime	cpushare systime	cpuquota systime
2000/22000(2)	2m20.115s	2m21.024s	2m21.854s
4000/20000(4)	2m16.450s	2m21.103s	2m20.352s
6000/18000(6)	2m18.273s	2m20.455s	2m20.039s
8000/16000(8)	2m18.054s	2m20.611s	2m19.891s
12000/12000(12)	2m20.358s	2m18.331s	2m20.363s
16000/8000(16)	2m17.724s	2m18.958s	2m18.637s
24000/0(24)	2m16.723s	2m17.707s	2m16.176s

这次我们多了一个表格专门记录systime时间占用。根据数据结果我们会发现，在这次测试循环中，三种隔离方式都呈现出随着资源的增加进程是执行的总时间线性下降，并且隔离效果区别不大。由于调用read的次数一样，systime的使用基本都稳定在一个固定的时间范围内。这说明，在sys占用较高的情况下，各种cpu资源隔离手段都表现出比较理想的效果。

###内核资源冲突的情况

但是现实的生产环境往往并不是这么理想的，有没有可能在某种情况下，各种CPU资源隔离的手段并不会表现出这么理想的效果呢？有没有可能不同的隔离方式会导致进程的执行会有影响呢？其实这是很可能发生的。我们上一轮测试中，每个cgroup中的线程打开的文件都不是同一个文件，内核在处理这种场景的时候，并不需要使用内核中的一些互斥资源(比如自旋锁或者屏障)进行竞争条件的处理。如果环境变成大家read的是同一个文件，那么情况就可能有很大不同了。下面我们来测试一下每个zorro组中的所有线程都open同一个文件并且read时的执行效果，我们照例把测试用例代码贴出来：

```
 #include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

#define NUM 48
#define START 1010001
#define END 1020000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int count = 0;
#define PATH "/etc/passwd"

void *prime(void *p)
{
    int n, i, flag;
    int num, fd, ret;
    char name[BUFSIZ];
    char buf[BUFSIZ];

    fd = open(PATH, O_RDONLY|O_NONBLOCK);
    if (fd < 0) {
        perror("open()");
```

```
        exit(1);
    }

    while (1) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock()");
            pthread_exit(NULL);
        }
        while (count == 0) {
            if (pthread_cond_wait(&cond, &mutex) != 0) {
                perror("pthread_cond_wait()");
                pthread_exit(NULL);
            }
        }
        if (count == -1) {
            if (pthread_mutex_unlock(&mutex) != 0) {
                perror("pthread_mutex_unlock()");
                pthread_exit(NULL);
            }
            break;
        }
        n = count;
        count = 0;
        if (pthread_cond_broadcast(&cond) != 0) {
            perror("pthread_cond_broadcast()");
            pthread_exit(NULL);
        }
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("pthread_mutex_unlock()");
            pthread_exit(NULL);
        }
        flag = 1;
        for (i=2;i<n/2;i++) {
            ret = read(fd, buf, BUFSIZ);
            if (ret < 0) {
                perror("read()");
            }
            if (n%i == 0) {
                flag = 0;
                break;
            }
        }
        if (flag == 1) {
            printf("%d is a prime form %d!\n", n, pthread_self());
        }
    }

    close(fd);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid[NUM];
    int ret, i, num;

    for (i=0;i<NUM;i++) {
        ret = pthread_create(&tid[i], NULL, prime, (void *)i);
        if (ret != 0) {
            perror("pthread_create()");
            exit(1);
        }
    }

    for (i=START;i<END;i+=2) {
```

```
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock()");
            pthread_exit(NULL);
        }
        while (count != 0) {
            if (pthread_cond_wait(&cond, &mutex) != 0) {
                perror("pthread_cond_wait()");
                pthread_exit(NULL);
            }
        }
        count = i;
        if (pthread_cond_broadcast(&cond) != 0) {
            perror("pthread_cond_broadcast()");
            pthread_exit(NULL);
        }
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("pthread_mutex_unlock()");
            pthread_exit(NULL);
        }
    }
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock()");
        pthread_exit(NULL);
    }
    while (count != 0) {
        if (pthread_cond_wait(&cond, &mutex) != 0) {
            perror("pthread_cond_wait()");
            pthread_exit(NULL);
        }
    }
    count = -1;
    if (pthread_cond_broadcast(&cond) != 0) {
        perror("pthread_cond_broadcast()");
        pthread_exit(NULL);
    }
    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("pthread_mutex_unlock()");
        pthread_exit(NULL);
    }

    for (i=0;i<NUM;i++) {
        ret = pthread_join(tid[i], NULL);
        if (ret != 0) {
            perror("pthread_join()");
            exit(1);
        }
    }

    exit(0);
}
```

此时jerry组中的所有线程仍然是每个线程一个文件，与上一轮测试一样。测试结果如下：

shares zorro/shares jerry（核心数）	**cpuset realtime**	**cpushare realtime**	**cpuquota realtime**
2000/22000(2)	2m27.402s	2m41.015s	4m37.149s
4000/20000(4)	1m18.178s	1m25.214s	2m42.455s
6000/18000(6)	0m52.592s	1m2.691s	1m48.492s

8000/16000(8)	0m43.598s	0m57.000s	1m21.044s
12000/12000(12)	0m52.182s	0m59.613s	0m58.004s
16000/8000(16)	0m50.712s	0m54.371s	0m56.911s
24000/0(24)	0m50.599s	0m50.550s	0m50.496s

shares zorro/shares jerry（核心数）	**cpuset systime**	**cpushare systime**	**cpuquota systime**
2000/22000(2)	2m19.829s	2m47.706s	6m39.800s
4000/20000(4)	2m41.928s	3m6.575s	8m14.087s
6000/18000(6)	2m45.671s	3m38.722s	8m13.668s
8000/16000(8)	3m14.434s	4m54.451s	8m12.904s
12000/12000(12)	7m39.542s	9m7.751s	8m57.332s
16000/8000(16)	10m47.425s	11m41.443s	12m21.056s
24000/0(24)	17m17.661s	17m7.311s	17m14.788s

观察这轮测试的结果我们会发现，当线程同时read同一个文件时，时间的消耗并不在呈现线性下降的趋势了，而且，随着分配的资源越来越多，sys占用时间也越来越高，这种现象如何解释呢？本质上来讲，使用cgroup进行资源隔离时，内核资源仍然是共享的。如果业务使用内核资源如果没有产生冲突，那么隔离效果应该会比较理想，但是业务一旦使用了会导致内核资源冲突的逻辑时，那么业务的执行效率就会下降，此时可能所有进程在内核中处理的时候都可能会在竞争的资源上忙等（如果使用了spinlock）。自然的，如果多个cgroup的进程之间也正好使用了可能会导致内核触发竞争条件的资源时，自然也会发生所谓的cgroup之间的相互影响。可能的现象就是，当某一个业务A的cgroup正在运行着，突然B业务的cgroup有请求要处理，会导致A业务的响应速度和处理能力下降。而这种相互干扰，正是资源隔离手段想要尽量避免的。我们认为，如果出现了上述效果，那么资源隔离手段就是打了折扣的。

根据我们的实验结果可以推论，在内核资源有竞争条件的情况下，cpuset的资源隔离方式表现出了相对其他方式的优势，cpushare方式的性能折损尚可接受，而cpuquota表现出了最差的性能，或者说在cpuquota的隔离条件下，cgroup之间进程相互影响的可能性最大。

那么在内核资源存在竞争的时候，cgroup的cpu资源隔离会有相互干扰。结论就是这样了么？这个推断靠谱么？我们再来做一轮实验，这次只对比cpuset和cpuquota。这次我们不用jerry来运行干扰程序测试隔离性，我们让zorro只在单纯的隔离状态下，再有内核资源竞争的条件下进行运算效率测试，就是说这个环境没有多个cgroup可能造成的相互影响。先来看数据：

cpu比率（核心数）	**cpuset realtime**	**cpuquota realtime**
8.3%(2)	2m26.815s	9m4.490s
16.7%(4)	1m17.894s	4m49.167s
25%(6)	0m52.356s	3m13.144s
33.3%(8)	0m42.946s	2m23.010s
50%(12)	0m52.014s	1m33.571s

66.7%(16)	0m50.903s	1m10.553s
100%(24)	0m50.331s	0m50.304s

cpu比率（核心数）	**cpuset systime**	**cpuquota systime**
8.3%(2)	2m18.713s	15m27.738s
16.7%(4)	2m41.172s	16m30.741s
25%(6)	2m44.618s	16m30.964s
33.3%(8)	3m12.587s	16m18.366s
50%(12)	7m36.929s	15m55.407s
66.7%(16)	10m49.327s	16m1.463s
100%(24)	17m9.482s	17m9.533s

不知道看完这组数据之后，大家会不会困惑？cpuset的测试结果根上一轮基本一样，这可以理解。但是为什么cpuquota这轮测试反倒比刚才有jerry用户进程占用cpu进行干扰的时候的性能更差了？

如果了解了内核在这种资源竞争条件的原理的话，这个现象并不难解释。可以这样想，如果某一个资源存在竞争的话，那么是不是同时竞争的人越多，那么对于每个人来说，单次得到资源的可能性更低？比如说，老师给学生发苹果，每次只发一个，但是同时有10个人一起抢，每个人每次抢到苹果的几率是10%，如果20个人一起抢，那么每次每人强到苹果的几率就只有5％了。在内核竞争条件下，也是一样的道理，资源只有一个，当抢的进程少的时候，每个进程抢到资源的概率大，于是浪费在忙等上的时间就少。本轮测试的cpuset就可以说明这个现象，可以观察到，cpuset systime随着分配的核心数的增多而上升，就是同时跑的进程越多，sys消耗在忙等资源上的时间就越大。而cpuquota systime消耗从头到尾都基本变化不大，意味着再以quota方式分配cpu的时候，所有核心都是用得上的，所以一直都有24个进程在抢资源，大家消耗在忙等上的时间是一样的。
为什么有jerry进程同时占用cpu的情况下，cpuquota反倒效率要快些呢？这个其实也好理解。在jerry进程执行的时候，这个cgroup的相关线程打开的是不同的文件，所以从内核竞争上没有冲突。另外，jerry消耗了部分cpu，导致内核会在zorro的进程和jerry的进程之间发生调度，这意味着，同一时刻核心数只有24个，可能有18个在给jerry的线程使用，另外6个在给zorro的进程使用，这导致zorro同时争抢资源的进程个数不能始终保持24个，所以内核资源冲突反倒减小了。 这导致，使用cpuquota的情况下，有其他cgroup执行的时候，还可能会使某些业务的执行效率提升，而不是下降。 这种相互影响实在太让人意外了！但这确实是事实！

那么什么情况下会导致cgroup之间的相互影响使性能下降呢？也好理解，当多个cgroup的应用之间使用了相同的内核资源的时候。请大家思考一个问题：现实情况是同一种业务使用冲突资源的可能性更大还是不同业务使用冲突资源的可能性更大呢？从概率上说应该是同一种业务。从这个角度出发来看，如果我们有两台多核服务器，有两个跟我们测试逻辑类似的业务A、B，让你选择一种部署方案，你是选择让A、B两个业务分别独占一个服务器？还是让A、B业务使用资源隔离分别在两个服务器上占用50%的资源？通过这轮分析我想答案很明确了：

1. 从容灾的角度说，让某一个业务使用多台服务器肯定会增加容灾能力。
2. 从资源利用率的角度说，如果让一个业务放在一个服务器上，那么他在某些资源冲突的情况下并不能发挥会最大效率。然而如果使用group分布在两个不同的服务器上，无论你用cpuset，还是cpushare，又或是cpuquota，它的cpu性能表现都应该强于在一个独立的服务器上部署。况且cgroup的cpu隔离是在cfs中实现的，这种隔离几乎是不会浪费额外的计算能力的，就是说，做隔离相比不做隔离，系统本身的性能损耗都可以忽略不计。

那么，究竟还有什么会妨碍我们使用cgoup的cpu资源隔离呢？

一月 22, 2016 / Linux / 留下评论

Hello world!

按照惯例，第一篇博文就应该是hello world！这代表开启了一个新的世界，这代表开始了一段新的旅程。

我应该会坚持写这个博客的，就跟我那些其他的爱好一样。

太阳升起了，跑道的终点也许也是起点……

一月 21, 2016 / Uncategorized / 有1条评论