目錄

穷佐罗的Linux书

这里是穷佐罗的Linux书。在这里，你能看到来自穷佐罗的个人技术文章。

希望你喜欢。

Linux的进程优先级

Hi，我是Zorro。扫上面二维码或搜索：**Linux**系统技术 关注我的公众号，可以订阅我的最新文章哦。

这是我的博客地址，欢迎来一起探讨。 我的微博地址，有兴趣可以来关注我吻。

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的进程优先级

为什么要有进程优先级？这似乎不用过多的解释，毕竟自从多任务操作系统诞生以来，进程执行占用cpu的能力就是一个必须要可以人为控制的事情。因为有的进程相对重要，而有的进程则没那么重要。进程优先级起作用的方式从发明以来基本没有什么变化，无论是只有一个cpu的时代，还是多核cpu时代，都是通过控制进程占用cpu时间的长短来实现的。就是说在同一个调度周期中，优先级高的进程占用的时间长些，而优先级低的进程占用的短些。从这个

角度看，进程优先级其实也跟cgroup的cpu限制一样，都是一种针对cpu占用的QOS机制。我曾经一直很困惑一点，为什么已经有了优先级，还要再设计一个针对cpu的cgroup？得到的答案大概是因为，优先级这个值不能很直观的反馈出资源分配的比例吧？不过这不重要，实际上从内核目前的进程调度器cfs的角度说，同时实现cpushare方式的cgroup和优先级这两个机制完全是相同的概念，并不会因为增加一个机制而提高什么实现成本。既然如此，而cgroup又显得那么酷，那么何乐而不为呢？

再系统上我们最熟悉的优先级设置方式是nice喝renice命令。那么我们首先解释一个概念，什么是：

NICE值

nice值应该是熟悉Linux/UNIX的人很了解的概念了，我们都知它是反应一个进程“优先级”状态的值，其取值范围是-20至19，一共40个级别。这个值越小，表示进程”优先级”越高，而值越大“优先级”越低。我们可以通过nice命令来对一个将要执行的命令进行nice值设置，方法是：

```
[root@zorrozou-pc0 zorro]# nice -n 10 bash
```

这样我就又打开了一个bash，并且其nice值设置为10，而默认情况下，进程的优先级应该是从父进程继承来的，这个值一般是0。我们可以通过nice命令直接查看到当前shell的nice值

```
[root@zorrozou-pc0 zorro]# nice
10
```

对比一下正常情况：

```
[root@zorrozou-pc0 zorro]# exit
```

推出当前nice值为10的bash，打开一个正常的bash：

```
[root@zorrozou-pc0 zorro]# bash
[root@zorrozou-pc0 zorro]# nice
0
```

另外，使用renice命令可以对一个正在运行的进程进行nice值的调整，我们也可以使用比如top、ps等命令查看进程的nice值，具体方法我就不多说了，大家可以参阅相关manpage。

需要大家注意的是，我在这里都在使用nice值这一称谓，而非优先级（priority）这个说法。当然，nice和renice的man手册中，也说的是priority这个概念，但是要强调一下，请大家真的不要混淆了系统中的这两个概念，一个是nice值，一个是priority值，他们有着千丝万缕的关系，但对于当前的Linux系统来说，它们并不是同一个概念。

我们看这个命令：

```
[root@zorrozou-pc0 zorro]# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  6924  5776  0  80   0 - 17952 poll_s pts/5    00:00:00 sudo
4 S     0  6925  6924  0  80   0 -  4435 wait   pts/5    00:00:00 bash
0 R     0 12971  6925  0  80   0 -  8514 -      pts/5    00:00:00 ps
```

大家是否真的明白其中PRI列和NI列的具体含义有什么区别？同样的，如果是top命令：

```
Tasks: 1587 total,   7 running, 1570 sleeping,   0 stopped,  10 zombie
Cpu(s): 13.0%us,  6.9%sy,  0.0%ni, 78.6%id,  0.0%wa,  0.0%hi,  1.5%si,  0.0%st
Mem:  132256952k total, 107483920k used, 24773032k free,  2264772k buffers
Swap:  2101192k total,      508k used,  2100684k free, 88594404k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND



 3001 root      20   0  232m  21m 4500 S 12.9  0.0   0:15.09 python



11541 root      20   0 17456 2400  888 R  7.4  0.0   0:00.06 top
```

大家是否搞清楚了这其中PR值和NI值的差别？如果没有，那么我们可以首先搞清楚什么是nice值。

nice值虽然不是priority，但是它确实可以影响进程的优先级。在英语中，如果我们形容一个人nice，那一般说明这个人的人缘比较好。什么样的人人缘好？往往是谦让、有礼貌的人。比如，你跟一个nice的人一起去吃午饭，点了两个一样的饭，先上了一份后，nice的那位一般都会说：“你先吃你先吃！”，这就是人缘好，这人nice！但是如果另一份上的很晚，那么这位nice的人就要饿着了。这说明什么？越nice的人抢占资源的能力就越差，而越不nice的人抢占能力就越强。这就是nice值大小的含义，nice值越低，说明进程越不nice，抢占cpu的能力就越强，优先级就越高。在原来使用O1调度的Linux上，我们还会把nice值叫做静态优先级，这也基本符合nice值的特点，就是当nice值设定好了之后，除非我们用renice去改它，否则它是不变的。而priority的值在之前内核的O1调度器上表现是会变化的，所以也叫做动态优先级。

优先级和实时进程

简单了解nice值的概念之后，我们再来看看什么是priority值，就是ps命令中看到的PRI值或者top命令中看到的PR值。本文为了区分这些概念，以后统一用nice值表示NI值，或者叫做静态优先级，也就是用nice和renice命令来调整的优先级；而实用priority值表示PRI和PR值，或者叫动态优先级。我们也统一将“优先级”这个词的概念规定为表示priority值的意思。

在内核中，进程优先级的取值范围是通过一个宏定义的，这个宏的名称是MAX_PRIO，它的值为140。而这个值又是由另外两个值相加组成的，一个是代表nice值取值范围的NICE_WIDTH宏，另一个是代表实时进程（realtime）优先级范围的MAX_RT_PRIO宏。说白了就是，Linux实际上实现了140个优先级范围，取值范围是从0-139，这个值越小，优先级越高。nice值的-20到19，映射到实际的优先级范围是100-139。新产生进程的默认优先级被定义为：

```
#define DEFAULT_PRIO            (MAX_RT_PRIO + NICE_WIDTH / 2)
```

实际上对应的就是nice值的0。正常情况下，任何一个进程的优先级都是这个值，即使我们通过nice和renice命令调整了进程的优先级，它的取值范围也不会超出100-139的范围，除非这个进程是一个实时进程，那么它的优先级取值才会变成0-99这个范围中的一个。这里隐含了一个信息，就是说当前的Linux是一种已经支持实时进程的操作系统。

什么是实时操作系统，我们就不再这里详细解释其含义以及在工业领域的应用了，有兴趣的可以参考一下实时操作系统的维基百科。简单来说，实时操作系统需要保证相关的实时进程在较短的时间内响应，不会有较长的延时，并且要求最小的中断延时和进程切换延时。对于这样的需求，一般的进程调度算法，无论是O1还是CFS都是无法满足的，所以内核在设计的时候，将实时进程单独映射了100个优先级，这些优先级都要高与正常进程的优先级（nice值），而实时进程的调度算法也不同，它们采用更简单的调度算法来减少调度开销。总的来说，Linux系统中运行的进程可以分成两类：

1. 实时进程
2. 非实时进程

它们的主要区别就是通过优先级来区分的。所有优先级值在0-99范围内的，都是实时进程，所以这个优先级范围也可以叫做实时进程优先级，而100-139范围内的是非实时进程。在系统中可以使用chrt命令来查看、设置一个进程的实时优先级状态。我们可以先来看一下chrt命令的使用：

```
[root@zorrozou-pc0 zorro]# chrt
Show or change the real-time scheduling attributes of a process.

Set policy:
 chrt [options] <priority> <command> [<arg>...]
 chrt [options] -p <priority> <pid>

Get policy:
 chrt [options] -p <pid>

Policy options:
 -b, --batch          set policy to SCHED_OTHER
 -f, --fifo           set policy to SCHED_FIFO
 -i, --idle           set policy to SCHED_IDLE
 -o, --other          set policy to SCHED_OTHER
 -r, --rr             set policy to SCHED_RR (default)

Scheduling flag:
 -R, --reset-on-fork  set SCHED_RESET_ON_FORK for FIFO or RR

Other options:
 -a, --all-tasks      operate on all the tasks (threads) for a given pid
 -m, --max            show min and max valid priorities
 -p, --pid            operate on existing given pid
 -v, --verbose        display status information

 -h, --help     display this help and exit
 -V, --version  output version information and exit

For more details see chrt(1).
```

我们先来关注显示出的Policy options部分，会发现系统给个种进程提供了5种调度策略。但是这里并没有说明的是，这五种调度策略是分别给两种进程用的，对于实时进程可以用的调度策略是：SCHED_FIFO、SCHED_RR，而对于非实时进程则是：SCHED_OTHER、SCHED_OTHER、SCHED_IDLE。

系统的整体优先级策略是：如果系统中存在需要执行的实时进程，则优先执行实时进程。直到实时进程退出或者主动让出CPU时，才会调度执行非实时进程。实时进程可以指定的优先级范围为1-99，将一个要执行的程序以实时方式执行的方法为：

```
[root@zorrozou-pc0 zorro]# chrt 10 bash
[root@zorrozou-pc0 zorro]# chrt -p $$
pid 14840's current scheduling policy: SCHED_RR
pid 14840's current scheduling priority: 10
```

可以看到，新打开的bash已经是实时进程，默认调度策略为SCHED_RR，优先级为10。如果想修改调度策略，就加个参数：

```
[root@zorrozou-pc0 zorro]# chrt -f 10 bash
[root@zorrozou-pc0 zorro]# chrt -p $$
pid 14843's current scheduling policy: SCHED_FIFO
pid 14843's current scheduling priority: 10
```

刚才说过，SCHED_RR和SCHED_FIFO都是实时调度策略，只能给实时进程设置。对于所有实时进程来说，优先级高的（就是priority数字小的）进程一定会保证先于优先级低的进程执行。SCHED_RR和SCHED_FIFO的调度策略只有当两个实时进程的优先级一样的时候才会发生作用，其区别也是顾名思义：

SCHED_FIFO:以先进先出的队列方式进行调度，在优先级一样的情况下，谁先执行的就先调度谁，除非它退出或者主动释放CPU。

SCHED_RR:以时间片轮转的方式对相同优先级的多个进程进行处理。时间片长度为100ms。

这就是Linux对于实时进程的优先级和相关调度算法的描述。整体很简单，也很实用。而相对更麻烦的是非实时进程，它们才是Linux上进程的主要分类。对于非实时进程优先级的处理，我们首先还是要来介绍一下它们相关的调度算法：O1和CFS。

O1调度

O1调度算法是在Linux 2.6开始引入的，到Linux 2.6.23之后内核将调度算法替换成了CFS。虽然O1算法已经不是当前内核所默认使用的调度算法了，但是由于大量线上的服务器可能使用的Linux版本还是老版本，所以我相信很多服务器还是在使用着O1调度器，那么费一点口舌简单交代一下这个调度器也是有意义的。这个调度器的名字之所以叫做O1，主要是因为其算法的时间复杂度是O1。

O1调度器仍然是根据经典的时间片分配的思路来进行整体设计的。简单来说，时间片的思路就是将CPU的执行时间分成一小段一小段的，假如是5ms一段。于是多个进程如果要“同时”执行，实际上就是每个进程轮流占用5ms的cpu时间，而从1s的时间尺度上看，这些进程就是在“同时”执行的。当然，对于多核系统来说，就是把每个核心都这样做就行了。而在这种情况下，如何支持优先级呢？实际上就是将时间片分配成大小不等的若干种，优先级高的进程使用大的时间片，优先级小的进程使用小的时间片。这样在一个周期结速后，优先级大的进程就会占用更多的时间而因此得到特殊待遇。O1算法还有一个比较特殊的地方是，即使是相同的nice值的进程，也会再根据其CPU的占用情况将其分成两种类型：CPU消耗型和IO消耗性。典型的CPU消耗型的进程的特点是，它总是要一直占用CPU进行运算，分给它的时间片总是会被耗尽之后，程序才可能发生调度。比如常见的各种算数运算程序。而IO消耗型的特点是，它经常时间片没有耗尽就自己主动先释放CPU了，比如vi，emacs这样的编辑器就是典型的IO消耗型进程。

为什么要这样区分呢？因为IO消耗型的进程经常是跟人交互的进程，比如shell、编辑器等。当系统中既有这种进程，又有CPU消耗型进程存在，并且其nice值一样时，假设给它们分的时间片长度是一样的，都是500ms，那么人的操作可能会因为CPU消耗型的进程一直占用

CPU而变的卡顿。可以想象，当bash在等待人输入的时候，是不占CPU的，此时CPU消耗的程序会一直运算，假设每次都分到500ms的时间片，此时人在bash上敲入一个字符的时候，那么bash很可能要等个几百ms才能给出响应，因为在人敲入字符的时候，别的进程的时间片很可能并没有耗尽，所以系统不会调度bash程度进行处理。为了提高IO消耗型进程的响应速度，系统将区分这两类进程，并动态调整CPU消耗的进程将其优先级降低，而IO消耗型的将其优先级变高，以降低CPU消耗进程的时间片的实际长度。已知nice值的范围是-20-19，其对应priority值的范围是100－139，对于一个默认nice值为0的进程来说，其初始priority值应该是120，随着其不断执行，内核会观察进程的CPU消耗状态，并动态调整priority值，可调整的范围是+-5。就是说，最高其优先级可以呗自动调整到115，最低到125。这也是为什么nice值叫做静态优先级而priority值叫做动态优先级的原因。不过这个动态调整的功能在调度器换成CFS之后就不需要了，因为CFS换了另外一种CPU时间分配方式，这个我们后面再说。

再简单了解了O1算法按时间片分配CPU的思路之后，我们再来结合进程的状态简单看看其算法描述。我们都知道进程有5种状态：

S（Interruptible sleep）：可中断休眠状态。

D（Uninterruptible sleep）：不可中断休眠状态。

R（Running or runnable）：执行或者在可执行队列中。

Z（Zombie process）：僵尸。

T（Stopped）：暂停。

在CPU调度时，主要只关心R状态进程，因为其他状态进程并不会被放倒调度队列中进行调度。调度队列中的进程一般主要有两种情况，一种是进程已经被调度到CPU上执行，另一种是进程正在等待被调度。出现这两种状态的原因应该好理解，因为需要执行的进程数可能多于硬件的CPU核心数，比如需要执行的进程有8个而CPU核心只有4个，此时cpu满载的时候，一定会有4个进程处在“等待”状态，因为此时有另外四个进程正在占用CPU执行。

根据以上情况我们可以理解，系统当下需要同时进行调度处理的进程数（R状态进程数）和系统CPU的比值，可以一定程度的反应系统的“繁忙”程度。需要调度的进程越多，核心越少，则意味着系统越繁忙。除了进程执行本身需要占用CPU以外，多个进程的调度切换也会让系统繁忙程度增加的更多。所以，我们往往会发现，R状态进程数量在增长的情况下，系统的性能表现会下降。系统中可以使用uptime命令查看系统平均负载指数（load average）：

```
[zorro@zorrozou-pc0 ~]$ uptime
 16:40:56 up  2:12,  1 user,  load average: 0.05, 0.11, 0.16
```

其中load average中分别显示的是1分钟，5分钟，15分钟之内的平均负载指数（可以简单认为是相映时间范围内的R状态进程个数）。但是这个命令显示的数字是绝对个数，并没有表示出不同CPU核心数的实际情况。比如，如果我们的1分钟load average为16，而CPU核心数为32的话，那么这个系统的其实并不繁忙。但是如果CPU个数是8的话，那可能就意味着比较忙

了。但是实际情况往往可能比这更加复杂，比如进程消耗类型也会对这个数字的解读有影响。总之，这个值的绝对高低并不能直观的反馈出来当前系统的繁忙程度，还需要根据系统的其它指标综合考虑。

O1调度器在处理流程上大概是这样进行调度的：

1. 首先，进程产生（fork）的时候会给一个进程分配一个时间片长度。这个新进程的时间片一般是父进程的一半，而父进程也会因此减少它的时间片长度为原来的一半。就是说，如果一个进程产生了子进程，那么它们将会平分当前时间片长度。比如，如果父进程时间片还剩100ms，那么一个fork产生一个子进程之后，子进程的时间片是50ms，父进程剩余的时间片是也是50ms。这样设计的目的是，为了防止进程通过fork的方式让自己所处理的任务一直有时间片。不过这样做也会带来少许的不公平，因为先产生的子进程获得的时间片将会比后产生的长，第一个子进程分到父进程的一半，那么第二个子进程就只能分到1/4。对于一个长期工作的进程组来说，这种影响可以忽略，因为第一轮时间片在耗尽后，系统会在给它们分配长度相当的时间片。
2. 针对所有R状态进程，O1算法使用两个队列组织进程，其中一个叫做活动队列，另一个叫做过期队列。活动队列中放的都是时间片未被耗尽的进程，而过期队列中放时间片被耗尽的进程。
3. 如1所述，新产生的进程都会先获得一个时间片，进入活动队列等待调度到CPU执行。而内核会在每个tick间隔期间对正在CPU上执行的进程进行检查。一般的tick间隔时间就是cpu时钟中断间隔，每秒钟会有1000个，即频率为1000HZ。每个tick间隔周期主要检查两个内容：1、当前正在占用CPU的进程是不是时间片已经耗尽了？2、是不是有更高优先级的进程在活动队列中等待调度？如果任何一种情况成立，就把则当前进程的执行状态终止，放到等待队列中，换当前在等待队列中优先级最高的那个进程执行。

以上就是O1调度的基本调度思路，当然实际情况是，还要加上SMP（对称多处理）的逻辑，以满足多核CPU的需求。目前在我的archlinux上可以用以下命令查看内核HZ的配置：

```
[zorro@zorrozou-pc0 ~]$ zgrep CONFIG_HZ /proc/config.gz
# CONFIG_HZ_PERIODIC is not set
# CONFIG_HZ_100 is not set
# CONFIG_HZ_250 is not set
CONFIG_HZ_300=y
# CONFIG_HZ_1000 is not set
CONFIG_HZ=300
```

我们发现我当前系统的HZ配置为300，而不是一般情况下的1000。大家也可以思考一下，配置成不同的数字（100、250、300、1000），对系统的性能到底会有什么影响？

CFS完全公平调度

O1已经是上一代调度器了，由于其对多核、多CPU系统的支持性能并不好，并且内核功能上要加入cgroup等因素，Linux在2.6.23之后开始启用CFS作为对一般优先级(SCHED_OTHER)进程调度方法。在这个重新设计的调度器中，时间片，动态、静态优先级以及IO消耗，CPU消耗的概念都不再重要。CFS采用了一种全新的方式，对上述功能进行了比较完善的支持。

其设计的基本思路是，我们想要实现一个对所有进程完全公平的调度器。又是那个老问题：如何做到完全公平？答案跟上一篇IO调度中CFQ的思路类似：如果当前有n个进程需要调度执行，那么调度器应该再一个比较小的时间范围内，把这n个进程全都调度执行一遍，并且它们平分cpu时间，这样就可以做到所有进程的公平调度。那么这个比较小的时间就是任意一个R状态进程被调度的最大延时时间，即：任意一个R状态进程，都一定会在这个时间范围内被调度相应。这个时间也可以叫做调度周期，其英文名字叫做：sched_latency_ns。进程越多，每个进程在周期内被执行的时间就会被平分的越小。调度器只需要对所有进程维护一个累积占用CPU时间数，就可以衡量出每个进程目前占用的CPU时间总量是不是过大或者过小，这个数字记录在每个进程的vruntime中。所有待执行进程都以vruntime为key放到一个由红黑树组成的队列中，每次被调度执行的进程，都是这个红黑树的最左子树上的那个进程，即vruntime时间最少的进程，这样就保证了所有进程的相对公平。

在基本驱动机制上CFS跟O1一样，每次时钟中断来临的时候，都会进行队列调度检查，判断是否要进程调度。当然还有别的时机需要调度检查，发生调度的时机可以总结为这样几个：

1. 当前进程的状态转换时。主要是指当前进程终止退出或者进程休眠的时候。
2. 当前进程主动放弃CPU时。状态变为sleep也可以理解为主动放弃CPU，但是当前内核给了一个方法，可以使用sched_yield()在不发生状态切换的情况下主动让出CPU。
3. 当前进程的vruntime时间大于每个进程的理想占用时间时（delta_exec > ideal_runtime）。这里的ideal_runtime实际上就是上文说的sched_latency_ns／进程数n。当然这个值并不是一定这样得出，下文会有更详细解释。
4. 当进程从中断、异常或系统调用返回时，会发生调度检查。比如时钟中断。

CFS的优先级

当然，CFS中还需要支持优先级。在新的体系中，优先级是以时间消耗（vruntime增长）的快慢来决定的。就是说，对于CFS来说，衡量的时间累积的绝对值都是一样纪录在vruntime中的，但是不同优先级的进程时间增长的比率是不同的，高优先级进程时间增长的慢，低优先级时间增长的快。比如，优先级为19的进程，实际占用cpu为1秒，那么在vruntime中就记录1s。但是如果是-20优先级的进程，那么它很可能实际占CPU用10s，在vruntime中才会纪录1s。CFS眞实实现的不同nice值的cpu消耗时间比例在内核中是按照“每差一级cpu占用时间差10%左右”这个原则来设定的。这里的大概意思是说，如果有两个nice值为0的进程同时占用cpu，那么它们应该每人占50%的cpu，如果将其中一个进程的nice值调整为1的话，那么此时应保证优先级高的进程比低的多占用10%的cpu，就是nice值为0的占55%，nice值为1的占45%。那么它们占用cpu时间的比例为55:45。这个值的比例约为1.25。就是说，相邻的两个nice值之间的cpu占用时间比例的差别应该大约为1.25。根据这个原则，内核对40个nice值做了时间计算比例的对应关系，它在内核中以一个数组存在：

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

我们看到，实际上nice值的最高优先级和最低优先级的时间比例差距还是很大的，绝不仅仅是例子中的十倍。由此我们也可以推导出每一个nice值级别计算vruntime的公式为：

```
delta vruntime ＝ delta Time * 1024 / load
```

这个公式的意思是说，在nice值为0的时候（对应的比例值为1024），计算这个进程vruntime的实际增长时间值（delta vruntime）为：CPU占用时间（delta Time）* 1024 / load。在这个公式中load代表当前sched_entity的值，其实就可以理解为需要调度的进程（R状态进程）个数。load越大，那么每个进程所能分到的时间就越少。CPU调度是内核中会频繁进行处理的一个时间，于是上面的delta vruntime的运算会被频繁计算。除法运算会占用更多的cpu时间，所以内核编程中的一个原则就是，尽可能的不用除法。内核中要用除法的地方，基本都用乘法和位移运算来代替，所以上面这个公式就会变成：

```
delta vruntime ＝ delta time * 1024 * (2^32 / (load * 2^32)) = (delta time * 1024 * Inv
erse（load）) >> 32
```

内核中为了方便不同nice值的Inverse(load)的相关计算，对做好了一个跟prio_to_weight数组一一对应的数组，在计算中可以直接拿来使用，减少计算时的CPU消耗：

```
static const u32 prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

具体计算细节不在这里细解释了，有兴趣的可以自行阅读代码：kernel/shced/fair.c（Linux 4.4）中的__calc_delta（）函数实现。

根据CFS的特性，我们知道调度器总是选择vruntime最小的进程进行调度。那么如果有两个进程的初始化vruntime时间一样时，一个进程被选择进行调度处理，那么只要一进行处理，它的vruntime时间就会大于另一个进程，CFS难道要马上换另一个进程处理么？出于减少频繁切换进程所带来的成本考虑，显然并不应该这样。CFS设计了一个sched_min_granularity_ns参数，用来设定进程被调度执行之后的最小CPU占用时间。

```
[zorro@zorrozou-pc0 ~]$ cat /proc/sys/kernel/sched_min_granularity_ns
2250000
```

一个进程被调度执行后至少要被执行这么长时间才会发生调度切换。

我们知道无论到少个进程要执行，它们都有一个预期延迟时间，即：sched_latency_ns，系统中可以通过如下命令来查看这个时间：

```
[zorro@zorrozou-pc0 ~]$ cat /proc/sys/kernel/sched_latency_ns
18000000
```

在这种情况下，如果需要调度的进程个数为n，那么平均每个进程占用的CPU时间为sched_latency_ns／n。显然，每个进程实际占用的CPU时间会因为n的增大而减小。但是实现上不可能让它无限的变小，所以sched_min_granularity_ns的值也限定了每个进程可以获得的执行时间周期的最小值。当进程很多，导致使用了sched_min_granularity_ns作为最小调度周期时，对应的调度延时也就不在遵循sched_latency_ns的限制，而是以实际的需要调度的进程个数n * sched_min_granularity_ns进行计算。当然，我们也可以把这理解为CFS的"时间片"，不过我们还是要强调，CFS是没有跟O1类似的“时间片“的概念的，具体区别大家可以自己琢磨一下。

新进程的**vruntime**值

CFS是通过vruntime最小值来选择需要调度的进程的，那么可以想象，在一个已经有多个进程执行了相对较长的系统中，这个队列中的vruntime时间纪录的数值都会比较长。如果新产生的进程直接将自己的vruntime值设置为0的话，那么它将在执行开始的时间内抢占很多的CPU时间，直到自己的vruntime追赶上其他进程后才可能调度其他进程，这种情况显然是不公平的。所以CFS对每个CPU的执行队列都维护一个min_vruntime值，这个值纪录了这个CPU执行队列中vruntime的最小值，当队列中出现一个新建的进程时，它的初始化vruntime将不会被设置为0，而是根据min_vruntime的值为基础来设置。这样就保证了新建进程的vruntime与老进程的差距在一定范围内，不会因为vruntime设置为0而在进程开始的时候占用过多的CPU。

新建进程获得的实际vruntime值跟一些设置有关，比如：

```
[zorro@zorrozou-pc0 ~]$ cat /proc/sys/kernel/sched_child_runs_first
0
```

这个文件是fork之后是否让子进程优先于父进程执行的开关。0为关闭，1为打开。如果这个开关打开，就意味着子进程创建后，保证子进程在父进程之前被调度。另外，在源代码目录下的kernel/sched/features.h文件中，还规定了一系列调度器属性开关。而其中：

```
/*
 * Place new tasks ahead so that they do not starve already running
 * tasks
 */
SCHED_FEAT(START_DEBIT, true)
```

这个参数规定了新进程启动之后第一次运行会有延时。这意味着新进程的vruntime设置要比默认值大一些，这样做的目的是防止应用通过不停的fork来尽可能多的获得执行时间。子进程在创建的时候，vruntime的定义的步骤如下，首先vruntime被设置为min_vruntime。然后判断START_DEBIT位是否被值为true，如果是则会在min_vruntime的基础上增大一些，增大的时间实际上就是一个进程的调度延时时间，即上面描述过的calc_delta_fair()函数得到的结果。这个时间设置完毕之后，就检查sched_child_runs_first开关是否打开，如果打开（值被设置为1），就比较新进程的vruntime和父进程的vruntime哪个更小，并将新进程的vruntime设置为更小的那个值，而父进程的vruntime设置为更大的那个值，以此保证子进程一定在父进程之前被调度。

IO消耗型进程的处理

根据前文，我们知道除了可能会一直占用CPU时间的CPU消耗型进程以外，还有一类叫做IO消耗类型的进程，它们的特点是基本不占用CPU，主要行为是在S状态等待响应。这类进程典型的是vim，bash等跟人交互的进程，以及一些压力不大的，使用了多进程（线程）的或select、poll、epoll的网络代理程序。如果CFS采用默认的策略处理这些程序的话，相比CPU消耗程序来说，这些应用由于绝大多数时间都处在sleep状态，它们的vruntime时间基本是不变的，一旦它们进入了调度队列，将会很快被选择调度执行。对比O1调度算法，这种行为相当于自然的提高了这些IO消耗型进程的优先级，于是就不需要特殊对它们的优先级进行“动态调整”了。

但这样的默认策略也是有问题的，有时CPU消耗型和IO消耗型进程的区分不是那么明显，有些进程可能会等一会，然后调度之后也会长时间占用CPU。这种情况下，如果休眠的时候进程的vruntime保持不变，那么等到休眠被唤醒之后，这个进程的vruntime时间就可能会比别人小很多，从而导致不公平。所以对于这样的进程，CFS也会对其进行时间补偿。补偿方式为，如果进程是从sleep状态被唤醒的，而且GENTLE_FAIR_SLEEPERS属性的值为true，则vruntime被设置为sched_latency_ns的一半和当前进程的vruntime值中比较大的那个。sched_latency_ns的值可以在这个文件中进行设置：

```
[zorro@zorrozou-pc0 ~]$ cat /proc/sys/kernel/sched_latency_ns
18000000
```

因为系统中这种调度补偿的存在，IO消耗型的进程总是可以更快的获得响应速度。这是CFS处理与人交互的进程时的策略，即：通过提高响应速度让人的操作感受更好。但是有时候也会因为这样的策略导致整体性能受损。在很多使用了多进程（线程）或select、poll、epoll的网络代理程序，一般是由多个进程组成的进程组进行工作，典型的如apche、nginx和php-fpm这样的处理程序。它们往往都是由一个或者多个进程使用nanosleep()进行周期性的检查是否有新任务，如果有责唤醒一个子进程进行处理，子进程的处理可能会消耗CPU，而父进程则主要是sleep等待唤醒。这个时候，由于系统对sleep进程的补偿策略的存在，新唤醒的进程就可能会打断正在处理的子进程的过程，抢占CPU进行处理。当这种打断很多很频繁的时候，CPU处理的过程就会因为频繁的进程上下文切换而变的很低效，从而使系统整体吞吐量下降。此时我们可以使用开关禁止唤醒抢占的特性。

```
[root@zorrozou-pc0 zorro]# cat /sys/kernel/debug/sched_features
GENTLE_FAIR_SLEEPERS START_DEBIT NO_NEXT_BUDDY LAST_BUDDY CACHE_HOT_BUDDY WAKEUP_PREEM
PTION NO_HRTICK NO_DOUBLE_TICK LB_BIAS NONTASK_CAPACITY TTWU_QUEUE RT_PUSH_IPI NO_FORC
E_SD_OVERLAP RT_RUNTIME_SHARE NO_LB_MIN ATTACH_AGE_LOAD
```

上面显示的这个文件的内容就是系统中用来控制kernel/sched/features.h这个文件所列内容的开关文件，其中WAKEUP_PREEMPTION表示：目前的系统状态是打开sleep唤醒进程的抢占属性的。可以使用如下命令关闭这个属性：

```
[root@zorrozou-pc0 zorro]# echo NO_WAKEUP_PREEMPTION > /sys/kernel/debug/sched_feature
s
[root@zorrozou-pc0 zorro]# cat /sys/kernel/debug/sched_features
GENTLE_FAIR_SLEEPERS START_DEBIT NO_NEXT_BUDDY LAST_BUDDY CACHE_HOT_BUDDY NO_WAKEUP_PR
EEMPTION NO_HRTICK NO_DOUBLE_TICK LB_BIAS NONTASK_CAPACITY TTWU_QUEUE RT_PUSH_IPI NO_F
ORCE_SD_OVERLAP RT_RUNTIME_SHARE NO_LB_MIN ATTACH_AGE_LOAD
```

其他相关参数的调整也是类似这样的方式。其他我没讲到的属性的含义，大家可以看kernel/sched/features.h文件中的注释。

系统中还提供了一个sched_wakeup_granularity_ns配置文件，这个文件的值决定了唤醒进程是否可以抢占的一个时间粒度条件。默认CFS的调度策略是，如果唤醒的进程vruntime小于当前正在执行的进程，那么就会发生唤醒进程抢占的情况。而sched_wakeup_granularity_ns这个参数是说，只有在当前进程的vruntime时间减唤醒进程的vruntime时间所得的差大于sched_wakeup_granularity_ns时，才回发生抢占。就是说sched_wakeup_granularity_ns的值越大，越不容易发生抢占。

CFS和其他调度策略

SCHED_BATCH

在上文中我们说过，CFS调度策略主要是针对chrt命令显示的SCHED_OTHER范围的进程，实际上就是一般的非实时进程。我们也已经知道，这样的一般进程还包括另外两种：SCHED_BATCH和SCHED_IDLE。在CFS的实现中，集成了对SCHED_BATCH策略的支持，并且其功能和SCHED_OTHER策略几乎是一致的。唯一的区别在于，如果一个进程被用chrt命令标记成SCHED_OTHER策略的话，CFS将永远认为这个进程是CPU消耗型的进程，不会对其进行IO消耗进程的时间补偿。这样做的唯一目的是，可以在确认进程是CPU消耗型的进程的前提下，对其尽可能的进行批处理方式调度（batch），以减少进程切换带来的损耗，提高吞度量。实际上这个策略的作用并不大，内核中真正的处理区别只是在标记为SCHED_BATCH时进程在sched_yield主动让出cpu的行为发生是不去更新cfs的队列时间，这样就让这些进程在主动让出CPU的时候（执行sched_yield）不会纪录其vruntime的更新，从而可以继续优先被调度到。对于其他行为，并无不同。

SCHED_IDLE

如果一个进程被标记成了SCHED_IDLE策略，调度器将认为这个优先级是很低很低的，比nice值为19的优先级还要低。系统将只在CPU空闲的时候才会对这样的进程进行调度执行。若果存在多个这样的进程，它们之间的调度方式跟正常的CFS相同。

SCHED_DEADLINE

最新的Linux内核还实现了一个最新的调度方式叫做SCHED_DEADLINE。跟IO调度类似，这个算法也是要实现一个可以在最终期限到达前让进程可以调度执行的方法，保证进程不会饿死。目前大多数系统上的chrt还没给配置接口，暂且不做深入分析。

另外要注意的是，SCHED_BATCH和SCHED_IDLE一样，只能对静态优先级（即nice值）为0的进程设置。操作命令如下：

```
[zorro@zorrozou-pc0 ~]$ chrt -i 0 bash
[zorro@zorrozou-pc0 ~]$ chrt -p $$
pid 5478's current scheduling policy: SCHED_IDLE
pid 5478's current scheduling priority: 0

[zorro@zorrozou-pc0 ~]$ chrt -b 0 bash
[zorro@zorrozou-pc0 ~]$ chrt -p $$
pid 5502's current scheduling policy: SCHED_BATCH
pid 5502's current scheduling priority: 0
```

多CPU的CFS调度

在上面的叙述中，我们可以认为系统中只有一个CPU，那么相关的调度队列只有一个。实际情况是系统是有多核甚至多个CPU的，CFS从一开始就考虑了这种情况，它对每个CPU核心都维护一个调度队列，这样每个CPU都对自己的队列进程调度即可。这也是CFS比O1调度算法更高效的根本原因：每个CPU一个队列，就可以避免对全局队列使用大内核锁，从而提高了并行效率。当然，这样最直接的影响就是CPU之间的负载可能不均，为了维持CPU之间的

负载均衡，CFS要定期对所有CPU进行load balance操作，于是就有可能发生进程在不同CPU的调度队列上切换的行为。这种操作的过程也需要对相关的CPU队列进行锁操作，从而降低了多个运行队列带来的并行性。不过总的来说，CFS的并行队列方式还是要比O1的全局队列方式要高效。尤其是在CPU核心越来越多的情况下，全局锁的效率下降显著增加。

CFS对多个CPU进行负载均衡的行为是idle_balance()函数实现的，这个函数会在CPU空闲的时候由schedule()进行调用，让空闲的CPU从其他繁忙的CPU队列中取进程来执行。我们可以通过查看/proc/sched_debug的信息来查看所有CPU的调度队列状态信息以及系统中所有进程的调度信息。内容较多，我就不在这里一一列出了，有兴趣的同学可以自己根据相关参考资料（最好的资料就是内核源码）了解其中显示的相关内容分别是什么意思。

在CFS对不同CPU的调度队列做均衡的时候，可能会将某个进程切换到另一个CPU上执行。此时，CFS会在将这个进程出队的时候将vruntime减去当前队列的min_vruntime，其差值作为结果会在入队另一个队列的时候再加上所入队列的min_vruntime，以此来保持队列切换后CPU队列的相对公平。

最后

本文的目的是从Linux系统进程的优先级为出发点，通过了解相关的知识点，希望大家对系统的进程调度有个整体的了解。其中我们也对CFS调度算法进行了比较深入的分析。在我的经验来看，这些知识对我们在观察系统的状态和相关优化的时候都是非常有用的。比如在使用top命令的时候，NI和PR值到底是什么意思？类似的地方还有ps命令中的NI和PRI值、ulimit命令-e和-r参数的区别等等。当然，希望看完本文后，能让大家对这些命令显示的了解更加深入。除此之外，我们还会发现，虽然top命令中的PR值和ps -l命令中的PRI值的含义是一样的，但是在优先级相同的情况下，它们显示的值确不一样。那么你知道为什么它们显示会有区别吗？这个问题的答案留给大家自己去寻找吧。

Linux内存中的Cache真的能被回收么？

在Linux系统中，我们经常用free命令来查看系统内存的使用状态。在一个RHEL6的系统上，free命令的显示内容大概是这样一个状态：

```
[root@tencent64 ~]# free
             total       used       free     shared    buffers     cached
Mem:     132256952   72571772   59685180          0    1762632   53034704
-/+ buffers/cache:   17774436  114482516
Swap:      2101192        508    2100684
```

这里的默认显示单位是kb，我的服务器是128G内存，所以数字显得比较大。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，对此命令输出的理解可以分这几个层次：

1. 不了解。这样的人的第一反应是：天啊，内存用了好多，70个多G，可是我几乎没有运行什么大程序啊？为什么会这样？Linux好占内存！
2. 自以为很了解。这样的人一般自习评估过会说：嗯，根据我专业的眼光看出来，内存才用了17G左右，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. 真的很了解。这种人的反应反而让人感觉最不懂Linux，他们的反应是：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？

根据目前网络上技术文档的内容，我相信绝大多数了解一点Linux的人应该处在第二种层次。大家普遍认为，buffers和cached所占用的内存空间是可以在内存压力较大的时候被释放当做空闲空间用的。但真的是这样么？在论证这个题目之前，我们先简要介绍一下buffers和cached是什么意思：

什么是buffer/cache？

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Linux的内存管理中，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作为其缓存来管理使用。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。

明白了这两套缓存系统的区别，就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？在当前的系统实现里，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。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。

如何回收**cache**？

Linux内核会在内存将要耗尽的时候，触发内存回收的工作，以便释放出内存给急需内存的进程使用。一般情况下，这个操作中主要的内存释放都来自于对buffer／cache的释放。尤其是被使用更多的cache空间。既然它主要用来做缓存，只是在内存够用的时候加快进程对文件的读写速度，那么在内存压力较大的情况下，当然有必要清空释放cache，作为free空间分给相关进程使用。所以一般情况下，我们认为buffer/cache空间可以被释放，这个理解是正确的。

但是这种清缓存的工作也并不是没有成本。理解cache是干什么的就可以明白清缓存必须保证cache中的数据跟对应文件中的数据一致，才能对cache进行释放。所以伴随着**cache**清除的行为的，一般都是系统**IO**飙高。因为内核要对比cache中的数据和对应硬盘文件上的数据是否一致，如果不一致需要写回，之后才能回收。

在系统中除了内存将被耗尽的时候可以清缓存以外，我们还可以使用下面这个文件来人工触发缓存清除的操作：

```
[root@tencent64 ~]# cat /proc/sys/vm/drop_caches
1
```

方法是：

```
echo 1 > /proc/sys/vm/drop_caches
```

当然，这个文件可以设置的值分别为1、2、3。它们所表示的含义为：
echo 1 > /proc/sys/vm/drop_caches:表示清除pagecache。

echo 2 > /proc/sys/vm/drop_caches:表示清除回收slab分配器中的对象（包括目录项缓存和inode缓存）。slab分配器是内核中管理内存的一种机制，其中很多缓存数据实现都是用的pagecache。

echo 3 > /proc/sys/vm/drop_caches:表示清除pagecache和slab分配器中的缓存对象。

cache都能被回收么？

我们分析了cache能被回收的情况，那么有没有不能被回收的cache呢？当然有。我们先来看第一种情况：

tmpfs

大家知道Linux提供一种“临时”文件系统叫做tmpfs，它可以将内存的一部分空间拿来当做文件系统使用，使内存空间可以当做目录文件来用。现在绝大多数Linux系统都有一个叫做/dev/shm的tmpfs目录，就是这样一种存在。当然，我们也可以手工创建一个自己的tmpfs，方法如下：

```
[root@tencent64 ~]# mkdir /tmp/tmpfs
[root@tencent64 ~]# mount -t tmpfs -o size=20G none /tmp/tmpfs/

[root@tencent64 ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             10325000   3529604   6270916  37% /
/dev/sda3             20646064   9595940  10001360  49% /usr/local
/dev/mapper/vg-data  103212320  26244284  71725156  27% /data
tmpfs                 66128476  14709004  51419472  23% /dev/shm
none                  20971520         0  20971520   0% /tmp/tmpfs
```

于是我们就创建了一个新的tmpfs，空间是20G，我们可以在/tmp/tmpfs中创建一个20G以内的文件。如果我们创建的文件实际占用的空间是内存的话，那么这些数据应该占用内存空间的什么部分呢？根据pagecache的实现功能可以理解，既然是某种文件系统，那么自然该使用pagecache的空间来管理。我们试试是不是这样？

```
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         36         89          0          1         19
-/+ buffers/cache:         15        111
Swap:            2          0          2
[root@tencent64 ~]# dd if=/dev/zero of=/tmp/tmpfs/testfile bs=1G count=13
13+0 records in
13+0 records out
13958643712 bytes (14 GB) copied, 9.49858 s, 1.5 GB/s
[root@tencent64 ~]#
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         49         76          0          1         32
-/+ buffers/cache:         15        110
Swap:            2          0          2
```

我们在tmpfs目录下创建了一个13G的文件，并通过前后free命令的对比发现，cached增长了13G，说明这个文件确实放在了内存里并且内核使用的是cache作为存储。再看看我们关心的指标： -/+ buffers/cache那一行。我们发现，在这种情况下free命令仍然提示我们有110G内存可用，但是真的有这么多么？我们可以人工触发内存回收看看现在到底能回收多少内存：

```
[root@tencent64 ~]# echo 3 > /proc/sys/vm/drop_caches
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         43         82          0          0         29
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

可以看到，cached占用的空间并没有像我们想象的那样完全被释放，其中13G的空间仍然被/tmp/tmpfs中的文件占用的。当然，我的系统中还有其他不可释放的cache占用着其余16G内存空间。那么tmpfs占用的cache空间什么时候会被释放呢？是在其文件被删除的时候.如果不删除文件，无论内存耗尽到什么程度，内核都不会自动帮你把tmpfs中的文件删除来释放cache空间。

```
[root@tencent64 ~]# rm /tmp/tmpfs/testfile
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         30         95          0          0         16
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

这是我们分析的第一种cache不能被回收的情况。还有其他情况，比如：

共享内存

共享内存是系统提供给我们的一种常用的进程间通信（IPC）方式，但是这种通信方式不能在shell中申请和使用，所以我们需要一个简单的测试程序，代码如下：

```
[root@tencent64 ~]# cat shm.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define MEMSIZE 2048*1024*1023

int
main()
{
    int shmid;
    char *ptr;
    pid_t pid;
    struct shmid_ds buf;
    int ret;

    shmid = shmget(IPC_PRIVATE, MEMSIZE, 0600);
    if (shmid<0) {
        perror("shmget()");
        exit(1);
    }

    ret = shmctl(shmid, IPC_STAT, &buf);
    if (ret < 0) {
        perror("shmctl()");
        exit(1);
    }

    printf("shmid: %d\n", shmid);
    printf("shmsize: %d\n", buf.shm_segsz);

    buf.shm_segsz *= 2;

    ret = shmctl(shmid, IPC_SET, &buf);
    if (ret < 0) {
        perror("shmctl()");
        exit(1);
    }

    ret = shmctl(shmid, IPC_SET, &buf);
    if (ret < 0) {
        perror("shmctl()");
        exit(1);
    }
```

```
    printf("shmid: %d\n", shmid);
    printf("shmsize: %d\n", buf.shm_segsz);


    pid = fork();
    if (pid<0) {
        perror("fork()");
        exit(1);
    }
    if (pid==0) {
        ptr = shmat(shmid, NULL, 0);
        if (ptr==(void*)-1) {
            perror("shmat()");
            exit(1);
        }
        bzero(ptr, MEMSIZE);
        strcpy(ptr, "Hello!");
        exit(0);
    } else {
        wait(NULL);
        ptr = shmat(shmid, NULL, 0);
        if (ptr==(void*)-1) {
            perror("shmat()");
            exit(1);
        }
        puts(ptr);
        exit(0);
    }
}
```

程序功能很简单，就是申请一段不到2G共享内存，然后打开一个子进程对这段共享内存做一个初始化操作，父进程等子进程初始化完之后输出一下共享内存的内容，然后退出。但是退出之前并没有删除这段共享内存。我们来看看这个程序执行前后的内存使用：

```
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         30         95          0          0         16
-/+ buffers/cache:         14        111
Swap:            2          0          2
[root@tencent64 ~]# ./shm
shmid: 294918
shmsize: 2145386496
shmid: 294918
shmsize: -4194304
Hello!
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         32         93          0          0         18
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

cached空间由16G涨到了18G。那么这段cache能被回收么？继续测试：

```
[root@tencent64 ~]# echo 3 > /proc/sys/vm/drop_caches
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         32         93          0          0         18
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

结果是仍然不可回收。大家可以观察到，这段共享内存即使没人使用，仍然会长期存放在cache中，直到其被删除。删除方法有两种，一种是程序中使用shmctl()去IPC_RMID，另一种是使用ipcrm命令。我们来删除试试：

```
[root@tencent64 ~]# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00005feb 0          root       666        12000      4
0x00005fe7 32769      root       666        524288     2
0x00005fe8 65538      root       666        2097152    2
0x00038c0e 131075     root       777        2072       1
0x00038c14 163844     root       777        5603392    0
0x00038c09 196613     root       777        221248     0
0x00000000 294918     root       600        2145386496 0

[root@tencent64 ~]# ipcrm -m 294918
[root@tencent64 ~]# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00005feb 0          root       666        12000      4
0x00005fe7 32769      root       666        524288     2
0x00005fe8 65538      root       666        2097152    2
0x00038c0e 131075     root       777        2072       1
0x00038c14 163844     root       777        5603392    0
0x00038c09 196613     root       777        221248     0

[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         30         95          0          0         16
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

删除共享内存后，cache被正常释放了。这个行为与tmpfs的逻辑类似。内核底层在实现共享内存（shm）、消息队列（msg）和信号量数组（sem）这些POSIX:XSI的IPC机制的内存存储时，使用的都是tmpfs。这也是为什么共享内存的操作逻辑与tmpfs类似的原因。当然，一般情况下是shm占用的内存更多，所以我们在此重点强调共享内存的使用。说到共享内存，Linux还给我们提供了另外一种共享内存的方法，就是：

mmap

mmap()是一个非常重要的系统调用，这仅从mmap本身的功能描述上是看不出来的。从字面上看，mmap就是将一个文件映射进进程的虚拟内存地址，之后就可以通过操作内存的方式对文件的内容进行操作。但是实际上这个调用的用途是很广泛的。当malloc申请内存时，小段内存内核使用sbrk处理，而大段内存就会使用mmap。当系统调用exec族函数执行时，因为其本质上是将一个可执行文件加载到内存执行，所以内核很自然的就可以使用mmap方式进行处理。我们在此仅仅考虑一种情况，就是使用mmap进行共享内存的申请时，会不会跟shmget()一样也使用cache？

同样，我们也需要一个简单的测试程序：

```
[root@tencent64 ~]# cat mmap.c
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

#define MEMSIZE 1024*1024*1023*2
#define MPFILE "./mmapfile"

int main()
{
    void *ptr;
    int fd;

    fd = open(MPFILE, O_RDWR);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }

    ptr = mmap(NULL, MEMSIZE, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, fd, 0);
    if (ptr == NULL) {
        perror("malloc()");
        exit(1);
    }

    printf("%p\n", ptr);
    bzero(ptr, MEMSIZE);

    sleep(100);

    munmap(ptr, MEMSIZE);
    close(fd);

    exit(1);
}
```

这次我们干脆不用什么父子进程的方式了，就一个进程，申请一段2G的mmap共享内存，然后初始化这段空间之后等待100秒，再解除影射所以我们需要在它sleep这100秒内检查我们的系统内存使用，看看它用的是什么空间？当然在这之前要先创建一个2G的文件./mmapfile。结果如下：

```
[root@tencent64 ~]# dd if=/dev/zero of=mmapfile bs=1G count=2
[root@tencent64 ~]# echo 3 > /proc/sys/vm/drop_caches
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         30         95          0          0         16
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

然后执行测试程序：

```
[root@tencent64 ~]# ./mmap &
[1] 19157
0x7f1ae3635000
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         32         93          0          0         18
-/+ buffers/cache:         14        111
Swap:            2          0          2

[root@tencent64 ~]# echo 3 > /proc/sys/vm/drop_caches
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         32         93          0          0         18
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

我们可以看到，在程序执行期间，cached一直为18G，比之前涨了2G，并且此时这段cache仍然无法被回收。然后我们等待100秒之后程序结束。

```
[root@tencent64 ~]#
[1]+  Exit 1                  ./mmap
[root@tencent64 ~]#
[root@tencent64 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           126         30         95          0          0         16
-/+ buffers/cache:         14        111
Swap:            2          0          2
```

程序退出之后，cached占用的空间被释放。这样我们可以看到，使用mmap申请标志状态为MAP_SHARED的内存，内核也是使用的cache进行存储的。在进程对相关内存没有释放之前，这段cache也是不能被正常释放的。实际上，mmap的MAP_SHARED方式申请的内存，在内核中也是由tmpfs实现的。由此我们也可以推测，由于共享库的只读部分在内存中都是以mmap的MAP_SHARED方式进行管理，实际上它们也都是要占用cache且无法被释放的。

最后

我们通过三个测试例子，发现Linux系统内存中的cache并不是在所有情况下都能被释放当做空闲空间用的。并且也也明确了，即使可以释放cache，也并不是对系统来说没有成本的。总结一下要点，我们应该记得这样几点：

1. 当cache作为文件缓存被释放的时候会引发IO变高，这是cache加快文件访问速度所要付出的成本。
2. tmpfs中存储的文件会占用cache空间，除非文件删除否则这个cache不会被自动释放。
3. 使用shmget方式申请的共享内存会占用cache空间，除非共享内存被ipcrm或者使用shmctl去IPC_RMID，否则相关的cache空间都不会被自动释放。
4. 使用mmap方法申请的MAP_SHARED标志的内存会占用cache空间，除非进程将这段内存munmap，否则相关的cache空间都不会被自动释放。
5. 实际上shmget、mmap的共享内存，在内核层都是通过tmpfs实现的，tmpfs实现的存储用的都是cache。

当理解了这些的时候，希望大家对free命令的理解可以达到我们说的第三个层次。我们应该明白，内存的使用并不是简单的概念，cache也并不是真的可以当成空闲空间用的。如果我们要真正深刻理解你的系统上的内存到底使用的是否合理，是需要理解清楚很多更细节知识，并且对相关业务的实现做更细节判断的。我们当前实验场景是Centos 6的环境，不同版本的Linux的free现实的状态可能不一样，大家可以自己去找出不同的原因。

当然，本文所述的也不是所有的cache不能被释放的情形。那么，在你的应用场景下，还有那些cache不能被释放的场景呢？

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：


@orroz
weibo.com/30007147

Linux的内存回收和交换

前言

Linux的swap相关部分代码从2.6早期版本到现在的4.6版本在细节之处已经有不少变化。本文讨论的swap基于Linux 4.4内核代码。Linux内存管理是一套非常复杂的系统，而swap只是其中一个很小的处理逻辑。希望本文能让读者了解Linux对swap的使用大概是什么样子。阅读完本文，应该可以帮你解决以下问题：

1. swap到底是干嘛的？
2. swappiness到底是用来调节什么的？
3. 什么是内存水位标记？
4. kswapd什么时候会进行swap操作？
5. swap分区的优先级（priority）有啥用？

什么是SWAP？

我们一般所说的swap，指的是一个交换分区或文件。在Linux上可以使用swapon -s命令查看当前系统上正在使用的交换空间有哪些，以及相关信息：

```
[zorro@zorrozou-pc0 linux-4.4]$ swapon -s
Filename                Type        Size    Used    Priority
/dev/dm-4                                   partition    33554428    0    -1
```

从功能上讲，交换分区主要是在内存不够用的时候，将部分内存上的数据交换到swap空间上，以便让系统不会因内存不够用而导致oom或者更致命的情况出现。所以，当内存使用存在压力，开始触发内存回收的行为时，就可能会使用swap空间。内核对swap的使用实际上是跟内存回收行为紧密结合的。那么内存回收和swap的关系，我们可以提出以下几个问题：

1. 什么时候会进行内存回收呢？
2. 哪些内存会可能被回收呢？
3. 回收的过程中什么时候会进行交换呢？
4. 具体怎么交换？

下面我们就从这些问题出发，一个一个进行分析。

内存回收

内核之所以要进行内存回收，主要原因有两个：

第一、内核需要为任何时刻突发到来的内存申请提供足够的内存。所以一般情况下保证有足够的free空间对于内核来说是必要的。另外，Linux内核使用cache的策略虽然是不用白不用，内核会使用内存中的page cache对部分文件进行缓存，以便提升文件的读写效率。所以内核有必要设计一个周期性回收内存的机制，以便cache的使用和其他相关内存的使用不至于让系统的剩余内存长期处于很少的状态。

第二，当真的有大于空闲内存的申请到来的时候，会触发强制内存回收。

所以，内核在应对这两类回收的需求下，分别实现了两种不同的机制。一个是使用kswapd进程对内存进行周期检查，以保证平常状态下剩余内存尽可能够用。另一个是直接内存回收（direct page reclaim），就是当内存分配时没有空闲内存可以满足要求时，触发直接内存回收。

这两种内存回收的触发路径不同，一个是由内核进程kswapd直接调用内存回收的逻辑进行内存回收（参见mm/vmscan.c中的kswapd()主逻辑），另一个是内存申请的时候进入slow path的内存申请逻辑进行回收（参见内核代码中的mm/page_alloc.c中的__alloc_pages_slowpath方法）。这两个方法中实际进行内存回收的过程殊途同归，最终都是调用shrink_zone()方法进行针对每个zone的内存页缩减。这个方法中会再调用shrink_lruvec()这个方法对每个组织页的链表进程检查。找到这个线索之后，我们就可以清晰的看到内存回收操作究竟针对的page有哪些了。这些链表主要定义在mm/vmscan.c一个enum中：

```
#define LRU_BASE 0
#define LRU_ACTIVE 1
#define LRU_FILE 2

enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};
```

根据这个enum可以看到，内存回收主要需要进行扫描的包括anon的inactive和active以及file的inactive和active四个链表。就是说，内存回收操作主要针对的就是内存中的文件页（file cache）和匿名页。关于活跃（active）还是不活跃（inactive）的判断内核会使用lru算法进行处理并进行标记，我们这里不详细解释这个过程。

整个扫描的过程分几个循环，首先扫描每个zone上的cgroup组。然后再以cgroup的内存为单元进行page链表的扫描。内核会先扫描anon的active链表，将不频繁的放进inactive链表中，然后扫描inactive链表，将里面活跃的移回active中。进行swap的时候，先对inactive的页进行换出。如果是file的文件映射page页，则判断其是否为脏数据，如果是脏数据就写回，不是脏数据可以直接释放。

这样看来，内存回收这个行为会对两种内存的使用进行回收，一种是anon的匿名页内存，主要回收手段是swap，另一种是file-backed的文件映射页，主要的释放手段是写回和清空。因为针对file based的内存，没必要进行交换，其数据原本就在硬盘上，回收这部分内存只要在有脏数据时写回，并清空内存就可以了，以后有需要再从对应的文件读回来。内存对匿名页和文件缓存一共用了四条链表进行组织，回收过程主要是针对这四条链表进行扫描和操作。

swappiness的作用究竟是什么？

我们应该都知道/proc/sys/vm/swappiness这个文件，是个可以用来调整跟swap相关的参数。这个文件的默认值是60，可以的取值范围是0-100。这很容易给大家一个暗示：我是个百分比哦！那么这个文件具体到底代表什么意思呢？我们先来看一下说明：

> ==========
>
> swappiness
>
> This control is used to define how aggressive the kernel will swap memory pages. Higher values will increase agressiveness, lower values decrease the amount of swap. A value of 0 instructs the kernel not to initiate swap until the amount of free and file-backed pages is less than the high water mark in a zone.
>
> The default value is 60.
>
> ==========

这个文件的值用来定义内核使用swap的积极程度。值越高，内核就会越积极的使用swap，值越低就会降低对swap的使用积极性。如果这个值为0，那么内存在free和file-backed使用的页面总量小于高水位标记（high water mark）之前，不会发生交换。

在这里我们可以理解file-backed这个词的含义了，实际上就是上文所说的文件映射页的大小。那么这个swappiness到底起到了什么作用呢？我们换个思路考虑这个事情。假设让我们设计一个内存回收机制，要去考虑将一部分内存写到swap分区上，将一部分file-backed的内存写

回并清空，剩余部分内存出来，我们将怎么设计？

我想应该主要考虑这样几个问题。

1. 如果回收内存可以有两种途径（匿名页交换和file缓存清空），那么我应该考虑在本次回收的时候，什么情况下多进行file写回，什么情况下应该多进行swap交换。说白了就是平衡两种回收手段的使用，以达到最优。
2. 如果符合交换条件的内存较长，是不是可以不用全部交换出去？比如可以交换的内存有100M，但是目前只需要50M内存，实际只要交换50M就可以了，不用把能交换的都交换出去。

分析代码会发现，Linux内核对这部分逻辑的实现代码在get_scan_count()这个方法中，这个方法被shrink_lruvec()调用。get_sacn_count()就是处理上述逻辑的，swappiness是它所需要的一个参数，这个参数实际上是指导内核在清空内存的时候，是更倾向于清空file-backed内存还是更倾向于进行匿名页的交换的。当然，这只是个倾向性，是指在两个都够用的情况下，更愿意用哪个，如果不够用了，那么该交换还是要交换。

简单看一下get_sacn_count()函数的处理部分代码，其中关于swappiness的第一个处理是：

```
/*
 * With swappiness at 100, anonymous and file have the same priority.
 * This scanning priority is essentially the inverse of IO cost.
 */
anon_prio = swappiness;
file_prio = 200 - anon_prio;
```

这里注释的很清楚，如果swappiness设置为100，那么匿名页和文件将用同样的优先级进行回收。很明显，使用清空文件的方式将有利于减轻内存回收时可能造成的IO压力。因为如果file-backed中的数据不是脏数据的话，那么可以不用写回，这样就没有IO发生，而一旦进行交换，就一定会造成IO。所以系统默认将swappiness的值设置为60，这样回收内存时，对file-backed的文件cache内存的清空比例会更大，内核将会更倾向于进行缓存清空而不是交换。

这里的swappiness值如果是60，那么是不是说内核回收的时候，会按照60:140的比例去做相应的swap和清空file-backed的空间呢？并不是。在做这个比例计算的时候，内核还要参考当前内存使用的其他信息。对这里具体是怎么处理感兴趣的人，可以自己详细看get_sacn_count()的实现，本文就不多解释了。我们在此要明确的概念是：**swappiness**的值是用来控制内存回收时，回收的匿名页更多一些还是回收的**file cache**更多一些。

那么swappiness设置为0的话，是不是内核就根本不会进行swap了呢？这个答案也是否定的。首先是内存真的不够用的时候，该swap的话还是要swap。其次在内核中还有一个逻辑会导致直接使用swap，内核代码是这样处理的：

```
/*
 * Prevent the reclaimer from falling into the cache trap: as
 * cache pages start out inactive, every cache fault will tip
 * the scan balance towards the file LRU.  And as the file LRU
 * shrinks, so does the window for rotation from references.
 * This means we have a runaway feedback loop where a tiny
 * thrashing file LRU becomes infinitely more attractive than
 * anon pages.  Try to detect this based on file LRU size.
 */
if (global_reclaim(sc)) {
        unsigned long zonefile;
        unsigned long zonefree;

        zonefree = zone_page_state(zone, NR_FREE_PAGES);
        zonefile = zone_page_state(zone, NR_ACTIVE_FILE) +
                   zone_page_state(zone, NR_INACTIVE_FILE);

        if (unlikely(zonefile + zonefree <= high_wmark_pages(zone))) {
                scan_balance = SCAN_ANON;
                goto out;
        }
}
```

这里的逻辑是说，如果触发的是全局回收，并且zonefile + zonefree <= high_wmark_pages(zone)条件成立时，就将scan_balance这个标记置为SCAN_ANON。后续处理scan_balance的时候，如果它的值是SCAN_ANON，则一定会进行针对匿名页的swap操作。要理解这个行为，我们首先要搞清楚什么是高水位标记（high_wmark_pages）。

内存水位标记(watermark)

我们回到kswapd周期检查和直接内存回收的两种内存回收机制。直接内存回收比较好理解，当申请的内存大于剩余内存的时候，就会触发直接回收。那么kswapd进程在周期检查的时候触发回收的条件是什么呢？还是从设计角度来看，kswapd进程要周期对内存进行检测，达到一定阈值的时候开始进行内存回收。这个所谓的阈值可以理解为内存目前的使用压力，就是说，虽然我们还有剩余内存，但是当剩余内存比较小的时候，就是内存压力较大的时候，就应该开始试图回收些内存了，这样才能保证系统尽可能的有足够的内存给突发的内存申请所使用。

那么如何描述内存使用的压力呢？Linux内核使用水位标记（watermark）的概念来描述这个压力情况。Linux为内存的使用设置了三种内存水位标记，high、low、min。他们所标记的分别含义为：剩余内存在high以上表示内存剩余较多，目前内存使用压力不大；high-low的范围表示目前剩余内存存在一定压力；low-min表示内存开始有较大使用压力，剩余内存不多了；min是最小的水位标记，当剩余内存达到这个状态时，就说明内存面临很大压力。小于min这部分内存，内核是保留给特定情况下使用的，一般不会分配。内存回收行为就是基于剩余内

存的水位标记进行决策的，当系统剩余内存低于**watermark[low]**的时候，内核的**kswapd**开始起作用，进行内存回收。直到剩余内存达到**watermark[high]**的时候停止。如果内存消耗导致剩余内存达到了或超过了**watermark[min]**时，就会触发直接回收（**direct reclaim**）。

明白了水位标记的概念之后，zonefile + zonefree <= high_wmark_pages(zone)这个公式就能理解了。这里的zonefile相当于内存中文件映射的总量，zonefree相当于剩余内存的总量。内核一般认为，如果zonefile还有的话，就可以尽量通过清空文件缓存获得部分内存，而不必只使用swap方式对anon的内存进行交换。整个判断的概念是说，在全局回收的状态下（有global_reclaim(sc)标记），如果当前的文件映射内存总量+剩余内存总量的值评估小于等于watermark[high]标记的时候，就可以进行直接swap了。这样是为了防止进入cache陷阱，具体描述可以见代码注释。这个判断对系统的影响是，**swappiness**设置为**0**时，有剩余内存的情况下也可能发生交换。

那么watermark相关值是如何计算的呢？所有的内存watermark标记都是根据当前内存总大小和一个可调参数进行运算得来的，这个参数是：/proc/sys/vm/min_free_kbytes。首先这个参数本身决定了系统中每个zone的watermark[min]的值大小，然后内核根据min的大小并参考每个zone的内存大小分别算出每个zone的low水位和high水位值。
想了解具体逻辑可以参见源代码目录下的：mm/page_alloc.c文件。在系统中可以从/proc/zoneinfo文件中查看当前系统的相关的信息和使用情况。

> 我们会发现以上内存管理的相关逻辑都是以zone为单位的，这里zone的含义是指内存的分区管理。Linux将内存分成多个区，主要有直接访问区(DMA)、一般区(Normal)和高端内存区(HighMemory)。内核对内存不同区域的访问因为硬件结构因素会有寻址和效率上的差别。如果在NUMA架构上，不同CPU所管理的内存也是不同的zone。

相关参数设置

zone_reclaim_mode：

zone_reclaim_mode模式是在2.6版本后期开始加入内核的一种模式，可以用来管理当一个内存区域(zone)内部的内存耗尽时，是从其内部进行内存回收还是可以从其他zone进行回收的选项，我们可以通过/proc/sys/vm/zone_reclaim_mode文件对这个参数进行调整。

在申请内存时(内核的get_page_from_freelist()方法中)，内核在当前zone内没有足够内存可用的情况下，会根据zone_reclaim_mode的设置来决策是从下一个zone找空闲内存还是在zone内部进行回收。这个值为0时表示可以从下一个zone找可用内存，非0表示在本地回收。这个文件可以设置的值及其含义如下：

1. echo 0 > /proc/sys/vm/zone_reclaim_mode：意味着关闭zone_reclaim模式，可以从其他zone或NUMA节点回收内存。
2. echo 1 > /proc/sys/vm/zone_reclaim_mode：表示打开zone_reclaim模式，这样内存回收只会发生在本地节点内。

3. echo 2 > /proc/sys/vm/zone_reclaim_mode：在本地回收内存时，可以将cache中的脏数据写回硬盘，以回收内存。
4. echo 4 > /proc/sys/vm/zone_reclaim_mode：可以用swap方式回收内存。

不同的参数配置会在NUMA环境中对其他内存节点的内存使用产生不同的影响，大家可以根据自己的情况进行设置以优化你的应用。默认情况下，zone_reclaim模式是关闭的。这在很多应用场景下可以提高效率，比如文件服务器，或者依赖内存中cache比较多的应用场景。这样的场景对内存cache速度的依赖要高于进程进程本身对内存速度的依赖，所以我们宁可让内存从其他zone申请使用，也不愿意清本地cache。

如果确定应用场景是内存需求大于缓存，而且尽量要避免内存访问跨越NUMA节点造成的性能下降的话，则可以打开zone_reclaim模式。此时页分配器会优先回收容易回收的可回收内存（主要是当前不用的page cache页），然后再回收其他内存。

打开本地回收模式的写回可能会引发其他内存节点上的大量的脏数据写回处理。如果一个内存zone已经满了，那么脏数据的写回也会导致进程处理速度收到影响，产生处理瓶颈。这会降低某个内存节点相关的进程的性能，因为进程不再能够使用其他节点上的内存。但是会增加节点之间的隔离性，其他节点的相关进程运行将不会因为另一个节点上的内存回收导致性能下降。

除非针对本地节点的内存限制策略或者cpuset配置有变化，对swap的限制会有效约束交换只发生在本地内存节点所管理的区域上。

min_unmapped_ratio：

这个参数只在NUMA架构的内核上生效。这个值表示NUMA上每个内存区域的pages总数的百分比。在zone_reclaim_mode模式下，只有当相关区域的内存使用达到这个百分比，才会发生区域内存回收。在zone_reclaim_mode设置为4的时候，内核会比较所有的file-backed和匿名映射页，包括swapcache占用的页以及tmpfs文件的总内存使用是否超过这个百分比。其他设置的情况下，只比较基于一般文件的未映射页，不考虑其他相关页。

page-cluster：

page-cluster是用来控制从swap空间换入数据的时候，一次连续读取的页数，这相当于对交换空间的预读。这里的连续是指在swap空间上的连续，而不是在内存地址上的连续。因为swap空间一般是在硬盘上，对硬盘设备的连续读取将减少磁头的寻址，提高读取效率。这个文件中设置的值是2的指数。就是说，如果设置为0，预读的swap页数是2的0次方，等于1页。如果设置为3，就是2的3次方，等于8页。同时，设置为0也意味着关闭预读功能。文件默认值为3。我们可以根据我们的系统负载状态来设置预读的页数大小。

swap的相关操纵命令

可以使用mkswap将一个分区或者文件创建成swap空间。swapon可以查看当前的swap空间和启用一个swap分区或者文件。swapoff可以关闭swap空间。我们使用一个文件的例子来演示一下整个操作过程：

制作swap文件：

```
[root@zorrozou-pc0 ~]# dd if=/dev/zero of=./swapfile bs=1M count=8G
dd: error writing './swapfile': No space left on device
14062+0 records in
14061+0 records out
14744477696 bytes (15 GB, 14 GiB) copied, 44.0824 s, 334 MB/s
[root@zorrozou-pc0 ~]# mkswap swapfile
mkswap: swapfile: insecure permissions 0644, 0600 suggested.
Setting up swapspace version 1, size = 13.7 GiB (14744473600 bytes)
no label, UUID=a0ac2a67-0f68-4189-939f-4801bec7e8e1
```

启用swap文件：

```
[root@zorrozou-pc0 ~]# swapon swapfile
swapon: /root/swapfile: insecure permissions 0644, 0600 suggested.
[root@zorrozou-pc0 ~]# swapon -s
Filename                Type        Size    Used    Priority
/dev/dm-4                                   partition    33554428    9116    -1
/root/swapfile                              file        14398900    0    -2
```

关闭swap空间：

```
[root@zorrozou-pc0 ~]# swapoff /root/swapfile
[root@zorrozou-pc0 ~]# swapon -s
Filename                Type        Size    Used    Priority
/dev/dm-4                                   partition    33554428    9116    -1
```

在使用多个swap分区或者文件的时候，还有一个优先级的概念（Priority）。在swapon的时候，我们可以使用-p参数指定相关swap空间的优先级，值越大优先级越高，可以指定的数字范围是－1到32767。内核在使用swap空间的时候总是先使用优先级高的空间，后使用优先级低的。当然如果把多个swap空间的优先级设置成一样的，那么两个swap空间将会以轮询方式并行进行使用。如果两个swap放在两个不同的硬盘上，相同的优先级可以起到类似RAID0的效果，增大swap的读写效率。另外，编程时使用mlock()也可以将指定的内存标记为不会换出，具体帮助可以参考man 2 mlock。

最后

关于swap的使用建议，针对不同负载状态的系统是不一样的。有时我们希望swap大一些，可以在内存不够用的时候不至于触发oom-killer导致某些关键进程被杀掉，比如数据库业务。也有时候我们希望不要swap，因为当大量进程爆发增长导致内存爆掉之后，会因为swap导致IO跑死，整个系统都卡住，无法登录，无法处理。这时候我们就希望不要swap，即使出现oom-killer也造成不了太大影响，但是不能允许服务器因为IO卡死像多米诺骨牌一样全部死机，而且无法登陆。跑cpu运算的无状态的apache就是类似这样的进程池架构的程序。

所以，swap到底怎么用?要还是不要？设置大还是小？相关参数应该如何配置？是要根据我们自己的生产环境的情况而定的。阅读完本文后希望大家可以明白一些swap的深层次知识。我简单总结一下：

1. 一个内存剩余还比较大的系统中，是否有可能使用swap？有可能，如果运行中的某个阶段出发了这个条件：zonefile + zonefree <= high_wmark_pages(zone)，就可能会swap。
2. swappiness设置为0就相当于关闭swap么？不是的，关闭swap要使用swapoff命令。swappiness只是在内存发生回收操作的时候用来平衡cache回收和swap交换的一个参数，调整为0意味着，尽量通过清缓存来回收内存。
3. swappiness设置为100代表系统会尽量少用剩余内存而多使用swap么？不是的，这个值设置为100表示内存发生回收时，从cache回收内存和swap交换的优先级一样。就是说，如果目前需求100M内存，那么较大机率会从cache中清除50M内存，再将匿名页换出50M，把回收到的内存给应用程序使用。但是这还要看cache中是否能有空间，以及swap是否可以交换50m。内核只是试图对它们平衡一些而已。
4. kswapd进程什么时候开始内存回收？kswapd根据内存水位标记决定是否开始回收内存，如果标记达到low就开始回收，回收到剩余内存达到high标记为止。
5. 如何查看当前系统的内存水位标记？cat /proc/zoneinfo。

如果对本文有相关问题，可以在我的微博、微信或者博客上联系我。

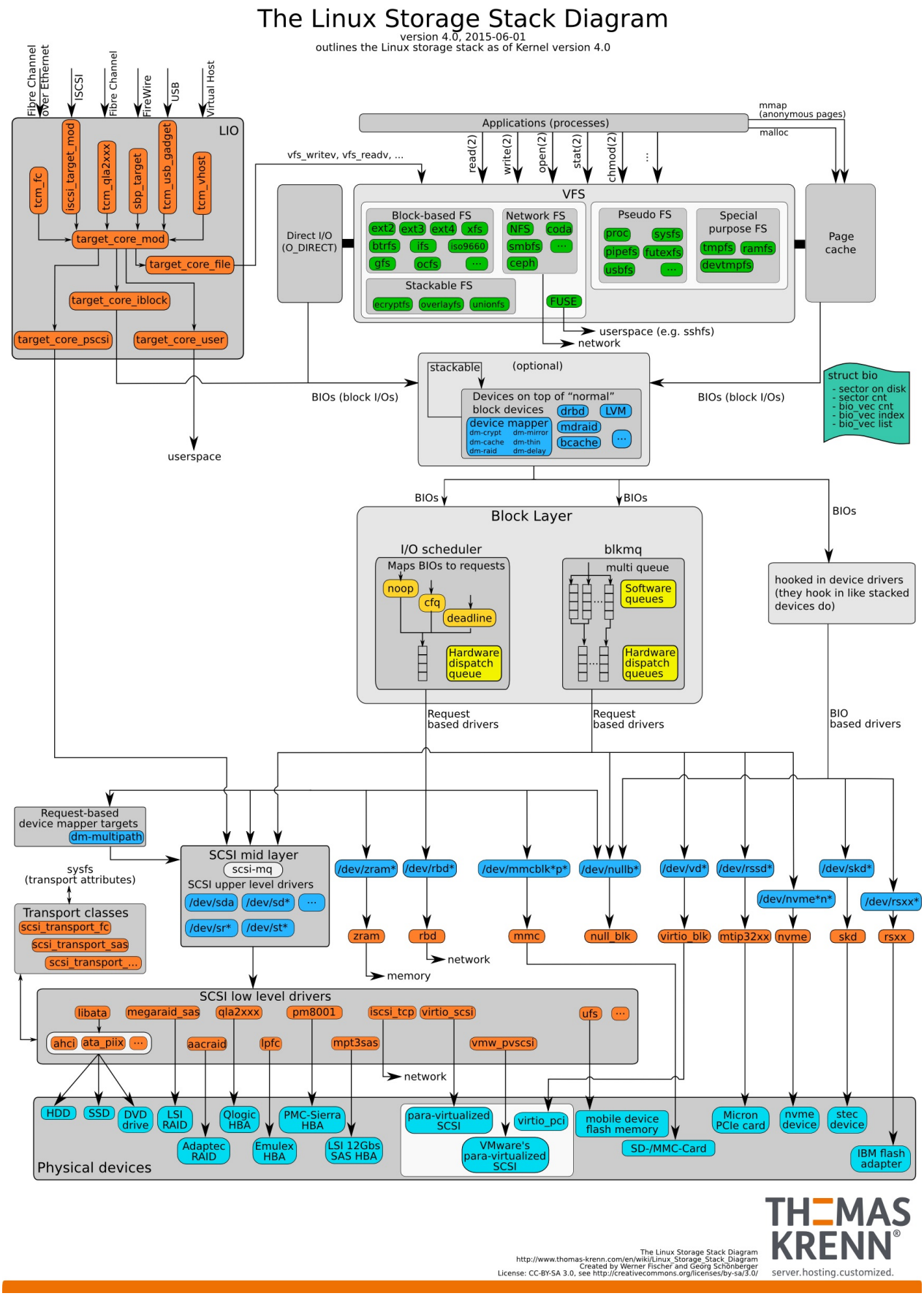

@orroz
weibo.com/30007147

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。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：


@orroz
weibo.com/30007147

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  %gni
ce   %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  %gni
ce   %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是内核可以实现眞对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_pres
sure     cpuset.mems                      cpuset.stat
cgroup.event_control   cpuset.cpuinfo        cpuset.mem_hardwall    cpuset.memory_spre
ad_page  cpuset.sched_load_balance        notify_on_release
cgroup.procs           cpuset.cpus           cpuset.memory_migrate  cpuset.memory_spre
ad_slab  cpuset.sched_relax_domain_level  tasks

/cgroup/cpuset/zorro:
cgroup.clone_children  cpuset.cpu_exclusive  cpuset.mem_exclusive   cpuset.memory_pres
sure     cpuset.mems                      cpuset.stat
cgroup.event_control   cpuset.cpuinfo        cpuset.mem_hardwall    cpuset.memory_spre
ad_page  cpuset.sched_load_balance        notify_on_release
cgroup.procs           cpuset.cpus           cpuset.memory_migrate  cpuset.memory_spre
ad_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  %gni
ce   %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 sse
4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm ab
m 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  %gni
ce   %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  %gni
ce   %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  %gni
ce   %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  %gni
ce   %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  %gni
ce   %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  %gni
ce   %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？

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命令确实给我们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，我们这次主要是眞对本文的重点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_leve
l       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_by
tes  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 {
    }
}
```

配置中添加了一个眞对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
```

让我们继续来看看眞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_lim
it_in_bytes
cgroup.event_control   memory.kmem.limit_in_bytes      memory.kmem.tcp.max_usage_in_by
tes  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.swappine
ss           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_hier
archy
```

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:" "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？

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

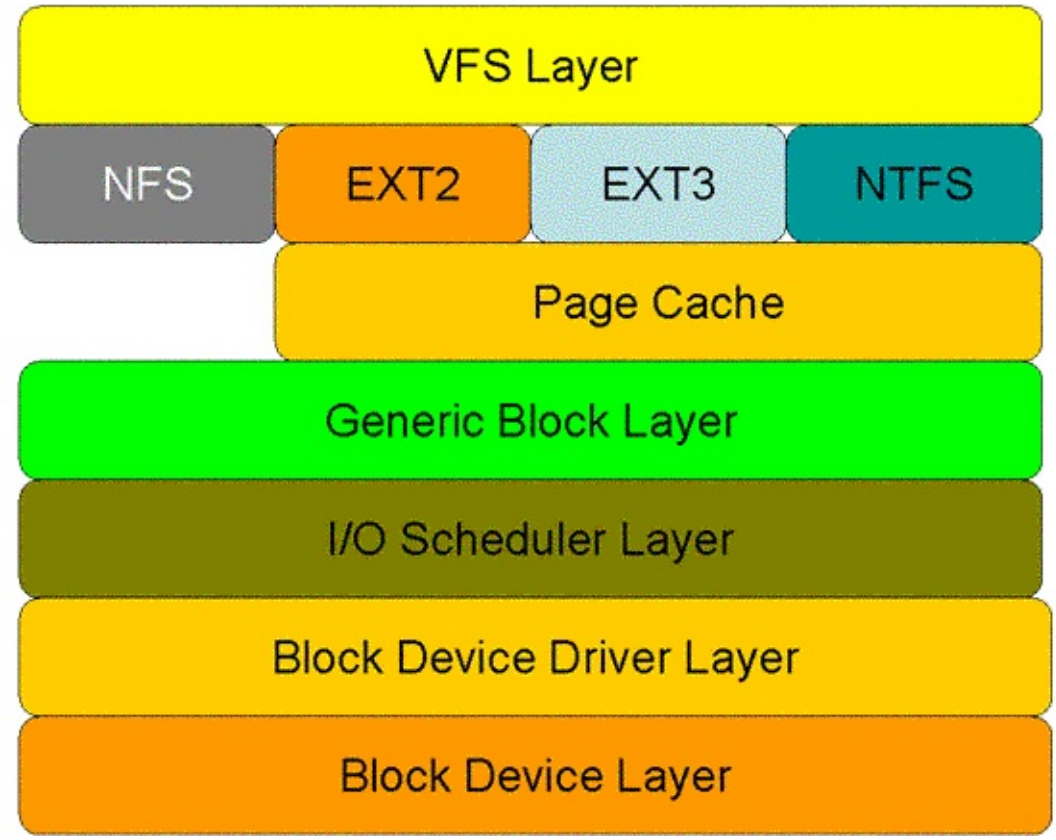
跟内存管理那部分复杂度类似，IO的资源隔离要讲清楚也是比较麻烦的。这部分内容都是这样，配置起来简单，但是要理解清楚确没那么简单。这次是跟Linux内核的IO实现有关系。对于IO的速度限制，实现思路跟CPU和内存都不一样。CPU是针对进程占用时间的比例限制，内存是空间限制，而当我们讨论IO资源隔离的时候，实际上有两个资源需要考虑，一个是空间，另一个是速度。对于空间来说，这个很简单，大不了分区就是了。现实手段中，分区、LVM、磁盘配额、目录配额等等，不同的分区管理方式，不同的文件系统都给出了很多不同的解决方案。所以，空间的限制实际上不是cgroup要解决的问题，那就是说，我们在这里要解决的问题是：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，定义了各种眞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，你会在哪里实现眞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         tas
ks
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_ch
ildren
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         tas
ks
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_ch
ildren
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         tas
ks
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_ch
ildren
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/t
est2 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/t
est1 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_d
evice
[root@zorrozou-pc0 zorro]# echo "8:16 200" > /sys/fs/cgroup/blkio/test2/blkio.weight_d
evice

[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/t
est1 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/t
est2 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.thr
ottle.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=d
irect 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 10
00.00    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  1
09.00    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  9
93.00    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 10
00.00    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.thr
ottle.read_bps_device
[root@zorrozou-pc0 zorro]# cgexec -g blkio:test1 dd if=/home/test of=/dev/null iflag=d
irect 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  8
25.00  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  8
25.00  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=d
irect 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  1
00.00    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  1
00.00    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

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，可以来关注我吻。

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

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 de
fault 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
```

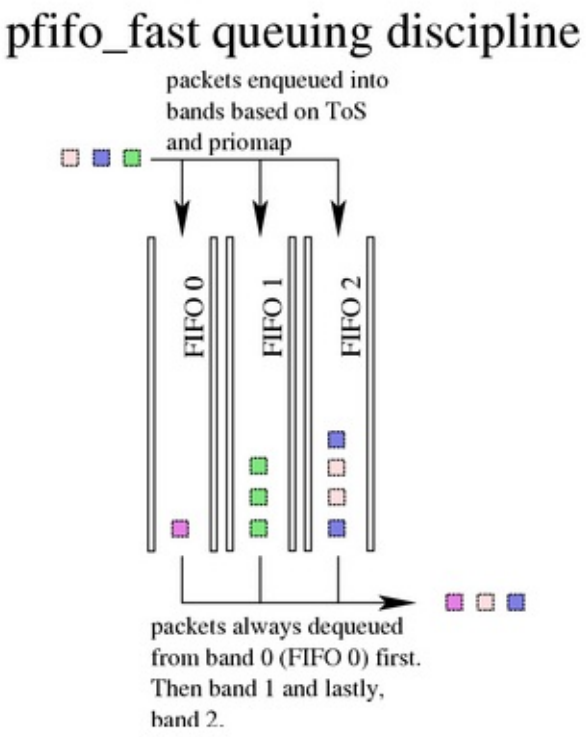
虽然看上去有些高深莫测，但是qdisc其实是个挺简单的概念，它就是它字面的意思：队列规则，或者叫做排队规则。我们都知道，网络数据都是被封装成一个一个的数据包进行传输的。如果网卡相当于数据包要出发的大门的话，那么qdisc无非就是规定了这些包在出发前如果需要排队的话该怎么排。我们先拿这个叫做pfifo_fast的队列规则来举例子描述一下吧，这个qdisc实现了一个以数据包（package）为单位的fifo队列，实际上可以认为是实现了三个队列（叫做bands），给每个队列定了一个优先级，以实现带优先级的排队规则。我们举个现实中的例子再来说明一下，大家都应该有去公交车站排队的经验吧？（神马？作为中国人你从来不排队？）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：

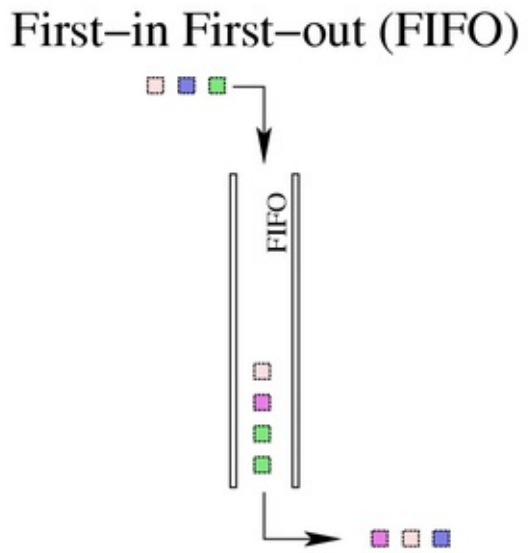


pfifo_fast一般情况下是内核对网卡默认选择的qdisc，它虽然提供了简单的优先级分类的支持，但是并没有提供可供修改的参数，就是说默认的优先级分类设置不能更改，也没有提供相关限速的功能。这个队列规则在一般情况下工作的都很稳定，但是最近Linux已经开始放弃使用这个qd作为默认的队列规则而改用一种叫做fq_codel的qdisc了。主要原因是，由于移动互联网的广泛应用，一种叫做Bufferbloat的现象影响越来越大了。

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 40
000 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 adapt
ive
```

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 int
erval 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 inte
rval 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：

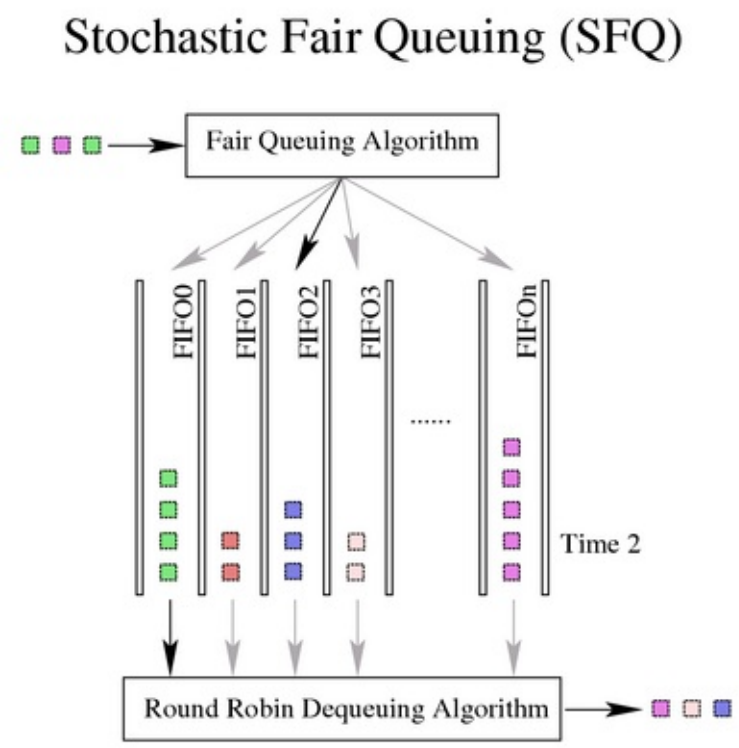
```
tc qdisc add dev eth0 parent 1:1 handle 10: sfq limit 3000 flows 512 divisor 16384 red
flowlimit 100000 min 8000 max 60000 probability 0.20 ecn headdrop
```

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

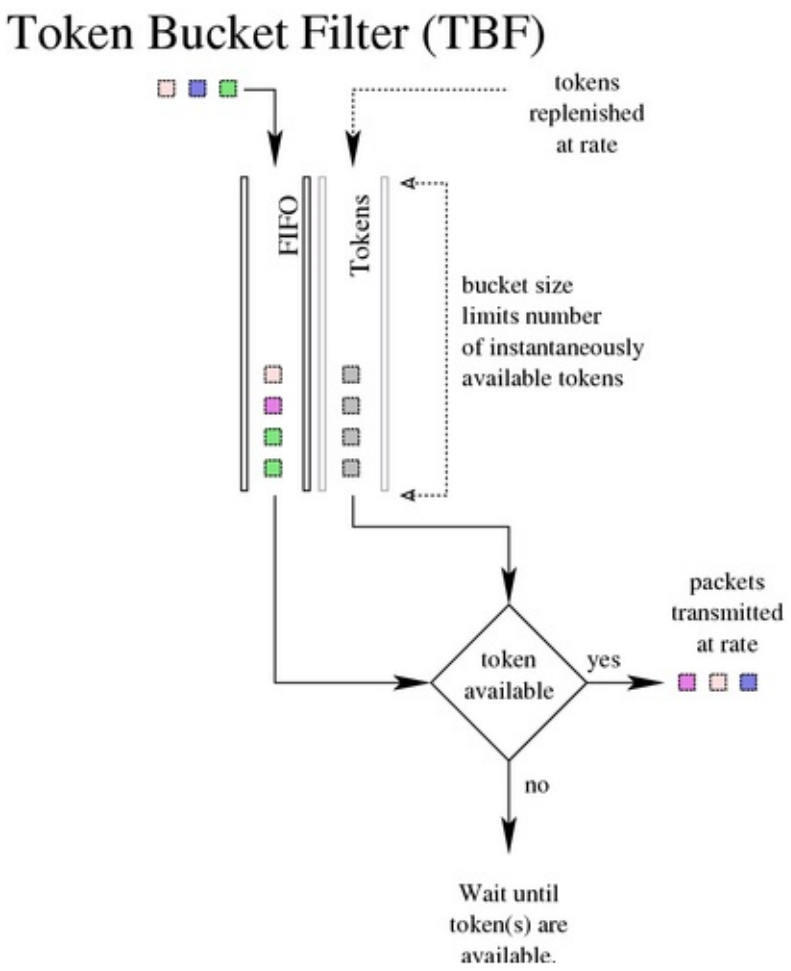
但是如果你的常规速率比较高，1Mbps 的峰值速率对我们就没有什么价值。要实现更高的峰值速率，可以在一个时钟周期内发送多个数据包。最有效的办法就是：再创建一个令牌桶！这第二个令牌桶缺省情况下为一个单个的数据包，并非一个眞正的桶。要计算峰值速率，用 mtu 乘以 100 就行了。 (应该说是乘以 HZ 数，Intel体系上是 100，Alpha 体系上是 1024)

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 matc
h ip dport 80 0xffff flowid 1:10
[root@zorrozou-pc0 zorro]# tc fi add dev enp2s0 protocol ip parent 1:0 prio 1 u32 matc
h 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 --se
t-mark 1
[root@zorrozou-pc0 zorro]# iptables -t mangle -A OUTPUT -p tcp --sport 25 -j MARK --se
t-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。

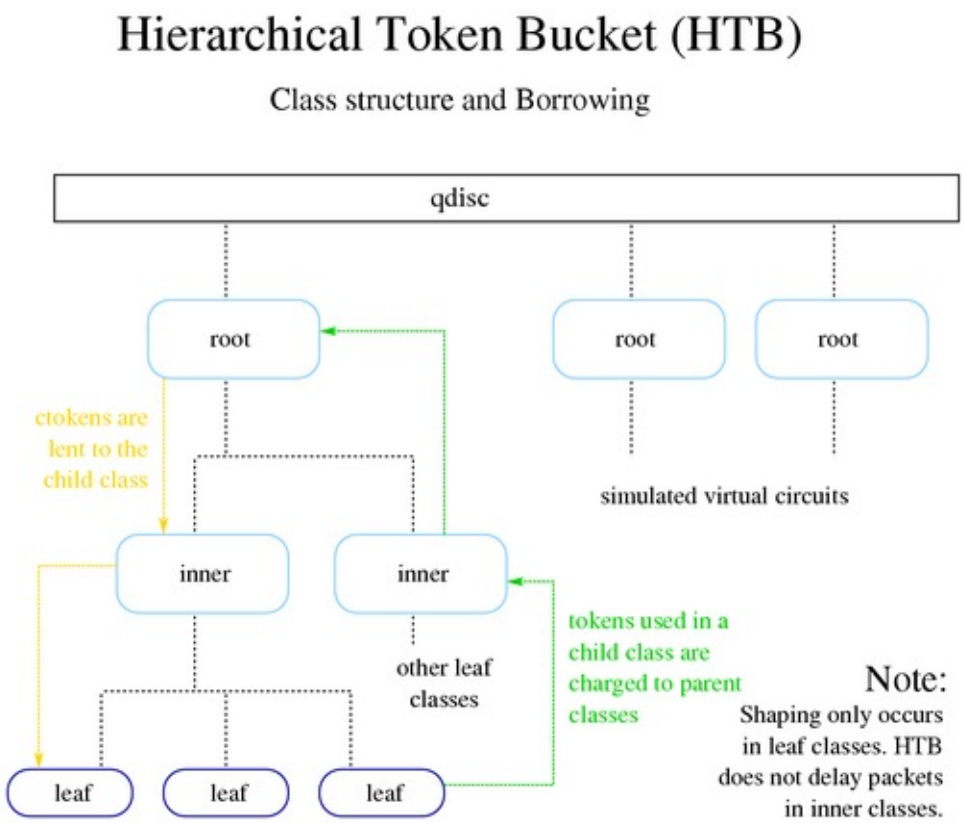
> 思考题：添加完iptables规则后，我们可以通过以下命令查看目前mangle表的内容：

```
[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 s
et 0x1
MARK       tcp  --  anywhere             anywhere             tcp spt:smtp MARK s
et 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 100
mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:10 htb rate 1
0mbit burst 20k
[root@localhost Desktop]# tc cl add dev eno16777736 parent 1:1 classid 1:20 htb rate 2
0mbit 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_c
ls.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.class
id
[root@localhost Desktop]# echo 0x00010020 > /sys/fs/cgroup/net_cls／zorro/net_cls.class
id
[root@localhost Desktop]# tc fi add dev eno16777736 parent 1: protocol ip prio 1 handl
e 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
```

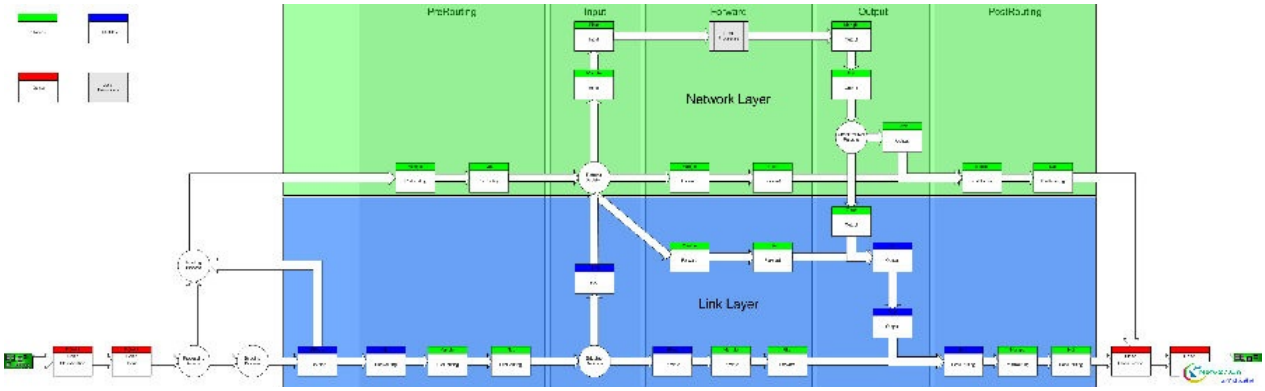
限速20mbps成功。如果想要修改对于一个分类的限速，使用如下命令即可：

```
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。



Linux的进程间通信 — 管道

前言

管道是UNIX环境中历史最悠久的进程间通信方式。本文主要说明在Linux环境上如何使用管道。阅读本文可以帮你解决以下问题：

1. 什么是管道和为什么要有管道？
2. 管道怎么分类？
3. 管道的实现是什么样的？
4. 管道有多大？
5. 管道的大小是不是可以调整？如何调整？

什么是管道？

管道，英文为pipe。这是一个我们在学习Linux命令行的时候就会引入的一个很重要的概念。它的发明人是道格拉斯.麦克罗伊，这位也是UNIX上早期shell的发明人。他在发明了shell之后，发现系统操作执行命令的时候，经常有需求要将一个程序的输出交给另一个程序进行处理，这种操作可以使用输入输出重定向加文件搞定，比如：

```
[zorro@zorro-pc pipe]$ ls  -l /etc/ > etc.txt
[zorro@zorro-pc pipe]$ wc -l etc.txt
183 etc.txt
```

但是这样未免显得太麻烦了。所以，管道的概念应运而生。目前在任何一个shell中，都可以使用“|”连接两个命令，shell会将前后两个进程的输入输出用一个管道相连，以便达到进程间通信的目的：

```
[zorro@zorro-pc pipe]$ ls -l /etc/ | wc -l
183
```

对比以上两种方法，我们也可以理解为，管道本质上就是一个文件，前面的进程以写方式打开文件，后面的进程以读方式打开。这样前面写完后面读，于是就实现了通信。实际上管道的设计也是遵循UNIX的“一切皆文件”设计原则的，它本质上就是一个文件。Linux系统直接把管道实现成了一种文件系统，借助VFS给应用程序提供操作接口。

虽然实现形态上是文件，但是管道本身并不占用磁盘或者其他外部存储的空间。在Linux的实现上，它占用的是内存空间。所以，Linux上的管道就是一个操作方式为文件的内存缓冲区。

管道的分类和使用

Linux上的管道分两种类型：

1. 匿名管道
2. 命名管道

这两种管道也叫做有名或无名管道。匿名管道最常见的形态就是我们在shell操作中最常用的"|"。它的特点是只能在父子进程中使用，父进程在产生子进程前必须打开一个管道文件，然后fork产生子进程，这样子进程通过拷贝父进程的进程地址空间获得同一个管道文件的描述符，以达到使用同一个管道通信的目的。此时除了父子进程外，没人知道这个管道文件的描述符，所以通过这个管道中的信息无法传递给其他进程。这保证了传输数据的安全性，当然也降低了管道了通用性，于是系统还提供了命名管道。

我们可以使用mkfifo或mknod命令来创建一个命名管道，这跟创建一个文件没有什么区别：

```
[zorro@zorro-pc pipe]$ mkfifo pipe
[zorro@zorro-pc pipe]$ ls -l pipe
prw-r--r-- 1 zorro zorro 0 Jul 14 10:44 pipe
```

可以看到创建出来的文件类型比较特殊，是p类型。表示这是一个管道文件。有了这个管道文件，系统中就有了对一个管道的全局名称，于是任何两个不相关的进程都可以通过这个管道文件进行通信了。比如我们现在让一个进程写这个管道文件：

```
[zorro@zorro-pc pipe]$ echo xxxxxxxxxxxxxx > pipe
```

此时这个写操作会阻塞，因为管道另一端没有人读。这是内核对管道文件定义的默认行为。此时如果有进程读这个管道，那么这个写操作的阻塞才会解除：

```
[zorro@zorro-pc pipe]$ cat pipe
xxxxxxxxxxxxxx
```

大家可以观察到，当我们cat完这个文件之后，另一端的echo命令也返回了。这就是命名管道。

Linux系统无论对于命名管道和匿名管道，底层都用的是同一种文件系统的操作行为，这种文件系统叫pipefs。大家可以在/etc/proc/filesystems文件中找到你的系统是不是支持这种文件系统：

```
[zorro@zorro-pc pipe]$ cat /proc/filesystems |grep pipefs
nodev    pipefs
```

观察完了如何在命令行中使用管道之后，我们再来看看如何在系统编程中使用管道。

PIPE

我们可以把匿名管道和命名管道分别叫做PIPE和FIFO。这主要因为在系统编程中，创建匿名管道的系统调用是pipe()，而创建命名管道的函数是mkfifo()。使用mknod()系统调用并指定文件类型为为S_IFIFO也可以创建一个FIFO。

使用pipe()系统调用可以创建一个匿名管道，这个系统调用的原型为：

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

这个方法将会创建出两个文件描述符，可以使用pipefd这个数组来引用这两个描述符进行文件操作。pipefd[0]是读方式打开，作为管道的读描述符。pipefd[1]是写方式打开，作为管道的写描述符。从管道写端写入的数据会被内核缓存直到有人从另一端读取为止。我们来看一下如何在一个进程中使用管道，虽然这个例子并没有什么意义：

```
[zorro@zorro-pc pipe]$ cat pipe.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define STRING "hello world!"

int main()
{
    int pipefd[2];
    char buf[BUFSIZ];

    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    if (write(pipefd[1], STRING, strlen(STRING)) < 0) {
        perror("write()");
        exit(1);
    }

    if (read(pipefd[0], buf, BUFSIZ) < 0) {
        perror("write()");
        exit(1);
    }

    printf("%s\n", buf);

    exit(0);
}
```

这个程序创建了一个管道，并且对管道写了一个字符串之后从管道读取，并打印在标准输出上。用一个图来说明这个程序的状态就是这样的：

一个进程自己给自己发送消息这当然不叫进程间通信，所以实际情况中我们不会在单个进程中使用管道。进程在pipe创建完管道之后，往往都要fork产生子进程，成为如下图表示的样子：

如图中描述，fork产生的子进程会继承父进程对应的文件描述符。利用这个特性，父进程先pipe创建管道之后，子进程也会得到同一个管道的读写文件描述符。从而实现了父子两个进程使用一个管道可以完成半双工通信。此时，父进程可以通过fd[1]给子进程发消息，子进程通过fd[0]读。子进程也可以通过fd[1]给父进程发消息，父进程用fd[0]读。程序实例如下：

```
[zorro@zorro-pc pipe]$ cat pipe_parent_child.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STRING "hello world!"

int main()
{
    int pipefd[2];
    pid_t pid;
    char buf[BUFSIZ];

    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
        /* this is child. */
        printf("Child pid is: %d\n", getpid());
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }

        printf("%s\n", buf);

        bzero(buf, BUFSIZ);
        snprintf(buf, BUFSIZ, "Message from child: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
        }

    } else {
        /* this is parent */
        printf("Parent pid is: %d\n", getpid());

        snprintf(buf, BUFSIZ, "Message from parent: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
```

```
        }

        sleep(1);

        bzero(buf, BUFSIZ);
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }

        printf("%s\n", buf);

        wait(NULL);
    }


    exit(0);
}
```

父进程先给子进程发一个消息，子进程接收到之后打印消息，之后再给父进程发消息，父进程再打印从子进程接收到的消息。程序执行效果：

```
[zorro@zorro-pc pipe]$ ./pipe_parent_child
Parent pid is: 8309
Child pid is: 8310
Message from parent: My pid is: 8309
Message from child: My pid is: 8310
```

从这个程序中我们可以看到，管道实际上可以实现一个半双工通信的机制。使用同一个管道的父子进程可以分时给对方发送消息。我们也可以看到对管道读写的一些特点，即：

在管道中没有数据的情况下，对管道的读操作会阻塞，直到管道内有数据为止。当一次写的数据量不超过管道容量的时候，对管道的写操作一般不会阻塞，直接将要写的数据写入管道缓冲区即可。

当然写操作也不会再所有情况下都不阻塞。这里我们要先来了解一下管道的内核实现。上文说过，管道实际上就是内核控制的一个内存缓冲区，既然是缓冲区，就有容量上限。我们把管道一次最多可以缓存的数据量大小叫做PIPESIZE。内核在处理管道数据的时候，底层也要调用类似read和write这样的方法进行数据拷贝，这种内核操作每次可以操作的数据量也是有限的，一般的操作长度为一个page，即默认为4k字节。我们把每次可以操作的数据量长度叫做PIPEBUF。POSIX标准中，对PIPEBUF有长度限制，要求其最小长度不得低于512字节。PIPEBUF的作用是，内核在处理管道的时候，如果每次读写操作的数据长度不大于PIPEBUF时，保证其操作是原子的。而PIPESIZE的影响是，大于其长度的写操作会被阻塞，直到当前管道中的数据被读取为止。

在Linux 2.6.11之前，PIPESIZE和PIPEBUF实际上是一样的。在这之后，Linux重新实现了一个管道缓存，并将它与写操作的PIPEBUF实现成了不同的概念，形成了一个默认长度为65536字节的PIPESIZE，而PIPEBUF只影响相关读写操作的原子性。从Linux 2.6.35之后，在fcntl系统调用方法中实现了F_GETPIPE_SZ和F_SETPIPE_SZ操作，来分别查看当前管道容量和设置管道容量。管道容量容量上限可以在/proc/sys/fs/pipe-max-size进行设置。

```
#define BUFSIZE 65536

......

ret = fcntl(pipefd[1], F_GETPIPE_SZ);
if (ret < 0) {
    perror("fcntl()");
    exit(1);
}

printf("PIPESIZE: %d\n", ret);

ret = fcntl(pipefd[1], F_SETPIPE_SZ, BUFSIZE);
if (ret < 0) {
    perror("fcntl()");
    exit(1);
}

......
```

PIPEBUF和PIPESIZE对管道操作的影响会因为管道描述符是否被设置为非阻塞方式而有行为变化，n为要写入的数据量时具体为：

O_NONBLOCK关闭，n <= PIPE_BUF：

n个字节的写入操作是原子操作，write系统调用可能会因为管道容量(PIPESIZE)没有足够的空间存放n字节长度而阻塞。

O_NONBLOCK打开，n <= PIPE_BUF：

如果有足够的空间存放n字节长度，write调用会立即返回成功，并且对数据进行写操作。空间不够则立即报错返回，并且errno被设置为EAGAIN。

O_NONBLOCK关闭，n > PIPE_BUF：

对n字节的写入操作不保证是原子的，就是说这次写入操作的数据可能会跟其他进程写这个管道的数据进行交叉。当管道容量长度低于要写的数据长度的时候write操作会被阻塞。

O_NONBLOCK打开，n > PIPE_BUF：

如果管道空间已满。write调用报错返回并且errno被设置为EAGAIN。如果没满，则可能会写入从1到n个字节长度，这取决于当前管道的剩余空间长度，并且这些数据可能跟别的进程的数据有交叉。

以上是在使用半双工管道的时候要注意的事情，因为在这种情况下，管道的两端都可能有多个进程进行读写处理。如果再加上线程，则事情可能变得更复杂。实际上，我们在使用管道的时候，并不推荐这样来用。管道推荐的使用方法是其单工模式：即只有两个进程通信，一个进程只写管道，另一个进程只读管道。实现为：

```
[zorro@zorro-pc pipe]$ cat pipe_parent_child2.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STRING "hello world!"

int main()
{
    int pipefd[2];
    pid_t pid;
    char buf[BUFSIZ];

    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
        /* this is child. */
        close(pipefd[1]);

        printf("Child pid is: %d\n", getpid());
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }

        printf("%s\n", buf);

    } else {
        /* this is parent */
        close(pipefd[0]);
```

```
        printf("Parent pid is: %d\n", getpid());

        snprintf(buf, BUFSIZ, "Message from parent: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
        }

        wait(NULL);
    }


    exit(0);
}
```

这个程序实际上比上一个要简单，父进程关闭管道的读端，只写管道。子进程关闭管道的写端，只读管道。整个管道的打开效果最后成为下图所示：

此时两个进程就只用管道实现了一个单工通信，并且这种状态下不用考虑多个进程同时对管道写产生的数据交叉的问题，这是最经典的管道打开方式，也是我们推荐的管道使用方式。另外，作为一个程序员，即使我们了解了Linux管道的实现，我们的代码也不能依赖其特性，所以处理管道时该越界判断还是要判断，该错误检查还是要检查，这样代码才能更健壮。

FIFO

命名管道在底层的实现跟匿名管道完全一致，区别只是命名管道会有一个全局可见的文件名以供别人open打开使用。再程序中创建一个命名管道文件的方法有两种，一种是使用mkfifo函数。另一种是使用mknod系统调用，例子如下：

```
[zorro@zorro-pc pipe]$ cat mymkfifo.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

    if (argc != 2) {
        fprintf(stderr, "Argument error!\n");
        exit(1);
    }

/*
    if (mkfifo(argv[1], 0600) < 0) {
        perror("mkfifo()");
        exit(1);
    }
*/
    if (mknod(argv[1], 0600|S_IFIFO, 0) < 0) {
        perror("mknod()");
        exit(1);
    }

    exit(0);
}
```

我们使用第一个参数作为创建的文件路径。创建完之后，其他进程就可以使用open()、read()、write()标准文件操作等方法进行使用了。其余所有的操作跟匿名管道使用类似。需要注意的是，无论命名还是匿名管道，它的文件描述都没有偏移量的概念，所以不能用lseek进行偏移量调整。

关于管道的其它议题，比如popen、pclose的使用等话题，《UNIX环境高级编程》中的相关章节已经讲的很清楚了。如果想学习补充这些知识，请参见此书。

最后

希望这些内容对大家进一步深入了解管道有帮助。如果有相关问题，可以在我的微博、微信或者博客上联系我。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：

Linux的进程间通信-文件和文件锁

前言

使用文件进行进程间通信应该是最先学会的一种IPC方式。任何编程语言中，文件IO都是很重要的知识，所以使用文件进行进程间通信就成了很自然被学会的一种手段。考虑到系统对文件本身存在缓存机制，使用文件进行IPC的效率在某些多读少写的情况下并不低下。但是大家似乎经常忘记IPC的机制可以包括“文件”这一选项。

我们首先引入文件进行IPC，试图先使用文件进行通信引入一个竞争条件的概念，然后使用文件锁解决这个问题，从而先从文件的角度来管中窥豹的看一下后续相关IPC机制的总体要解决的问题。阅读本文可以帮你解决以下问题：

1. 什么是竞争条件（racing）？。
2. flock和lockf有什么区别？
3. flockfile函数和flock与lockf有什么区别？
4. 如何使用命令查看文件锁？

竞争条件（racing）

我们的第一个例子是多个进程写文件的例子，虽然还没做到通信，但是这比较方便的说明一个通信时经常出现的情况：竞争条件。假设我们要并发100个进程，这些进程约定好一个文件，这个文件初始值内容写0，每一个进程都要打开这个文件读出当前的数字，加一之后将结果写回去。在理想状态下，这个文件最后写的数字应该是100，因为有100个进程打开、读数、加1、写回，自然是有多少个进程最后文件中的数字结果就应该是多少。但是实际上并非如此，可以看一下这个例子：

```
[zorro@zorrozou-pc0 process]$ cat racing.c
#include <unistd.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>

#define COUNT 100
#define NUM 64
#define FILEPATH "/tmp/count"

int do_child(const char *path)
{
    /* 这个函数是每个子进程要做的事情
    每个子进程都会按照这个步骤进行操作：
    1. 打开FILEPATH路径的文件
    2. 读出文件中的当前数字
    3. 将字符串转成整数
    4. 整数自增加1
    5. 将证书转成字符串
    6. lseek调整文件当前的偏移量到文件头
    7. 将字符串写会文件
    当多个进程同时执行这个过程的时候，就会出现racing：竞争条件，
    多个进程可能同时从文件独到同一个数字，并且分别对同一个数字加1并写回，
    导致多次写回的结果并不是我们最终想要的累积结果。 */
    int fd;
    int ret, count;
    char buf[NUM];
    fd = open(path, O_RDWR);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }
    /*    */
    ret = read(fd, buf, NUM);
    if (ret < 0) {
        perror("read()");
        exit(1);
    }
    buf[ret] = '\0';
    count = atoi(buf);
    ++count;
    sprintf(buf, "%d", count);
    lseek(fd, 0, SEEK_SET);
    ret = write(fd, buf, strlen(buf));
    /*    */
    close(fd);
    exit(0);
}

int main()
{
```

```
    pid_t pid;
    int count;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(FILEPATH);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }
}
```

这个程序做后执行的效果如下：

```
[zorro@zorrozou-pc0 process]$ make racing
cc     racing.c   -o racing
[zorro@zorrozou-pc0 process]$ echo 0 > /tmp/count
[zorro@zorrozou-pc0 process]$ ./racing
[zorro@zorrozou-pc0 process]$ cat /tmp/count
71[zorro@zorrozou-pc0 process]$
[zorro@zorrozou-pc0 process]$ echo 0 > /tmp/count
[zorro@zorrozou-pc0 process]$ ./racing
[zorro@zorrozou-pc0 process]$ cat /tmp/count
61[zorro@zorrozou-pc0 process]$
[zorro@zorrozou-pc0 process]$ echo 0 > /tmp/count
[zorro@zorrozou-pc0 process]$ ./racing
[zorro@zorrozou-pc0 process]$ cat /tmp/count
64[zorro@zorrozou-pc0 process]$
```

我们执行了三次这个程序，每次结果都不太一样，第一次是71，第二次是61，第三次是64，全都没有得到预期结果，这就是竞争条件**(racing)**引入的问题。仔细分析这个进程我们可以发现这个竞争条件是如何发生的：

最开始文件内容是0，假设此时同时打开了3个进程，那么他们分别读文件的时候，这个过程是可能并发的，于是每个进程读到的数组可能都是0，因为他们都在别的进程没写入1之前就开始读了文件。于是三个进程都是给0加1，然后写了个1回到文件。其他进程以此类推，每次100个进程的执行顺序可能不一样，于是结果是每次得到的值都可能不太一样，但是一定都少于产生的实际进程个数。于是我们把这种多个执行过程（如进程或线程）中访问同一个共享资源，而这些共享资源又有无法被多个执行过程存取的的程序片段，叫做临界区代码。

那么该如何解决这个racing的问题呢？对于这个例子来说，可以用文件锁的方式解决这个问题。就是说，对临界区代码进行加锁，来解决竞争条件的问题。哪段是临界区代码？在这个例子中，两端/ /之间的部分就是临界区代码。一个正确的例子是：

```
...
    ret = flock(fd, LOCK_EX);
    if (ret == -1) {
        perror("flock()");
        exit(1);
    }

    ret = read(fd, buf, NUM);
    if (ret < 0) {
        perror("read()");
        exit(1);
    }
    buf[ret] = '\0';
    count = atoi(buf);
    ++count;
    sprintf(buf, "%d", count);
    lseek(fd, 0, SEEK_SET);
    ret = write(fd, buf, strlen(buf));
    ret = flock(fd, LOCK_UN);
    if (ret == -1) {
        perror("flock()");
        exit(1);
    }
...
```

我们将临界区部分代码前后都使用了flock的互斥锁，防止了临界区的racing。这个例子虽然并没有真正达到让多个进程通过文件进行通信，解决某种协同工作问题的目的，但是足以表现出进程间通信机制的一些问题了。当涉及到数据在多个进程间进行共享的时候，仅仅只实现数据通信或共享机制本身是不够的，还需要实现相关的同步或异步机制来控制多个进程，达到保护临界区或其他让进程可以处理同步或异步事件的能力。我们可以认为文件锁是可以实现这样一种多进程的协调同步能力的机制，而除了文件锁以外，还有其他机制可以达到相同或者不同的功能，我们会在下文中继续详细解释。

再次，我们并不对flock这个方法本身进行功能性讲解。这种功能性讲解大家可以很轻易的在网上或者通过别的书籍得到相关内容。本文更加偏重的是Linux环境提供了多少种文件锁以及他们的区别是什么？

flock和lockf

从底层的实现来说，Linux的文件锁主要有两种：flock和lockf。需要额外对lockf说明的是，它只是fcntl系统调用的一个封装。从使用角度讲，lockf或fcntl实现了更细粒度文件锁，即：记录锁。我们可以使用lockf或fcntl对文件的部分字节上锁，而flock只能对整个文件加锁。这两种文件锁是从历史上不同的标准中起源的，flock来自BSD而lockf来自POSIX，所以lockf或fcntl实现的锁在类型上又叫做POSIX锁。

除了这个区别外，fcntl系统调用还可以支持强制锁（Mandatory locking）。强制锁的概念是传统UNIX为了强制应用程序遵守锁规则而引入的一个概念，与之对应的概念就是建议锁（Advisory locking）。我们日常使用的基本都是建议锁，它并不强制生效。这里的不强制生效的意思是，如果某一个进程对一个文件持有一把锁之后，其他进程仍然可以直接对文件进行各种操作的，比如open、read、write。只有当多个进程在操作文件前都去检查和对相关锁进行锁操作的时候，文件锁的规则才会生效。这就是一般建议锁的行为。而强制性锁试图实现一套内核级的锁操作。当有进程对某个文件上锁之后，其他进程即使不在操作文件之前检查锁，也会在open、read或write等文件操作时发生错误。内核将对有锁的文件在任何情况下的锁规则都生效，这就是强制锁的行为。由此可以理解，如果内核想要支持强制锁，将需要在内核实现open、read、write等系统调用内部进行支持。

从应用的角度来说，**Linux**内核虽然号称具备了强制锁的能力，但其对强制性锁的实现是不可靠的，建议大家还是不要在Linux下使用强制锁。事实上，在我目前手头正在使用的Linux环境上，一个系统在mount -o mand分区的时候报错(archlinux kernel 4.5)，而另一个系统虽然可以以强制锁方式mount上分区，但是功能实现却不完整，主要表现在只有在加锁后产生的子进程中open才会报错，如果直接write是没问题的，而且其他进程无论open还是read、write都没问题（Centos 7 kernel 3.10）。鉴于此，我们就不在此介绍如何在Linux环境中打开所谓的强制锁支持了。我们只需知道，在Linux环境下的应用程序，flock和lockf在是锁类型方面没有本质差别，他们都是建议锁，而非强制锁。

flock和lockf另外一个差别是它们实现锁的方式不同。这在应用的时候表现在flock的语义是针对文件的锁，而lockf是针对文件描述符（fd）的锁。我们用一个例子来观察这个区别：

```
[zorro@zorrozou-pc0 locktest]$ cat flock.c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <wait.h>

#define PATH "/tmp/lock"

int main()
{
    int fd;
    pid_t pid;
```

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

    if (flock(fd, LOCK_EX) < 0) {
        perror("flock()");
        exit(1);
    }
    printf("%d: locked!\n", getpid());

    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
/*
        fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
        if (fd < 0) {
                perror("open()");
                exit(1);
        }
*/
        if (flock(fd, LOCK_EX) < 0) {
            perror("flock()");
            exit(1);
        }
        printf("%d: locked!\n", getpid());
        exit(0);
    }
    wait(NULL);
    unlink(PATH);
    exit(0);
}
```

上面代码是一个flock的例子，其作用也很简单：

1. 打开/tmp/lock文件。
2. 使用flock对其加互斥锁。
3. 打印“PID：locked！”表示加锁成功。
4. 打开一个子进程，在子进程中使用flock对同一个文件加互斥锁。
5. 子进程打印“PID：locked！”表示加锁成功。如果没加锁成功子进程会推出，不显示相关内容。
6. 父进程回收子进程并推出。

这个程序直接编译执行的结果是：

```
[zorro@zorrozou-pc0 locktest]$ ./flock
23279: locked!
23280: locked!
```

父子进程都加锁成功了。这个结果似乎并不符合我们对文件加锁的本意。按照我们对互斥锁的理解，子进程对父进程已经加锁过的文件应该加锁失败才对。我们可以稍微修改一下上面程序让它达到预期效果，将子进程代码段中的注释取消掉重新编译即可：

```
...
/*
        fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
        if (fd < 0) {
                perror("open()");
                exit(1);
        }
*/
...
```

将这段代码上下的/*/删除重新编译。之后执行的效果如下：

```
[zorro@zorrozou-pc0 locktest]$ make flock
cc     flock.c   -o flock
[zorro@zorrozou-pc0 locktest]$ ./flock
23437: locked!
```

此时子进程flock的时候会阻塞，让进程的执行一直停在这。这才是我们使用文件锁之后预期该有的效果。而相同的程序使用lockf却不会这样。这个原因在于flock和lockf的语义是不同的。使用lockf或fcntl的锁，在实现上关联到文件结构体，这样的实现导致锁不会在fork之后被子进程继承。而flock在实现上关联到的是文件描述符，这就意味着如果我们在进程中复制了一个文件描述符，那么使用flock对这个描述符加的锁也会在新复制出的描述符中继续引用。在进程fork的时候，新产生的子进程的描述符也是从父进程继承（复制）来的。在子进程刚开始执行的时候，父子进程的描述符关系实际上跟在一个进程中使用dup复制文件描述符的状态一样（参见《UNIX环境高级编程》8.3节的文件共享部分）。这就可能造成上述例子的情况，通过fork产生的多个进程，因为子进程的文件描述符是复制的父进程的文件描述符，所以导致父子进程同时持有对同一个文件的互斥锁，导致第一个例子中的子进程仍然可以加锁成功。这个文件共享的现象在子进程使用open重新打开文件之后就不再存在了，所以重新对同一文件open之后，子进程再使用flock进行加锁的时候会阻塞。另外要注意：除非文件描述符被标记了close-on-exec标记，flock锁和lockf锁都可以穿越exec，在当前进程变成另一个执行镜像之后仍然保留。

上面的例子中只演示了fork所产生的文件共享对flock互斥锁的影响，同样原因也会导致dup或dup2所产生的文件描述符对flock在一个进程内产生相同的影响。dup造成的锁问题一般只有在多线程情况下才会产生影响，所以应该避免在多线程场景下使用flock对文件加锁，而

lockf/fcntl则没有这个问题。

为了对比flock的行为，我们在此列出使用lockf的相同例子，来演示一下它们的不同：

```
[zorro@zorrozou-pc0 locktest]$ cat lockf.c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <wait.h>

#define PATH "/tmp/lock"

int main()
{
    int fd;
    pid_t pid;

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

    if (lockf(fd, F_LOCK, 0) < 0) {
        perror("lockf()");
        exit(1);
    }
    printf("%d: locked!\n", getpid());

    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
/*
        fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
        if (fd < 0) {
            perror("open()");
            exit(1);
        }
*/
        if (lockf(fd, F_LOCK, 0) < 0) {
            perror("lockf()");
            exit(1);
        }
        printf("%d: locked!\n", getpid());
```

```
        exit(0);
    }
    wait(NULL);
    unlink(PATH);
    exit(0);
}
```

编译执行的结果是：

```
[zorro@zorrozou-pc0 locktest]$ ./lockf
27262: locked!
```

在子进程不用open重新打开文件的情况下，进程执行仍然被阻塞在子进程lockf加锁的操作上。关于fcntl对文件实现记录锁的详细内容，大家可以参考《UNIX环境高级编程》中关于记录锁的14.3章节。

标准IO库文件锁

C语言的标准IO库中还提供了一套文件锁，它们的原型如下：

```
#include <stdio.h>

void flockfile(FILE *filehandle);
int ftrylockfile(FILE *filehandle);
void funlockfile(FILE *filehandle);
```

从实现角度来说，stdio库中实现的文件锁与flock或lockf有本质区别。作为一种标准库，其实现的锁必然要考虑跨平台的特性，所以其结构都是在用户态的FILE结构体中实现的，而非内核中的数据结构来实现。这直接导致的结果就是，标准IO的锁在多进程环境中使用是有问题的。进程在fork的时候会复制一整套父进程的地址空间，这将导致子进程中的FILE结构与父进程完全一致。就是说，父进程如果加锁了，子进程也将持有这把锁，父进程没加锁，子进程由于地址空间跟父进程是独立的，所以也无法通过FILE结构体检查别的进程的用户态空间是否家了标准IO库提供的文件锁。这种限制导致这套文件锁只能处理一个进程中的多个线程之间共享的FILE 的进行文件操作。就是说，多个线程必须同时操作一个用*fopen*打开的*FILE* 变量，如果内部自己使用fopen重新打开文件，那么返回的FILE *地址不同，也起不到线程的互斥作用。

我们分别将两种使用线程的状态的例子分别列出来，第一种是线程之间共享同一个FILE *的情况，这种情况互斥是没问题的：

```
[zorro@zorro-pc locktest]$ cat racing_pthread_sharefp.c
#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <pthread.h>

#define COUNT 100
#define NUM 64
#define FILEPATH "/tmp/count"
static FILE *filep;

void *do_child(void *p)
{
    int fd;
    int ret, count;
    char buf[NUM];

    flockfile(filep);

    if (fseek(filep, 0L, SEEK_SET) == -1) {
        perror("fseek()");
    }
    ret = fread(buf, NUM, 1, filep);

    count = atoi(buf);
    ++count;
    sprintf(buf, "%d", count);
    if (fseek(filep, 0L, SEEK_SET) == -1) {
        perror("fseek()");
    }
    ret = fwrite(buf, strlen(buf), 1, filep);

    funlockfile(filep);

    return NULL;
}

int main()
{
    pthread_t tid[COUNT];
    int count;

    filep = fopen(FILEPATH, "r+");
    if (filep == NULL) {
        perror("fopen()");
        exit(1);
    }

    for (count=0;count<COUNT;count++) {
        if (pthread_create(tid+count, NULL, do_child, NULL) != 0) {
            perror("pthread_create()");
```

```
            exit(1);
        }
    }

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

    fclose(filep);

    exit(0);
}
```

另一种情况是每个线程都fopen重新打开一个描述符，此时线程是不能互斥的：

```
[zorro@zorro-pc locktest]$ cat racing_pthread_threadfp.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <pthread.h>

#define COUNT 100
#define NUM 64
#define FILEPATH "/tmp/count"

void *do_child(void *p)
{
    int fd;
    int ret, count;
    char buf[NUM];
    FILE *filep;

    filep = fopen(FILEPATH, "r+");
    if (filep == NULL) {
        perror("fopen()");
        exit(1);
    }

    flockfile(filep);

    if (fseek(filep, 0L, SEEK_SET) == -1) {
        perror("fseek()");
    }
```

```
    ret = fread(buf, NUM, 1, filep);

    count = atoi(buf);
    ++count;
    sprintf(buf, "%d", count);
    if (fseek(filep, 0L, SEEK_SET) == -1) {
        perror("fseek()");
    }
    ret = fwrite(buf, strlen(buf), 1, filep);

    funlockfile(filep);

    fclose(filep);
    return NULL;
}

int main()
{
    pthread_t tid[COUNT];
    int count;


    for (count=0;count<COUNT;count++) {
        if (pthread_create(tid+count, NULL, do_child, NULL) != 0) {
            perror("pthread_create()");
            exit(1);
        }
    }

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


    exit(0);
}
```

以上程序大家可以自行编译执行看看效果。

文件锁相关命令

系统为我们提供了flock命令，可以方便我们在命令行和shell脚本中使用文件锁。需要注意的是，flock命令是使用flock系统调用实现的，所以在使用这个命令的时候请注意进程关系对文件锁的影响。flock命令的使用方法和在脚本编程中的使用可以参见我的另一篇文章《shell编程之常用技巧》中的**bash**并发编程和**flock**这部分内容，在此不在赘述。

我们还可以使用lslocks命令来查看当前系统中的文件锁使用情况。一个常见的现实如下：

```
[root@zorrozou-pc0 ~]# lslocks
COMMAND           PID   TYPE  SIZE MODE  M      START        END PATH
firefox         16280  POSIX    0B WRITE 0          0          0 /home/zorro/.mozilla/
firefox/bk2bfsto.default/.parentlock
dmeventd          344  POSIX    4B WRITE 0          0          0 /run/dmeventd.pid
gnome-shell       472  FLOCK    0B WRITE 0          0          0 /run/user/120/wayland
-0.lock
flock           27452  FLOCK    0B WRITE 0          0          0 /tmp/lock
lvmetad           248  POSIX    4B WRITE 0          0          0 /run/lvmetad.pid
```

这其中，TYPE主要表示锁类型，就是上文我们描述的flock和lockf。lockf和fcntl实现的锁事POSIX类型。M表示是否事强制锁，0表示不是。如果是记录锁的话，START和END表示锁住文件的记录位置，0表示目前锁住的是整个文件。MODE主要用来表示锁的权限，实际上这也说明了锁的共享属性。在系统底层，互斥锁表示为WRITE，而共享锁表示为READ，如果这段出现*则表示有其他进程正在等待这个锁。其余参数可以参考man lslocks。

最后

本文通过文件盒文件锁的例子，引出了竞争条件这样在进程间通信中需要解决的问题。并深入探讨了系统编程中常用的文件锁的实现和应用特点。希望大家对进程间通信和文件锁的使用有更深入的理解。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：


@orroz
weibo.com/30007147

Linux进程间通信-共享内存

前言

本文主要说明在Linux环境上如何使用共享内存。阅读本文可以帮你解决以下问题：

1. 什么是共享内存和为什么要有共享内存？
2. 如何使用mmap进行共享内存？
3. 如何使用XSI共享内存？
4. 如何使用POSIX共享内存？
5. 如何使用hugepage共享内存以及共享内存的相关限制如何配置？
6. 共享内存都是如何实现的？

使用文件或管道进行进程间通信会有很多局限性，比如效率问题以及数据处理使用文件描述符而不如内存地址访问方便，于是多个进程以共享内存的方式进行通信就成了很自然要实现的IPC方案。Linux系统在编程上为我们准备了多种手段的共享内存方案。包括：

1. mmap内存共享映射。
2. XSI共享内存。
3. POSIX共享内存。

下面我们就来分别介绍一下这三种内存共享方式。

mmap内存共享映射

mmap本来的是存储映射功能。它可以将一个文件映射到内存中，在程序里就可以直接使用内存地址对文件内容进行访问，这可以让程序对文件访问更方便。其相关调用API原型如下：

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

由于这个系统调用的特性可以用在很多场合，所以Linux系统用它实现了很多功能，并不仅局限于存储映射。在这主要介绍的就是用mmap进行多进程的内存共享功能。Linux产生子进程的系统调用是fork，根据fork的语义以及其实现，我们知道新产生的进程在内存地址空间上跟父进程是完全一致的。所以Linux的mmap实现了一种可以在父子进程之间共享内存地址的方式，其使用方法是：

1. 父进程将flags参数设置MAP_SHARED方式通过mmap申请一段内存。内存可以映射某个具体文件，也可以不映射具体文件（fd置为-1，flag设置为MAP_ANONYMOUS）。
2. 父进程调用fork产生子进程。之后在父子进程内都可以访问到mmap所返回的地址，就可以共享内存了。

我们写一个例子试一下，这次我们并发100个进程写共享内存来看看竞争条件racing的情况：

```
[zorro@zorrozou-pc0 sharemem]$ cat racing_mmap.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>

#define COUNT 100

int do_child(int *count)
{
    int interval;

    /* critical section */
    interval = *count;
    interval++;
    usleep(1);
    *count = interval;
    /* critical section */

    exit(0);
}

int main()
{
    pid_t pid;
```

```
    int count;
    int *shm_p;

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED|MAP_ANONYM
OUS, -1, 0);
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(shm_p);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
    munmap(shm_p, sizeof(int));
    exit(0);
}
```

这个例子中，我们在子进程中为了延长临界区（critical section）处理的时间，使用了一个中间变量进行数值交换，并且还使用了usleep加强了一下racing的效果，最后执行结果：

```
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 20
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 17
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 14
[zorro@zorrozou-pc0 sharemem]$ ./racing_mmap
shm_p: 15
```

这段共享内存的使用是有竞争条件存在的，从文件锁的例子我们知道，进程间通信绝不仅仅是通信这么简单，还需要处理类似这样的临界区代码。在这里，我们也可以使用文件锁进行处理，但是共享内存使用文件锁未免显得太不协调了。除了不方便以及效率低下以外，文件

锁还不能够进行更高级的进程控制。所以，我们在此需要引入更高级的进程同步控制原语来实现相关功能，这就是信号量（semaphore）的作用。我们会在后续章节中集中讲解信号量的使用，在此只需了解使用mmap共享内存的方法。

我们有必要了解一下mmap的内存占用情况，以便知道使用它的成本。我们申请一段比较大的共享内存进行使用，并察看内存占用情况。测试代码：

```
[zorro@zorrozou-pc0 sharemem]$ cat mmap_mem.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>

#define COUNT 100
#define MEMSIZE 1024*1024*1023*2

int main()
{
    pid_t pid;
    int count;
    void *shm_p;

    shm_p = mmap(NULL, MEMSIZE, PROT_WRITE|PROT_READ, MAP_SHARED|MAP_ANONYMOUS, -1, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    bzero(shm_p, MEMSIZE);

    sleep(3000);

    munmap(shm_p, MEMSIZE);
    exit(0);
}
```

我们申请一段大概2G的共享内存，并置0。然后在执行前后分别看内存用量，看什么部分有变化：

```
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache   available
Mem:             15           2           2           0          10          11
Swap:            31           0          31
[zorro@zorrozou-pc0 sharemem]$ ./mmap_mem &
[1] 32036
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache   available
Mem:             15           2           0           2          12           9
Swap:            31           0          31
```

可以看到，这部分内存的使用被记录到了shared和buff/cache中。当然这个结果在不同版本的Linux上可能是不一样的，比如在Centos 6的环境中mmap的共享内存只会记录到buff/cache中。除了占用空间的问题，还应该注意，mmap方式的共享内存只能在通过fork产生的父子进程间通信，因为除此之外的其它进程无法得到共享内存段的地址。

XSI共享内存

为了满足多个无关进程共享内存的需求，Linux提供了更具通用性的共享内存手段，XSI共享内存就是这样一种实现。XSI是X/Open组织对UNIX定义的一套接口标准(X/Open System Interface)。由于UNIX系统的历史悠久，在不同时间点的不同厂商和标准化组织定义过一些列标准，而目前比较通用的标准实际上是POSIX。我们还会经常遇到的标准还包括SUS（Single UNIX Specification）标准，它们大概的关系是，SUS是POSIX标准的超集，定义了部分额外附加的接口，这些接口扩展了基本的POSIX规范。相应的系统接口全集被称为XSI标准，除此之外XSI还定义了实现必须支持的POSIX的哪些可选部分才能认为是遵循XSI的。它们包括文件同步，存储映射文件，存储保护及线程接口。只有遵循XSI标准的实现才能称为UNIX操作系统。

XSI共享内存在Linux底层的实现实际上跟mmap没有什么本质不同，只是在使用方法上有所区别。其使用的相关方法为：

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

我们首先要来理解的是key这一个参数。想象一下我们现在需要解决的问题："在一个操作系统内，如何让两个不相关（没有父子关系）的进程可以共享一个内存段？"系统中是否有现成的解决方案呢？当然有，就是文件。我们知道，文件的设计就可以让无关的进程可以进行数据交换。文件采用路径和文件名作为系统全局的一个标识符，但是每个进程打开这个文件之后，在进程内部都有一个"文件描述符"去指向文件。此时进程通过fork打开的子进程可以继承父进程的文件描述符，但是无关进程依然可以通过系统全局的文件名用open系统调用再次打开同一个文件，以便进行进程间通信。

实际上对于XSI的共享内存，其key的作用就类似文件的文件名，shmget返回的int类型的shmid就类似文件描述符，注意只是"类似"，而并非是同样的实现。这意味着，我们在进程中不能用select、poll、epoll这样的方法去控制一个XSI共享内存，因为它并不是"文件描述符"。对于一个XSI的共享内存，其key是系统全局唯一的，这就方便其他进程使用同样的key，打开同样一段共享内存，以便进行进程间通信。而使用fork产生的子进程，则可以直接通过shmid访问到相关共享内存段。这就是key的本质：系统中对XSI共享内存的全局唯一表示符。

明白了这个本质之后，我们再来看看这个key应该如何产生。相关方法为：

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

一个key是通过ftok函数，使用一个pathname和一个proj_jd产生的。就是说，在一个可能会使用共享内存的项目组中，大家可以约定一个文件名和一个项目的proj_id，来在同一个系统中确定一段共享内存的key。ftok并不会去创建文件，所以必须指定一个存在并且进程可以访问的pathname路径。这里还要指出的一点是，ftok实际上并不是根据文件的文件路径和文件名（pathname）产生key的，在实现上，它使用的是指定文件的inode编号和文件所在设备的设备编号。所以，不要以为你是用了不同的文件名就一定会得到不同的key，因为不同的文件名是可以指向相同inode编号的文件的（硬连接）。也不要认为你是用了相同的文件名就一定可以得到相同的key，在一个系统上，同一个文件名会被删除重建的几率是很大的，这种行为很有可能导致文件的inode变化。所以一个ftok的执行会隐含stat系统调用也就不难理解了。

最后大家还应该明白，key作为全局唯一标识不仅仅体现在XSI的共享内存中，XSI标准的其他进程间通信机制（信号量数组和消息队列）也使用这一命名方式。这部分内容在《UNIX环境高级编程》一书中已经有了很详尽的讲解，本文不在赘述。我们还是只来看一下它使用的例子，我们用XSI的共享内存来替换刚才的mmap：

```
[zorro@zorrozou-pc0 sharemem]$ cat racing_xsi_shm.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
```

```
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#define COUNT 100
#define PATHNAME "/etc/passwd"

int do_child(int proj_id)
{
    int interval;
    int *shm_p, shm_id;
    key_t shm_key;
    /* 使用ftok产生shmkey */
    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }
    /* 在子进程中使用shmget取到已经在父进程中创建好的共享内存id，注意shmget的第三个参数的使用。 */
    shm_id = shmget(shm_key, sizeof(int), 0);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }

    /* 使用shmat将相关共享内存段映射到本进程的内存地址。 */

    shm_p = (int *)shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    /* critical section */
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    /* critical section */

    /* 使用shmdt解除本进程内对共享内存的地址映射，本操作不会删除共享内存。 */
    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    exit(0);
}

int main()
{
```

```
    pid_t pid;
    int count;
    int *shm_p;
    int shm_id, proj_id;
    key_t shm_key;

    proj_id = 1234;

    /* 使用约定好的文件路径和proj_id产生shm_key。 */
    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }

    /* 使用shm_key创建一个共享内存，如果系统中已经存在此共享内存则报错退出，创建出来的共享内存权限为
0600。 */
    shm_id = shmget(shm_key, sizeof(int), IPC_CREAT|IPC_EXCL|0600);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }

    /* 将创建好的共享内存映射进父进程的地址以便访问。 */
    shm_p = (int *)shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    /* 共享内存赋值为0。 */
    *shm_p = 0;

    /*  打开100个子进程并发读写共享内存。 */
    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(proj_id);
        }
    }

    /* 等待所有子进程执行完毕。 */
    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    /* 显示当前共享内存的值。 */
    printf("shm_p: %d\n", *shm_p);
```

```
    /* 解除共享内存地质映射。 */
    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    /* 删除共享内存。 */
    if (shmctl(shm_id, IPC_RMID, NULL) < 0) {
        perror("shmctl()");
        exit(1);
    }

    exit(0);
}
```

XSI共享内存跟mmap在实现上并没有本质区别。而之所以引入key和shmid的概念，也主要是为了在非父子关系的进程之间可以共享内存。根据上面的例子可以看到，使用shmget可以根据key创建共享内存，并返回一个shmid。它的第二个参数size用来指定共享内存段的长度，第三个参数指定创建的标志，可以支持的标志为：IPC_CREAT、IPC_EXCL。从Linux 2.6之后，还引入了支持大页的共享内存，标志为：SHM_HUGETLB、SHM_HUGE_2MB等参数。shmget除了可以创建一个新的共享内存以外，还可以访问一个已经存在的共享内存，此时可以将shmflg置为0，不加任何标识打开。

在某些情况下，我们也可以不用通过一个key来生成共享内存。此时可以在key的参数所在位置填：IPC_PRIVATE，这样内核会在保证不产生冲突的共享内存段id的情况下新建一段共享内存。当然，这样调用则必然意味着是新创建，而不是打开已有得共享内存，所以flag位一定是IPC_CREAT。此时产生的共享内存只有一个shmid，而没有key，所以可以通过fork的方式将id传给子进程。

当获得shmid之后，就可以使用shmat来进行地址映射。shmat之后，通过访问返回的当前进程的虚拟地址就可以访问到共享内存段了。当然，在使用之后要记得使用shmdt解除映射，否则对于长期运行的程序可能造成虚拟内存地址泄漏，导致没有可用地址可用。shmdt并不能删除共享内存段，而只是解除共享内存和进程虚拟地址的映射，只要shmid对应的共享内存还存在，就仍然可以继续使用shmat映射使用。想要删除一个共享内存需要使用shmctl的IPC_RMID指令处理。也可以在命令行中使用ipcrm删除指定的共享内存id或key。

> 共享内存由于其特性，与进程中的其他内存段在使用习惯上有些不同。一般进程对栈空间分配可以自动回收，而堆空间通过malloc申请，free回收。这些内存在回收之后就可以认为是不存在了。但是共享内存不同，用shmdt之后，实际上其占用的内存还在，并仍然可以使用shmat映射使用。如果不是用shmctl或ipcrm命令删除的话，那么它将一直保留直到系统被关闭。对于刚接触共享内存的程序员来说这可能需要适应一下。实际上共享内存的生存周期根文件更像：进程对文件描述符执行close并不能删除文件，而只是关闭了本进程对文件的操作接口，这就像shmdt的作用。而真正删除文件要用unlink，活着使用rm命令，这就像是共享内存的shmctl的IPC_RMID和ipcrm命令。当然，文件如果不删除，下次重启依旧还在，因为它放在硬盘上，而共享内存下次重启就没了，因为它毕竟还是内存。

在这里，请大家理解关于为什么要使用key，和相关共享内存id的概念。后续我们还将看到，除了共享内存外，XSI的信号量、消息队列也都是通过这种方式进行相关资源标识的。

除了可以删除一个共享内存以外，shmctl还可以查看、修改共享内存的相关属性。这些属性的细节大家可以man 2 shmctl查看细节帮助。在系统中还可以使用ipcs -m命令查看系统中所有共享内存的的信息，以及ipcrm指定删除共享内存。

这个例子最后执行如下：

```
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 27
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 22
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 21
[zorro@zorrozou-pc0 sharemem]$ ./racing_xsi_shm
shm_p: 20
```

到目前为止，我们仍然没解决racing的问题，所以得到的结果仍然是不确定的，我们会在讲解信号量的时候引入锁解决这个问题，当然也可以用文件锁。我们下面再修改刚才的mmap_mem.c程序，换做shm方式再来看看内存的使用情况，代码如下：

```
[zorro@zorrozou-pc0 sharemem]$ cat xsi_shm_mem.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
```

```
#define COUNT 100
#define MEMSIZE 1024*1024*1023*2

int main()
{
    pid_t pid;
    int count, shm_id;
    void *shm_p;

    shm_id = shmget(IPC_PRIVATE, MEMSIZE, 0600);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }


    shm_p = shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    bzero(shm_p, MEMSIZE);

    sleep(3000);

    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    if (shmctl(shm_id, IPC_RMID, NULL) < 0) {
        perror("shmctl()");
        exit(1);
    }

    exit(0);
}
```

我们在这段代码中使用了IPC_PRIVATE方式共享内存，这是与前面程序不一样的地方。执行结果为：

```
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache   available
Mem:             15           2           2           0          10          11
Swap:            31           0          31
[zorro@zorrozou-pc0 sharemem]$ ./xsi_shm_mem &
[1] 4539
[zorro@zorrozou-pc0 sharemem]$ free -g
              total        used        free      shared  buff/cache   available
Mem:             15           2           0           2          12           9
Swap:            31           0          31
```

跟mmap的共享内存一样，XSI的共享内存在free现实中也会占用shared和buff/cache的消耗。实际上，在内核底层实现上，两种内存共享都是使用的tmpfs方式实现的，所以它们实际上的内存使用都是一致的。

对于Linux系统来说，使用XSI共享内存的时候可以通过shmget系统调用的shmflg参数来申请大页内存（huge pages），当然这样做将使进程的平台移植性变差。相关的参数包括：

SHM_HUGETLB (since Linux 2.6)

SHM_HUGE_2MB, SHM_HUGE_1GB (since Linux 3.8)

使用大页内存的好处是提高内核对内存管理的处理效率，这主要是因为在相同内存大小的情况下，使用大页内存（2M一页）将比使用一般内存页（4k一页）的内存页管理的数量大大减少，从而减少了内存页表项的缓存压力和CPU cache缓存内存地质映射的压力，提高了寻址能力和内存管理效率。大页内存还有其他一些使用时需要注意的地方：

1. 大页内存不能交换（SWAP）.
2. 使用不当时可能造成更大的内存泄漏。

我们继续使用上面的程序修改改为使用大页内存来做一下测试，大页内存需要使用root权限，代码跟上面程序一样，只需要修改一下shmget的参数，如下：

```
[root@zorrozou-pc0 sharemem]# cat xsi_shm_mem_huge.c
......
    shm_id = shmget(IPC_PRIVATE, MEMSIZE, SHM_HUGETLB|0600);
......
```

其余代码都不变。我们申请的内存大约不到2G，所以需要在系统内先给我们预留2G以上的大页内存：

```
[root@zorrozou-pc0 sharemem]# echo 2048 > /proc/sys/vm/nr_hugepages
[root@zorrozou-pc0 sharemem]# cat /proc/meminfo |grep -i huge
AnonHugePages:    841728 kB
HugePages_Total:    2020
HugePages_Free:     2020
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

2048是页数，每页2M，所以这里预留了几乎4G的内存空间给大页。之后我么好还需要确保共享内存的限制不会使我们申请失败：

```
[root@zorrozou-pc0 sharemem]# echo 2147483648 > /proc/sys/kernel/shmmax
[root@zorrozou-pc0 sharemem]# echo 33554432 > /proc/sys/kernel/shmall
```

之后编译执行相关命令：

```
[root@zorrozou-pc0 sharemem]# echo 1 > /proc/sys/vm/drop_caches
[root@zorrozou-pc0 sharemem]# free -g
              total        used        free      shared  buff/cache   available
Mem:             15           6           6           0           2           7
Swap:            31           0          31
[root@zorrozou-pc0 sharemem]# ./xsi_shm_mem_huge &
[1] 5508
[root@zorrozou-pc0 sharemem]# free -g
              total        used        free      shared  buff/cache   available
Mem:             15           6           6           0           2           7
Swap:            31           0          31
[root@zorrozou-pc0 sharemem]# cat /proc/meminfo |grep -i huge
AnonHugePages:    841728 kB
HugePages_Total:    2020
HugePages_Free:      997
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

大家可以根据这个程序的输出看到，当前系统环境（archlinux kernel 4.5）再使用大页内存之后，free命令是看不见其内存统计的。同样的设置在Centos 7环境下也是相同的显示。这就是说，如果使用大页内存作为共享内存使用，将在free中看不到相关内存统计，这估计是free命令目前暂时没将大页内存统计进内存使用所导致，暂时大家只能通过/proc/meminfo文件中的相关统计看到大页内存的使用信息。

XSI共享内存的系统相关限制如下：

/proc/sys/kernel/shmall：限制系统用在共享内存上的内存总页数。注意是页数，单位为4k。

/proc/sys/kernel/shmmax：限制一个共享内存段的最大长度，字节为单位。

/proc/sys/kernel/shmmni：限制整个系统可创建的最大的共享内存段个数。

XSI共享内存是历史比较悠久，也比较经典的共享内存手段。它几乎代表了共享内存的默认定义，当我们说有共享内存的时候，一般意味着使用了XSI的共享内存。但是这种共享内存也存在一切缺点，最受病垢的地方莫过于他提供的key＋projid的命名方式不够UNIX，没有遵循一切皆文件的设计理念。当然这个设计理念在一般的应用场景下并不是什么必须要遵守的理念，但是如果共享内存可以用文件描述符的方式提供给程序访问，毫无疑问可以在Linux上跟select、poll、epoll这样的IO异步事件驱动机制配合使用，做到一些更高级的功能。于是，遵循一切皆文件理念的POSIX标准的进程间通信机制应运而生。

POSIX共享内存

POSIX共享内存实际上毫无新意，它本质上就是mmap对文件的共享方式映射，只不过映射的是tmpfs文件系统上的文件。

什么是tmpfs？Linux提供一种“临时”文件系统叫做tmpfs，它可以将内存的一部分空间拿来当做文件系统使用，使内存空间可以当做目录文件来用。现在绝大多数Linux系统都有一个叫做/dev/shm的tmpfs目录，就是这样一种存在。具体使用方法，大家可以参考我的另一篇文章《Linux内存中的Cache真的能被回收么？》。

Linux提供的POSIX共享内存，实际上就是在/dev/shm下创建一个文件，并将其mmap之后映射其内存地址即可。我们通过它给定的一套参数就能猜到它的主要函数shm_open无非就是open系统调用的一个封装。大家可以通过man shm_overview来查看相关操作的方法。使用代码如下：

```
[root@zorrozou-pc0 sharemem]# cat racing_posix_shm.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>

#define COUNT 100
#define SHMPATH "shm"

int do_child(char * shmpath)
{
    int interval, shmfd, ret;
    int *shm_p;
    /* 使用shm_open访问一个已经创建的POSIX共享内存 */
    shmfd = shm_open(shmpath, O_RDWR, 0600);
    if (shmfd < 0) {
```

```
        perror("shm_open()");
        exit(1);
    }

    /* 使用mmap将对应的tmpfs文件映射到本进程内存 */
    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }
    /* critical section */
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    /* critical section */
    munmap(shm_p, sizeof(int));
    close(shmfd);

    exit(0);
}

int main()
{
    pid_t pid;
    int count, shmfd, ret;
    int *shm_p;

    /* 创建一个POSIX共享内存 */
    shmfd = shm_open(SHMPATH, O_RDWR|O_CREAT|O_TRUNC, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    /* 使用ftruncate设置共享内存段大小 */
    ret = ftruncate(shmfd, sizeof(int));
    if (ret < 0) {
        perror("ftruncate()");
        exit(1);
    }

    /* 使用mmap将对应的tmpfs文件映射到本进程内存 */
    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;
```

```
    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(SHMPATH);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
    munmap(shm_p, sizeof(int));
    close(shmfd);
    //sleep(3000);
    shm_unlink(SHMPATH);
    exit(0);
}
```

编译执行这个程序需要指定一个额外rt的库，可以使用如下命令进行编译：

```
[root@zorrozou-pc0 sharemem]# gcc -o racing_posix_shm -lrt racing_posix_shm.c
```

对于这个程序，我们需要解释以下几点：

1. shm_open的SHMPATH参数是一个路径，这个路径默认放在系统的/dev/shm目录下。这是shm_open已经封装好的，保证了文件一定会使用tmpfs。
2. shm_open实际上就是open系统调用的封装。我们当然完全可以使用open的方式模拟这个方法。
3. 使用ftruncate方法来设置“共享内存”的大小。其实就是更改文件的长度。
4. 要以共享方式做mmap映射，并且指定文件描述符为shmfd。
5. shm_unlink实际上就是unlink系统调用的封装。如果不做unlink操作，那么文件会一直存在于/dev/shm目录下，以供其它进程使用。
6. 关闭共享内存描述符直接使用close。

以上就是POSIX共享内存。其本质上就是个tmpfs文件。那么从这个角度说，mmap匿名共享内存、XSI共享内存和POSIX共享内存在内核实现本质上其实都是tmpfs。如果我们去查看POSIX共享内存的free空间占用的话，结果将跟mmap和XSI共享内存一样占用shared和buff/cache，所以我们就不再做这个测试了。这部分内容大家也可以参考《Linux内存中的Cache真的能被回收么？》。

根据以上例子，我们整理一下POSIX共享内存的使用相关方法：

```
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);
```

使用shm_open可以创建或者访问一个已经创建的共享内存。上面说过，实际上POSIX共享内存就是在/dev/shm目录中的的一个tmpfs格式的文件，所以shm_open无非就是open系统调用的封装，所以起函数使用的参数几乎一样。其返回的也是一个标准的我呢间描述符。

shm_unlink也一样是unlink调用的封装，用来删除文件名和文件的映射关系。在这就能看出POSIX共享内存和XSI的区别了，一个是使用文件名作为全局标识，另一个是使用key。

映射共享内存地址使用mmap，解除映射使用munmap。使用ftruncate设置共享内存大小，实际上就是对tmpfs的文件进行指定长度的截断。使用fchmod、fchown、fstat等系统调用修改和查看相关共享内存的属性。close调用关闭共享内存的描述符。实际上，这都是标准的文件操作。

最后

希望这些内容对大家进一步深入了共享内存有帮助。如果有相关问题，可以在我的微博、微信或者博客上联系我。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：


@orroz
weibo.com/30007147

Linux的进程间通信-消息队列

前言

Linux系统给我们提供了一种可以发送格式化数据流的通信手段，这就是消息队列。使用消息队列无疑在某些场景的应用下可以大大减少工作量，相同的工作如果使用共享内存，除了需要自己手工构造一个可能不够高效的队列外，我们还要自己处理竞争条件和临界区代码。而内核给我们提供的消息队列，无疑大大方便了我们的工作。

Linux环境提供了XSI和POSIX两套消息队列，本文将帮助您掌握以下内容：

1. 如何使用XSI消息队列。
2. 如何使用POSIX消息队列。
3. 它们的底层实现分别是什么样子的？
4. 它们分别有什么特点？以及相关资源限制。

XSI消息队列

系统提供了四个方法来操作XSI消息队列，它们分别是：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

我们可以使用msgget去创建或访问一个消息队列，与其他XSI IPC一样，msgget使用一个key作为创建消息队列的标识。这个key可以通过ftok生成或者指定为IPC_PRIVATE。指定为IPC_PRIVATE时，此队列会新建出来，而且内核会保证新建的队列key不会与已经存在的队列冲突，所以此时后面的msgflag应指定为IPC_CREAT。当msgflag指定为IPC_CREAT时，msgget会去试图创建一个新的消息队列，除非指定key的消息队列已经存在。可以使用O_CREAT | O_EXCL在指定key已经存在的情况下报错，而不是访问这个消息队列。我们来看创建一个消息队列的例子：

```
[zorro@zorro-pc mqueue]$ cat msg_create.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>

#define FILEPATH "/etc/passwd"
#define PROJID 1234

int main()
{
    int msgid;
    key_t key;
    struct msqid_ds msg_buf;

    key = ftok(FILEPATH, PROJID);
    if (key == -1) {
        perror("ftok()");
        exit(1);
    }

    msgid = msgget(key, IPC_CREAT|IPC_EXCL|0600);
    if (msgid == -1) {
        perror("msgget()");
        exit(1);
    }

    if (msgctl(msgid, IPC_STAT, &msg_buf) == -1) {
        perror("msgctl()");
        exit(1);
    }

    printf("msgid: %d\n", msgid);
    printf("msg_perm.uid: %d\n", msg_buf.msg_perm.uid);
    printf("msg_perm.gid: %d\n", msg_buf.msg_perm.gid);
    printf("msg_stime: %d\n", msg_buf.msg_stime);
    printf("msg_rtime: %d\n", msg_buf.msg_rtime);
    printf("msg_qnum: %d\n", msg_buf.msg_qnum);
    printf("msg_qbytes: %d\n", msg_buf.msg_qbytes);
}
```

这个程序可以创建并查看一个消息队列的相关状态，执行结果：

```
[zorro@zorro-pc mqueue]$ ./msg_create
msgid: 0
msg_perm.uid: 1000
msg_perm.gid: 1000
msg_stime: 0
msg_rtime: 0
msg_qnum: 0
msg_qbytes: 16384
```

如果我们在次执行这个程序，就会报错，因为key没有变化，我们使用了IPC_CREAT|IPC_EXCL，所以相关队列已经存在了就会报错：

```
[zorro@zorro-pc mqueue]$ ./msg_create
msgget(): File exists
```

顺便看一下msgctl方法，我们可以用它来取一个消息队列的相关状态。更详细的信息可以man 2 msgctl查看。除了查看队列状态以外，还可以使用msgctl设置相关队列状态以及删除指定队列。另外我们还可以使用ipcs -q命令查看系统中XSI消息队列的相关状态。其他相关参数请参考man ipcs。

使用msgsnd和msgrcv向队列发送和从队列接收消息。我们先来看看如何访问一个已经存在的消息队列和向其发送消息：

```
[zorro@zorro-pc mqueue]$ cat msg_send.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FILEPATH "/etc/passwd"
#define PROJID 1234
#define MSG "hello world!"

struct msgbuf {
    long mtype;
    char mtext[BUFSIZ];
};


int main()
{
    int msgid;
    key_t key;
    struct msgbuf buf;

    key = ftok(FILEPATH, PROJID);
    if (key == -1) {
        perror("ftok()");
        exit(1);
    }

    msgid = msgget(key, 0);
    if (msgid == -1) {
        perror("msgget()");
        exit(1);
    }

    buf.mtype = 1;
    strncpy(buf.mtext, MSG, strlen(MSG));
    if (msgsnd(msgid, &buf, strlen(buf.mtext), 0) == -1) {
        perror("msgsnd()");
        exit(1);
    }
}
```

使用msgget访问一个已经存在的消息队列时，msgflag指定为0即可。使用msgsnd发送消息时主要需要注意的是它的第二个和第三个参数。第二个参数用来指定要发送的消息，它实际上应该是一个指向某个特殊结构的指针，这个结构可以定义如下：

```
struct msgbuf {
    long mtype;
    char mtext[BUFSIZ];
};
```

这个结构的mtype实际上是用来指定消息类型的，可以指定的数字必需是个正整数。我们可以把这个概念理解为XSI消息队列对消息优先级的实现方法，即：需要传送的消息体的第一个long长度是用来指定类型的参数，而非消息本身，后面的内容才是消息。在我们实现的消息中，这个结构题可以传送的最大消息长度为BUFSIZE的字节数。当然，如果你的消息并不是一个字符串，也可以将mtype后面的信息实现成各种需要的格式，比如想要发送一个人的名字和他的数学语文成绩的话，可以这样实现：

```
struct msgbuf {
    long mtype;
    char name[NAMESIZE];
    int math, chinese;
};
```

这实际上就是让使用者自己去设计一个通讯协议，然后发送端和接收端使用约定好的协议进行通讯。msgsnd的第三个参数应该是这个消息结构体除了mtype以外的真实消息的长度，而不是这个结构题的总长度，这点是要注意的。所以，如果你定义了一个很复杂的消息协议的话，建议的长度写法是这样：

```
sizeof(buf)-sizeof(long)
```

msgsnd的最后一个参数可以用来指定IPC_NOWAIT。在消息队列满的情况下，默认的发送行为会阻塞等待，如果加了这个参数，则不会阻塞，而是立即返回，并且errno设置为EAGAIN。然后我们来看接收消息和删除消息队列的例子：

```
[zorro@zorro-pc mqueue]$ cat msg_receive.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FILEPATH "/etc/passwd"
#define PROJID 1234

struct msgbuf {
    long mtype;
    char mtext[BUFSIZ];
};


int main()
{
    int msgid;
    key_t key;
    struct msgbuf buf;

    key = ftok(FILEPATH, PROJID);
    if (key == -1) {
        perror("ftok()");
        exit(1);
    }

    msgid = msgget(key, 0);
    if (msgid == -1) {
        perror("msgget()");
        exit(1);
    }

    if (msgrcv(msgid, &buf, BUFSIZ, 1, 0) == -1) {
        perror("msgrcv()");
        exit(1);
    }

    printf("mtype: %d\n", buf.mtype);
    printf("mtype: %s\n", buf.mtext);

    if (msgctl(msgid, IPC_RMID, NULL) == -1) {
        perror("msgctl()");
        exit(1);
    }

    exit(0);
}
```

msgrcv会将消息从指定队列中删除，并将其内容填到其第二个参数指定的buf地址所在的内存中。第三个参数指定承接消息的buf长度，如果消息内容长度大于指定的长度，那么这个函数的行为将取决于最后一个参数msgflag是否设置了MSG_NOERROR，如果这个标志被设定，那消息将被截短，消息剩余部分将会丢失。如果没设置这个标志，msgrcv会失败返回，并且errno被设定为E2BIG。

第四个参数用来指定从消息队列中要取的消息类型msgtyp，如果设置为0，则无论什么类型，取队列中的第一个消息。如果值大于0，则读取符合这个类型的第一个消息，当最后一个参数msgflag设置为MSG_EXCEPT的时候，是对消息类型取逻辑非。即，不等于这个消息类型的第一个消息会被读取。如果指定一个小于0的值，那么将读取消息类型比这个负数的绝对值小的类型的所有消息中的第一个。

最后一个参数msgflag还可以设置为：

IPC_NOWAIT：非阻塞方式读取。当队列为空的时候，msgrcv会阻塞等待。加这个标志后将直接返回，errno被设置为ENOMSG。

MSG_COPY：从Linux 3.8之后开始支持以消息位置的方式读取消息。如果标志为置为MSG_COPY则表示启用这个功能，此时msgtyp的含义将从类型变为位置偏移量，第一个消息的起始值为0。如果指定位置的消息不存在，则返回并设置errno为ENOMSG。并且MSG_COPY和MSG_EXCEPT不能同时设置。另外还要注意这个功能需要内核配置打开CONFIG_CHECKPOINT_RESTORE选项。这个选项默认应该是不开的。

使用msgctl删除消息队列的方法比较简单，不在复述。另外关于msgctl的其他使用，请大家参考msgctl的手册。这部分内容的另外一个权威参考资料就是《UNIX环境高级编程》。我们在这里补充一下Linux系统对XSI消息队列的限制相关参数介绍：

/proc/sys/kernel/msgmax：这个文件限制了系统中单个消息最大的字节数。

/proc/sys/kernel/msgmni：这个文件限制了系统中可创建的最大消息队列个数。

/proc/sys/kernel/msgmnb：这个文件用来限制单个消息队列中可以存放的最大消息字节数。

以上文件都可以使用echo或者sysctl命令进行修改。

POSIX消息队列

POSIX消息队列是独立于XSI消息队列的一套新的消息队列API，让进程可以用消息的方式进行数据交换。这套消息队列在Linux 2.6.6版本之后开始支持，还需要你的glibc版本必须高于2.3.4。它们使用如下方法进行操作和控制：

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

类似对文件的open，我们可以用mq_open来打开一个已经创建的消息队列或者创建一个消息队列。这个函数返回一个叫做mqd_t类型的返回值，其本质上还是一个文件描述符，只是在这这里被叫做消息队列描述符（message queue descriptor），在进程里使用这个描述符对消息队列进程操作。所有被创建出来的消息队列在系统中都有一个文件与之对应，这个文件名是通过name参数指定的，这里需要注意的是：name必须是一个以"/"开头的字符串，比如我想让消息队列的名字叫"message"，那么name应该给的是"/message"。消息队列创建完毕后，会在/dev/mqueue目录下产生一个以name命名的文件，我们还可以通过cat这个文件来看这个消息队列的一些状态信息。其它进程在消息队列已经存在的情况下就可以通过mp_open打开名为name的消息队列来访问它。

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_pr
io, const struct timespec *abs_timeout);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)
;

ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_
prio, const struct timespec *abs_timeout);
```

在一个消息队列创建完毕之后，我们可以使用mq_send来对消息队列发送消息，mq_receive来对消息队列接收消息。正常的发送消息一般不会阻塞，除非消息队列处在某种异常状态或者消息队列已满的时候，而消息队列在空的时候，如果使用mq_receive去试图接受消息的行为也会被阻塞，所以就有必要为两个方法提供一个带超时时间的版本。这里要注意的是msg_prio这个参数，是用来指定消息优先级的。每个消息都有一个优先级，取值范围是0到sysconf(_SC_MQ_PRIO_MAX) - 1的大小。在Linux上，这个值为32768。默认情况下，消息队列会先按照优先级进行排序，就是msg_prio这个值越大的越先出队列。同一个优先级的消息按照fifo原则处理。在mq_receive方法中的msg_prio是一个指向int的地址，它并不是用来指定取的消息是哪个优先级的，而是会将相关消息的优先级取出来放到相关变量中，以便用户自己处理优先级。

```
int mq_close(mqd_t mqdes);
```

我们可以使用mq_close来关闭一个消息队列，这里的关闭并非删除了相关文件，关闭之后消息队列在系统中依然存在，我们依然可以继续打开它使用。这跟文件的close和unlink的概念是类似的。

```
int mq_unlink(const char *name);
```

使用mq_unlink真正删除一个消息队列。另外，我们还可以使用mq_getattr和mq_setattr来查看和设置消息队列的属性，其函数原型为：

```
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
```

mq_attr结构体是这样的结构：

```
struct mq_attr {
    long mq_flags;       /* 只可以通过此参数将消息队列设置为是否非阻塞O_NONBLOCK */
    long mq_maxmsg;      /* 消息队列的消息数上限 */
    long mq_msgsize;     /* 消息最大长度 */
    long mq_curmsgs;     /* 消息队列的当前消息个数 */
};
```

消息队列描述符河文件描述符一样，当进程通过fork打开一个子进程后，子进程中将从父进程继承相关描述符。此时父子进程中的描述符引用的是同一个消息队列，并且它们的mq_flags参数也将共享。下面我们使用几个简单的例子来看看他们的操作方法：

创建并向消息队列发送消息：

```
[zorro@zorro-pc mqueue]$ cat send.c
#include <fcntl.h>
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

#define MQNAME "/mqtest"


int main(int argc, char *argv[])
{

    mqd_t mqd;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "Argument error!\n");
        exit(1);
    }

    mqd = mq_open(MQNAME, O_RDWR|O_CREAT, 0600, NULL);
    if (mqd == -1) {
        perror("mq_open()");
        exit(1);
    }

    ret = mq_send(mqd, argv[1], strlen(argv[1]), atoi(argv[2]));
    if (ret == -1) {
        perror("mq_send()");
        exit(1);
    }

    exit(0);
}
```

注意相关方法在编译的时候需要链接一些库，所以我们可以创建Makefile来解决这个问题：

```
[zorro@zorro-pc mqueue]$ cat Makefile
CFLAGS+=-lrt -lpthread
```

我们添加了rt和pthread库，为以后的例子最好准备。当然大家也可以直接使用gcc -lrt -lpthread来解决这个问题，然后我们对程序编译并测试：

```
[zorro@zorro-pc mqueue]$ rm send
[zorro@zorro-pc mqueue]$ make send
cc -lrt -lpthread    send.c   -o send
[zorro@zorro-pc mqueue]$ ./send zorro 1
[zorro@zorro-pc mqueue]$ ./send shrek 2
[zorro@zorro-pc mqueue]$ ./send jerry 3
[zorro@zorro-pc mqueue]$ ./send zzzzz 1
[zorro@zorro-pc mqueue]$ ./send ssssss 2
[zorro@zorro-pc mqueue]$ ./send jjjjj 3
```

我们以不同优先级给消息队列添加了几条消息。然后我们可以通过文件来查看相关消息队列的状态：

```
[zorro@zorro-pc mqueue]$ cat /dev/mqueue/mqtest
QSIZE:31         NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

然后我们来看如何接收消息：

```
[zorro@zorro-pc mqueue]$ cat recv.c
#include <fcntl.h>
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

#define MQNAME "/mqtest"


int main()
{

    mqd_t mqd;
    int ret;
    int val;
    char buf[BUFSIZ];

    mqd = mq_open(MQNAME, O_RDWR);
    if (mqd == -1) {
        perror("mq_open()");
        exit(1);
    }

    ret = mq_receive(mqd, buf, BUFSIZ, &val);
    if (ret == -1) {
        perror("mq_send()");
        exit(1);
    }

    ret = mq_close(mqd);
    if (ret == -1) {
        perror("mp_close()");
        exit(1);
    }

    printf("msq: %s, prio: %d\n", buf, val);

    exit(0);
}
```

直接编译执行：

```
[zorro@zorro-pc mqueue]$ ./recv
msq: jerry, prio: 3
[zorro@zorro-pc mqueue]$ ./recv
msq: jjjjj, prio: 3
[zorro@zorro-pc mqueue]$ ./recv
msq: shrek, prio: 2
[zorro@zorro-pc mqueue]$ ./recv
msq: ssssss, prio: 2
[zorro@zorro-pc mqueue]$ ./recv
msq: zorro, prio: 1
[zorro@zorro-pc mqueue]$ ./recv
msq: zzzzz, prio: 1
```

可以看到优先级对消息队列内部排序的影响。然后是删除这个消息队列：

```
[zorro@zorro-pc mqueue]$ cat rmmq.c
#include <fcntl.h>
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

#define MQNAME "/mqtest"


int main()
{

    int ret;

    ret = mq_unlink(MQNAME);
    if (ret == -1) {
        perror("mp_unlink()");
        exit(1);
    }

    exit(0);
}
```

大家在从消息队列接收消息的时候会发现，当消息队列为空的时候，mq_receive会阻塞，直到有人给队列发送了消息才能返回并继续执行。在很多应用场景下，这种同步处理的方式会给程序本身带来性能瓶颈。为此，POSI消息队列使用mq_notify为处理过程增加了一个异步通知机制。使用这个机制，我们就可以让队列在由空变成不空的时候触发一个异步事件，通知调用进程，以便让进程可以在队列为空的时候不用阻塞等待。这个方法的原型为：

```
int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

其中sevp用来想内核注册具体的通知行为，可以man 7 sigevent查看相关帮助。这里我们不展开讲解，详细内容将在信号相关内容中详细说明。简单来说，我们可以使用nq_notify方法注册3种行为：SIGEV_NONE，SIGEV_SIGNAL和SIGEV_THREAD。它们分别的含义如下：

SIGEV_NONE：一个“空”提醒。其实就是不提醒。

SIGEV_SIGNAL：当队列中有了消息后给调用进程发送一个信号。可以使用struct sigevent结构体中的sigev_signo指定信号编号，信号的si_code字段将设置为SI_MESGQ以标示这是消息队列的信号。还可以通过si_pid和si_uid来指定信号来自什么pid和什么uid。

SIGEV_THREAD：当队列中有了消息后触发产生一个线程。当设置为线程时，可以使用struct sigevent结构体中的sigev_notify_function指定具体触发什么线程，使用sigev_notify_attributes设置线程属性，使用sigev_value.sival_ptr传递一个任何东西的指针。

我们先来看使用信号的简单例子：

```
[zorro@zorro-pc mqueue]$ cat notify_sig.c
#include <pthread.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

static mqd_t mqdes;

void mq_notify_proc(int sig_num)
{
    /* mq_notify_proc()是信号处理函数，
    当队列从空变成非空时，会给本进程发送信号，
    触发本函数执行。 */

    struct mq_attr attr;
    void *buf;
    ssize_t size;
    int prio;
    struct sigevent sev;

    /* 我们约定使用SIGUSR1信号进行处理，
    在此判断发来的信号是不是SIGUSR1。 */
    if (sig_num != SIGUSR1) {
        return;
    }

    /* 取出当前队列的消息长度上限作为缓存空间大小。 */
    if (mq_getattr(mqdes, &attr) < 0) {
        perror("mq_getattr()");
```

```
        exit(1);
    }

    buf = malloc(attr.mq_msgsize);
    if (buf == NULL) {
        perror("malloc()");
        exit(1);
    }

    /* 从消息队列中接收消息。 */
    size = mq_receive(mqdes, buf, attr.mq_msgsize, &prio);
    if (size == -1) {
        perror("mq_receive()");
        exit(1);
    }

    /* 打印消息和其优先级。 */
    printf("msq: %s, prio: %d\n", buf, prio);

    free(buf);

    /* 重新注册mq_notify，以便下次可以出触发。 */
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIGUSR1;
    if (mq_notify(mqdes, &sev) == -1) {
        perror("mq_notify()");
        exit(1);
    }

    return;
}

int main(int argc, char *argv[])
{
    struct sigevent sev;

    if (argc != 2) {
        fprintf(stderr, "Argument error!\n");
        exit(1);
    }

    /* 注册信号处理函数。 */
    if (signal(SIGUSR1, mq_notify_proc) == SIG_ERR) {
        perror("signal()");
        exit(1);
    }

    /* 打开消息队列，注意此队列需要先创建。 */
    mqdes = mq_open(argv[1], O_RDONLY);
    if (mqdes == -1) {
        perror("mq_open()");
        exit(1);
    }
```

```
    /* 注册mq_notify。 */
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIGUSR1;
    if (mq_notify(mqdes, &sev) == -1) {
        perror("mq_notify()");
        exit(1);
    }

    /* 主进程每秒打印一行x，等着从消息队列发来异步信号触发收消息。 */
    while (1) {
        printf("x\n");
        sleep(1);
    }
}
```

我们编译这个程序并执行：

```
[zorro@zorro-pc mqueue]$ ./notify_sig /mqtest
x
x
...
```

会一直打印x，等着队列变为非空，我们此时在别的终端给队列发送一个消息：

```
[zorro@zorro-pc mqueue]$ ./send hello 1
```

进程接收到信号，并且现实消息相关内容：

```
...
x
x
msq: hello, prio: 1
x
...
```

再发一个试试：

```
[zorro@zorro-pc mqueue]$ ./send zorro 3
```

显示：

```
...
x
msq: zorro, prio: 3
x
...
```

在mq_notify的man手册中，有一个触发线程进行异步处理的例子，我们在此就不再额外写一遍了，在此引用并注释一下，以方便大家理解：

```
[zorro@zorro-pc mqueue]$ cat example.c
#include <pthread.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void                     /* Thread start function */
tfunc(union sigval sv)
{
    /* 此函数在队列变为非空的时候会被触发执行 */

    struct mq_attr attr;
    ssize_t nr;
    void *buf;

    /* 上一个程序时将mqdes实现成了全局变量，而本例子中使用sival_ptr指针传递此变量的值 */
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);

    /* Determine max. msg size; allocate buffer to receive msg */

    if (mq_getattr(mqdes, &attr) == -1)
        handle_error("mq_getattr");
    buf = malloc(attr.mq_msgsize);
    if (buf == NULL)
        handle_error("malloc");

    /* 打印队列中相关消息信息 */
    nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
    if (nr == -1)
        handle_error("mq_receive");

    printf("Read %zd bytes from MQ\n", nr);
    free(buf);

    /* 本程序取到消息之后直接退出，不会循环处理。 */
    exit(EXIT_SUCCESS);         /* Terminate the process */
}
```

```
int
main(int argc, char *argv[])
{
    mqd_t mqdes;
    struct sigevent sev;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <mq-name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    mqdes = mq_open(argv[1], O_RDONLY);
    if (mqdes == (mqd_t) -1)
        handle_error("mq_open");

    /* 在此指定当异步事件来的时候以线程方式处理，
    触发的线程是：tfunc
    线程属性设置为：NULL
    需要给线程传递消息队列描述符mqdes，以便线程接收消息 */

    sev.sigev_notify = SIGEV_THREAD;
    sev.sigev_notify_function = tfunc;
    sev.sigev_notify_attributes = NULL;
    sev.sigev_value.sival_ptr = &mqdes;   /* Arg. to thread func. */
    if (mq_notify(mqdes, &sev) == -1)
        handle_error("mq_notify");

    pause();    /* Process will be terminated by thread function */
}
```

大家可以自行编译执行此程序进行测试。请注意mq_notify的行为：

1. 一个消息队列智能通过mq_notify注册一个进程进行异步处理。
2. 异步通知只会在消息队列从空变成非空的时候产生，其它队列的变动不会触发异步通知。
3. 如果有其他进程使用mq_receive等待队列的消息时，消息到来不会触发已注册mq_notify的程序产生异步通知。队列的消息会递送给在使用mq_receive等待的进程。
4. 一次mq_notify注册只会触发一次异步事件，此后如果队列再次由空变为非空也不会触发异步通知。如果需要一直可以触发，请处理异步通知之后再次注册mq_notify。
5. 如果sevp指定为NULL，表示取消注册异步通知。

POSIX消息队列相对XSI消息队列的一大优势是，我们又一个类似文件描述符的mqd的描述符可以进行操作，所以很自然的我们就会联想到是否可以使用多路IO转接机制对消息队列进程处理？在Linux上，答案是肯定的，我们可以使用select、poll和epoll对队列描述符进行处理，我们在此仅使用epoll举个简单的例子：

```
[zorro@zorro-pc mqueue]$ cat recv_epoll.c
```

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/epoll.h>

#define MQNAME "/mqtest"
#define EPSIZE 10


int main()
{

    mqd_t mqd;
    int ret, epfd, val, count;
    char buf[BUFSIZ];
    struct mq_attr new, old;
    struct epoll_event ev, rev;

    mqd = mq_open(MQNAME, O_RDWR);
    if (mqd == -1) {
        perror("mq_open()");
        exit(1);
    }

    /* 因为有epoll帮我们等待描述符是否可读，所以对mqd的处理可以设置为非阻塞 */
    new.mq_flags = O_NONBLOCK;

    if (mq_setattr(mqd, &new, &old) == -1) {
        perror("mq_setattr()");
        exit(1);
    }

    epfd = epoll_create(EPSIZE);
    if (epfd < 0) {
        perror("epoll_create()");
        exit(1);
    }

    /* 关注描述符是否可读 */
    ev.events = EPOLLIN;
    ev.data.fd = mqd;

    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, mqd, &ev);
    if (ret < 0) {
        perror("epoll_ctl()");
        exit(1);
    }

    while (1) {
```

```
        ret = epoll_wait(epfd, &rev, EPSIZE, -1);
        if (ret < 0) {
            /* 如果被信号打断则继续epoll_wait */
            if (errno == EINTR) {
                continue;
            } else {
                perror("epoll_wait()");
                exit(1);
            }
        }

        /* 此处处理所有返回的描述符（虽然本例子中只有一个） */
        for (count=0;count<ret;count++) {
            ret = mq_receive(rev.data.fd, buf, BUFSIZ, &val);
            if (ret == -1) {
                if (errno == EAGAIN) {
                    break;
                }
                perror("mq_receive()");
                exit(1);
            }
            printf("msq: %s, prio: %d\n", buf, val);
        }

    }

    /* 恢复描述符的flag */
    if (mq_setattr(mqd, &old, NULL) == -1) {
        perror("mq_setattr()");
        exit(1);
    }

    ret = mq_close(mqd);
    if (ret == -1) {
        perror("mp_close()");
        exit(1);
    }
    exit(0);
}
```

这就是POSIX消息队列比XSI更有趣的地方，XSI的消息队列并未遵守“一切皆文件”的原则。当然，使用select和poll这里就不再举例了，有兴趣的可以自己实现一下作为练习。

以上例子中，我们也分别演示了如何使用mq_setattr和mq_getattr，此处我们应该知道，在所有可以显示的属性中，O_NONBLOCK是mq_setattr唯一可以更改的参数设置，其他参数对于这个方法都是只读的，不能修改。系统提供了其他手段可以对这些限制进行修改：

/proc/sys/fs/mqueue/msg_default：在mq_open的attr参数设置为NULL的时候，这个文件中的数字限定了mq_maxmsg的值，就是队列的消息个数限制。默认为10个，当消息数达到上限之后，再使用mq_send发送消息会阻塞。

/proc/sys/fs/mqueue/msg_max：可以通过mq_open的attr参数设定的mq_maxmsg的数字上限。这个值默认也是10。

/proc/sys/fs/mqueue/msgsize_default：在mq_open的attr参数设置为NULL的时候，这个文件中的数字限定了mq_msgsize的值，就是队列的字节数数限制。

/proc/sys/fs/mqueue/msgsize_max：可以通过mq_open的attr参数设定的mq_msgsize的数字上限。

/proc/sys/fs/mqueue/queues_max：系统可以创建的消息队列个数上限。

最后

希望这些内容对大家进一步深入了解Linux的消息队列有帮助。如果有相关问题，可以在我的微博、微信或者博客上联系我。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：


@orroz
weibo.com/30007147

Linux的进程间通信-信号量

前言

信号量又叫信号灯，也有人把它叫做信号集，本文遵循《UNIX环境高级编程》的叫法，仍称其为信号量。它的英文是semaphores，本意是“旗语”“信号”的意思。由于其叫法中包含“信号”这个关键字，所以容易跟另一个信号signal搞混。在这里首先强调一下，Linux系统中的semaphore信号量和signal信号是完全不同的两个概念。我们将在其它文章中详细讲解信号signal。本文可以帮你学会：

1. 什么是XSI信号量？
2. 什么是PV操作及其应用。
3. 什么是POSIX信号量？
4. 信号量的操作方法及其实现。

我们已经知道文件锁对于多进程共享文件的必要性了，对一个文件加锁，可以防止多进程访问文件时的“竞争条件”。信号量提供了类似能力，可以处理不同状态下多进程甚至多线程对共享资源的竞争。它所起到的作用就像十字路口的信号灯或航船时的旗语，用来协调多个执行过程对临界区的访问。但是从本质上讲，信号量实际上是实现了一套可以实现类似锁功能的原语，我们不仅可以用它实现锁，还可以实现其它行为，比如经典的PV操作。

Linux环境下主要实现的信号量有两种。根据标准的不同，它们跟共享内存类似，一套XSI的信号量，一套POSIX的信号量。下面我们分别使用它们实现一套类似文件锁的方法，来简单看看它们的使用。

XSI信号量

XSI信号量就是内核实现的一个计数器，可以对计数器做甲减操作，并且操作时遵守一些基本操作原则，即：对计数器做加操作立即返回，做减操作要检查计数器当前值是否够减？（减被减数之后是否小于0）如果够，则减操作不会被阻塞；如果不够，则阻塞等待到够减为止。在此先给出其相关操作方法的原型：

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

可以使用semget创建或者打开一个已经创建的信号量数组。根据XSI共享内存中的讲解，我们应该已经知道第一个参数key用来标识系统内的信号量。这里除了可以使用ftok产生以外，还可以使用IPC_PRIVATE创建一个没有key的信号量。如果指定的key已经存在，则意味着打开这个信号量，这时nsems参数指定为0，semflg参数也指定为0。nsems参数表示在创建信号量数组的时候，这个数组中的信号量个数是几个。我们可以通过多个信号量的数组实现更复杂的信号量功能。最后一个semflg参数用来指定标志位，主要有：IPC_CREAT，IPC_EXCL和权限mode。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);

int semtimedop(int semid, struct sembuf *sops, size_t nsops, const struct timespec *ti
meout);
```

使用semop调用来对信号量数组进行操作。nsops指定对数组中的几个元素进行操作，如数组中只有一个信号量就指定为1。操作的所有参数都定义在一个sembuf结构体里，其内容如下：

```
unsigned short sem_num;  /* semaphore number */
short          sem_op;   /* semaphore operation */
short          sem_flg;  /* operation flags */
```

sem_flg可以指定的参数包括IPC_NOWAIT和SEM_UNDO。当制定了SEM_UNDO，进程退出的时候会自动UNDO它对信号量的操作。对信号量的操作会作用在指定的第sem_num个信号量。一个信号量集合中的第1个信号量的编号从0开始。所以，对于只有一个信号量的信号集，这个sem_num应指定为0。sem_op用来指定对信号量的操作，可以有的操作有三种：

正值操作：对信号量计数器的值（semval）进行加操作。

0值操作：对计数器的值没有影响，而且要求对进程对信号量必须有读权限。实际上这个行为是一个“等待计数器为0”的操作：如果计数器的值为0，则操作可以立即返回。如果不是0并且sem_flg被设置为IPC_NOWAIT的情况下，0值操作也不会阻塞，而是会立即返回，并且errno

被设置为EAGAIN。如果不是0，且没设置IPC_NOWAIT时，操作会阻塞，直到计数器值变成0为止，此时相关信号量的semncnt值会加1，这个值用来记录有多少个进程（线程）在此信号量上等待。除了计数器变为0会导致阻塞停止以外，还有其他情况也会导致停止等待：信号量被删除，semop操作会失败，并且errno被置为EIDRM。进程被信号（signal）打断，errno会被置为EINTR，切semzcnt会被正常做减处理。

负值操作：对计数器做减操作，且进程对信号量必须有写权限。如果当前计数器的值大于或等于指定负值的绝对值，则semop可以立即返回，并且计数器的值会被置为减操作的结果。如果sem_op的绝对值大于计数器的值semval，则说明目前不够减，测试如果sem_flg设置了IPC_NOWAIT，semop操作依然会立即返回并且errno被置为EAGAIN。如果没设置IPC_NOWAIT，则会阻塞，直到以下几种情况发生为止：

1. semval的值大于或等于sem_op的绝对值，这时表示有足够的值做减法了。
2. 信号量被删除，semop返回EIDRM。
3. 进程（线程）被信号打断，semop返回EINTR。

这些行为基本与0值操作类似。semtimedop提供了一个带超时机制的结构，以便实现等待超时。观察semop的行为我们会发现，有必要在一个信号量创建之后对其默认的计数器semval进行赋值。所以，我们需要在semop之前，使用semctl进行赋值操作。

```
int semctl(int semid, int semnum, int cmd, ...);
```

这个调用是一个可变参实现，具体参数要根据cmd的不同而变化。在一般的使用中，我们主要要学会使用它改变semval的值和查看、修改sem的属性。相关的cmd为：SETVAL、IPC_RMID、IPC_STAT。

一个简单的修改semval的例子：

```
semctl(semid, 0, SETVAL, 1);
```

这个调用可以将指定的sem的semval值设置为1。更具体的参数解释大家可以参考man 2 semctl。

以上就是信号量定义的原语意义。如果用它实现类似互斥锁的操作，那么我们就可以初始化一个默认计数器值为1的的信号量，当有人进行加锁操作的时候对其减1，解锁操作对其加1。于是对于一个已经被减1的信号量计数器来说，再有人加锁会导致阻塞等待，直到加锁的人解锁后才能再被别人加锁。

我们结合例子来看一下它们的使用，我们用sem实现一套互斥锁，这套锁除了可以锁文件，也可以用来给共享内存加锁，我们可以用它来保护上面共享内存使用的时的临界区。我们使用xsi共享内存的代码案例为例子：

```
[zorro@zorro-pc sem]$ cat racing_xsi_shm.c
```

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/sem.h>

#define COUNT 100
#define PATHNAME "/etc/passwd"

static int lockid;

int mylock_init(void)
{
    int semid;

    semid = semget(IPC_PRIVATE, 1, IPC_CREAT|0600);
    if (semid < 0) {
        perror("semget()");
        return -1;
    }
    if (semctl(semid, 0, SETVAL, 1) < 0) {
        perror("semctl()");
        return -1;
    }
    return semid;
}

void mylock_destroy(int lockid)
{
    semctl(lockid, 0, IPC_RMID);
}

int mylock(int lockid)
{
    struct sembuf sbuf;

    sbuf.sem_num = 0;
    sbuf.sem_op = -1;
    sbuf.sem_flg = 0;

    while (semop(lockid, &sbuf, 1) < 0) {
        if (errno == EINTR) {

            continue;
        }
```

```
        perror("semop()");
        return -1;
    }

    return 0;
}

int myunlock(int lockid)
{
    struct sembuf sbuf;

    sbuf.sem_num = 0;
    sbuf.sem_op = 1;
    sbuf.sem_flg = 0;

    if (semop(lockid, &sbuf, 1) < 0) {
        perror("semop()");
        return -1;
    }

    return 0;
}

int do_child(int proj_id)
{
    int interval;
    int *shm_p, shm_id;
    key_t shm_key;

    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }

    shm_id = shmget(shm_key, sizeof(int), 0);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }

    shm_p = (int *)shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    /* critical section */
    if (mylock(lockid) == -1) {
        exit(1);
    }
    interval = *shm_p;
    interval++;
    usleep(1);
```

```
    *shm_p = interval;
    if (myunlock(lockid) == -1) {
        exit(1);
    }
    /* critical section */

    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    exit(0);
}

int main()
{
    pid_t pid;
    int count;
    int *shm_p;
    int shm_id, proj_id;
    key_t shm_key;

    lockid = mylock_init();
    if (lockid == -1) {
        exit(1);
    }

    proj_id = 1234;
    if ((shm_key = ftok(PATHNAME, proj_id)) == -1) {
        perror("ftok()");
        exit(1);
    }

    shm_id = shmget(shm_key, sizeof(int), IPC_CREAT|IPC_EXCL|0600);
    if (shm_id < 0) {
        perror("shmget()");
        exit(1);
    }

    shm_p = (int *)shmat(shm_id, NULL, 0);
    if ((void *)shm_p == (void *)-1) {
        perror("shmat()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }
```

```
        if (pid == 0) {
            do_child(proj_id);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);

    if (shmdt(shm_p) < 0) {
        perror("shmdt()");
        exit(1);
    }

    if (shmctl(shm_id, IPC_RMID, NULL) < 0) {
        perror("shmctl()");
        exit(1);
    }

    mylock_destroy(lockid);

    exit(0);
}
```

此时可以得到正确的执行结果：

```
[zorro@zorro-pc sem]$ ./racing_xsi_shm
shm_p: 100
```

大家可以自己思考一下，如何使用信号量来完善这个所有的锁的操作行为，并补充以下方法：

1. 实现trylock。
2. 实现共享锁。
3. 在共享锁的情况下，实现查看当前有多少人以共享方式加了同一把锁。

系统中对于XSI信号量的限制都放在一个文件中，路径为：/proc/sys/kernel/sem。文件中包涵4个限制值，它们分别的含义是：

SEMMSL：一个信号量集（semaphore set）中，最多可以有多少个信号量。这个限制实际上就是semget调用的第二个参数的个数上限。

SEMMNS：系统中在所有信号量集中最多可以有多少个信号量。

SEMOPM：可以使用semop系统调用指定的操作数限制。这个实际上是semop调用中，第二个参数的结构体中的sem_op的数字上限。

SEMMNI：系统中信号量的id标示数限制。就是信号量集的个数上限。

PV操作原语

PV操作是操作系统原理中的重点内容之一，而根据上述的互斥锁功能的描述来看，实际上我们的互斥锁就是一个典型的PV操作。加锁行为就是P操作，解锁就是V操作。PV操作是计算机操作系统需要提供的基本功能之一。最开始它用来在只有1个CPU的计算机系统上实现多任务操作系统的功能原语。试想，多任务操作系统意味着系统中同时可以执行多个进程，但是CPU只有一个，那就意味着某一个时刻实际上只能有一个进程占用CPU，而其它进程此时都要等着。基于这个考虑，1962年狄克斯特拉在THE系统中提出了PV操作原语的设计，来实现多进程占用CPU资源的控制原语。在理解了互斥锁之后，我们能够意识到，临界区代码段实际上跟多进程使用一个CPU的环境类似，它们都是对竞争条件下的有限资源。对待这样的资源，就有必要使用PV操作原语进行控制。

根据这个思路，我们再扩展来看一个应用。我们都知道现在的计算机基本都是多核甚至多CPU的场景，所以很多计算任务如果可以并发执行，那么无疑可以增加计算能力。假设我们使用多进程的方式进行并发运算，那么并发多少个进程合适呢？虽然说这个问题会根据不同的应用场景发生变化，但是如果假定是一个极度消耗CPU的运算的话，那么无疑有几个CPU就应该并发几个进程。此时并发个数如果过多，则会增加调度开销导致整体吞度量下降，而过少则无法利用多个CPU核心。PV操作正好是一种可以实现类似方法的一种编程原语。我们假定一个应用模型，这个应用要找到从10010001到10020000数字范围内的质数。如果采用并发的方式，我们可以考虑给每一个要判断的数字都用一个进程去计算，但是这样无疑会使进程个数远远大于一般计算机的CPU个数。于是我们就可以在产生进程的时候，使用PV操作原语来控制同时进行运算的进程个数。这套PV原语的实现其实跟上面的互斥锁区别不大，对于互斥锁，计数器的初值为1，而对于这个PV操作，无非就是计数器的初值设置为当前计算机的核心个数，具体代码实现如下：

```
[zorro@zorro-pc sem]$ cat sem_pv_prime.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#define START 10010001
#define END 10020000
#define NPROC 4

static int pvid;
```

```
int mysem_init(int n)
{
    int semid;

    semid = semget(IPC_PRIVATE, 1, IPC_CREAT|0600);
    if (semid < 0) {
        perror("semget()");
        return -1;
    }
    if (semctl(semid, 0, SETVAL, n) < 0) {
        perror("semctl()");
        return -1;
    }
    return semid;
}

void mysem_destroy(int pvid)
{
    semctl(pvid, 0, IPC_RMID);
}

int P(int pvid)
{
    struct sembuf sbuf;

    sbuf.sem_num = 0;
    sbuf.sem_op = -1;
    sbuf.sem_flg = 0;

    while (semop(pvid, &sbuf, 1) < 0) {
        if (errno == EINTR) {
            continue;
        }
        perror("semop(p)");
        return -1;
    }

    return 0;
}

int V(int pvid)
{
    struct sembuf sbuf;

    sbuf.sem_num = 0;
    sbuf.sem_op = 1;
    sbuf.sem_flg = 0;

    if (semop(pvid, &sbuf, 1) < 0) {
        perror("semop(v)");
        return -1;
    }
```

```
    return 0;
}

int prime_proc(int n)
{
    int i, j, flag;

    flag = 1;
    for (i=2;i<n/2;++i) {
        if (n%i == 0) {
            flag = 0;
            break;
        }
    }
    if (flag == 1) {
        printf("%d is a prime\n", n);
    }
    /* 子进程判断完当前数字退出之前进行V操作 */
    V(pvid);
    exit(0);
}

void sig_child(int sig_num)
{
    while (waitpid(-1, NULL, WNOHANG) > 0);
}

int main(void)
{
    pid_t pid;
    int i;

    /* 当子进程退出的时候使用信号处理进行回收，以防止产生很多僵尸进程 */

    if (signal(SIGCHLD, sig_child) == SIG_ERR) {
        perror("signal()");
        exit(1);
    }

    pvid = mysem_init(NPROC);

    /* 每个需要运算的数字都打开一个子进程进行判断 */
    for (i=START;i<END;i+=2) {
        /* 创建子进程的时候进行P操作。 */
        P(pvid);
        pid = fork();
        if (pid < 0) {
            /* 如果创建失败则应该V操作 */
            V(pvid);
            perror("fork()");
            exit(1);
        }
        if (pid == 0) {
```

```
            /* 创建子进程进行这个数字的判断 */
            prime_proc(i);
        }
    }
    /* 在此等待所有数都运算完，以防止运算到最后父进程先mysem_destroy，导致最后四个子进程进行V操作时报错 */
    while (1) {sleep(1);};
    mysem_destroy(pvid);
    exit(0);
}
```

整个进程组的执行逻辑可以描述为，父进程需要运算判断10010001到10020000数字范围内所有出现的质数，采用每算一个数打开一个子进程的方式。为控制同时进行运算的子进程个数不超过CPU个数，所以申请了一个值为CPU个数的信号量计数器，每创建一个子进程，就对计数器做P操作，子进程运算完推出对计数器做V操作。由于P操作在计数器是0的情况下会阻塞，直到有其他子进程退出时使用V操作使计数器加1，所以整个进程组不会产生大于CPU个数的子进程进行任务的运算。

这段代码使用了信号处理的方式回收子进程，以防产生过多的僵尸进程，这种编程方法比较多用在daemon中。使用这个方法引出的问题在于，如果父进程不在退出前等所有子进程回收完毕，那么父进程将在最后几个子进程执行完之前就将信号量删除了，导致最后几个子进程进行V操作的时候会报错。当然，我们可以采用更优雅的方式进程处理，但是那并不是本文要突出讲解的内容，大家可以自行对相关方法进行完善。一般的daemon进程正常情况下父进程不会主动退出，所以不会有类似问题。

POSIX信号量

POSIX提供了一套新的信号量原语，其原型定义如下：

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

使用sem_open来创建或访问一个已经创建的POSIX信号量。创建时，可以使用value参数对其直接赋值。

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

sem_wait会对指定信号量进行减操作，如果信号量原值大于0，则减操作立即返回。如果当前值为0，则sem_wait会阻塞，直到能减为止。

```
int sem_post(sem_t *sem);
```

sem_post用来对信号量做加操作。这会导致某个已经使用sem_wait等在这个信号量上的进程返回。

```
int sem_getvalue(sem_t *sem, int *sval);
```

sem_getvalue用来返回当前信号量的值到sval指向的内存地址中。如果当前有进程使用sem_wait等待此信号量，POSIX可以允许有两种返回，一种是返回0，另一种是返回一个负值，这个负值的绝对值就是等待进程的个数。Linux默认的实现是返回0。

```
int sem_unlink(const char *name);

int sem_close(sem_t *sem);
```

使用sem_close可以在进程内部关闭一个信号量，sem_unlink可以在系统中删除信号量。

POSIX信号量实现的更清晰简洁，相比之下，XSI信号量更加复杂，但是却更佳灵活，应用场景更加广泛。在XSI信号量中，对计数器的加和减操作都是通过semop方法和一个sembuff的结构体来实现的，但是在POSIX中则给出了更清晰的定义：使用sem_post函数可以增加信号量计数器的值，使用sem_wait可以减少计数器的值。如果计数器的值当前是0，则sem_wait操作会阻塞到值大于0。

POSIX信号量也提供了两种方式的实现，命名信号量和匿名信号量。这有点类似XSI方式使用ftok文件路径创建和IPC_PRIVATE方式创建的区别。但是表现形式不太一样：

命名信号量：

命名信号量实际上就是有一个文件名的信号量。跟POSIX共享内存类似，信号量也会在/dev/shm目录下创建一个文件，如果有这个文件名就是一个命名信号量。其它进程可以通过这个文件名来通过sem_open方法使用这个信号量。除了访问一个命名信号量以外，sem_open方法还可以创建一个信号量。创建之后，就可以使用sem_wait、sem_post等方法进行操作了。这里要注意的是，一个命名信号量在用sem_close关闭之后，还要使用sem_unlink删除其文件名，才算彻底被删除。

匿名信号量：

一个匿名信号量仅仅就是一段内存区，并没有一个文件名与之对应。匿名信号量使用sem_init进行初始化，使用sem_destroy()销毁。操作方法跟命名信号量一样。匿名内存的初始化方法跟sem_open不一样，sem_init要求对一段已有内存进行初始化，而不是在/dev/shm下产生一

个文件。这就要求：如果信号量是在一个进程中的多个线程中使用，那么它所在的内存区应该是这些线程应该都能访问到的全局变量或者malloc分配到的内存。如果是在多个进程间共享，那么这段内存应该本身是一段共享内存（使用mmap、shmget或shm_open申请的内存）。

POSIX共享内存所涉及到的其它方法应该也都比较简单，更详细的帮助参考相关的man手册即可，下面我们分别给出使用命名和匿名信号量的两个代码例子：

命名信号量使用：

```
[zorro@zorro-pc sem]$ cat racing_posix_shm.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>

#define COUNT 100
#define SHMPATH "/shm"
#define SEMPATH "/sem"

static sem_t *sem;

sem_t *mylock_init(void)
{
    sem_t * ret;
    ret = sem_open(SEMPATH, O_CREAT|O_EXCL, 0600, 1);
    if (ret == SEM_FAILED) {
        perror("sem_open()");
        return NULL;
    }
    return ret;
}

void mylock_destroy(sem_t *sem)
{
    sem_close(sem);
    sem_unlink(SEMPATH);
}

int mylock(sem_t *sem)
{
    while (sem_wait(sem) < 0) {
        if (errno == EINTR) {
            continue;
```

```
        }
        perror("sem_wait()");
        return -1;
    }

    return 0;
}

int myunlock(sem_t *sem)
{
    if (sem_post(sem) < 0) {
        perror("semop()");
        return -1;
    }
}

int do_child(char * shmpath)
{
    int interval, shmfd, ret;
    int *shm_p;

    shmfd = shm_open(shmpath, O_RDWR, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }
    /* critical section */
    mylock(sem);
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    myunlock(sem);
    /* critical section */
    munmap(shm_p, sizeof(int));
    close(shmfd);

    exit(0);
}

int main()
{
    pid_t pid;
    int count, shmfd, ret;
    int *shm_p;
```

```
    sem = mylock_init();
    if (sem == NULL) {
        fprintf(stderr, "mylock_init(): error!\n");
        exit(1);
    }

    shmfd = shm_open(SHMPATH, O_RDWR|O_CREAT|O_TRUNC, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    ret = ftruncate(shmfd, sizeof(int));
    if (ret < 0) {
        perror("ftruncate()");
        exit(1);
    }

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(SHMPATH);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
    munmap(shm_p, sizeof(int));
    close(shmfd);
    shm_unlink(SHMPATH);
    sleep(3000);
    mylock_destroy(sem);
    exit(0);
}
```

匿名信号量使用：

```
[zorro@zorro-pc sem]$ cat racing_posix_shm_unname.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/file.h>
#include <wait.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <semaphore.h>

#define COUNT 100
#define SHMPATH "/shm"

static sem_t *sem;

void mylock_init(void)
{
    sem_init(sem, 1, 1);
}

void mylock_destroy(sem_t *sem)
{
    sem_destroy(sem);
}

int mylock(sem_t *sem)
{
    while (sem_wait(sem) < 0) {
        if (errno == EINTR) {
            continue;
        }
        perror("sem_wait()");
        return -1;
    }

    return 0;
}

int myunlock(sem_t *sem)
{
    if (sem_post(sem) < 0) {
        perror("semop()");
        return -1;
    }
}

int do_child(char * shmpath)
```

```
{
    int interval, shmfd, ret;
    int *shm_p;

    shmfd = shm_open(shmpath, O_RDWR, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }
    /* critical section */
    mylock(sem);
    interval = *shm_p;
    interval++;
    usleep(1);
    *shm_p = interval;
    myunlock(sem);
    /* critical section */
    munmap(shm_p, sizeof(int));
    close(shmfd);

    exit(0);
}

int main()
{
    pid_t pid;
    int count, shmfd, ret;
    int *shm_p;

    sem = (sem_t *)mmap(NULL, sizeof(sem_t), PROT_WRITE|PROT_READ, MAP_SHARED|MAP_ANON
YMOUS, -1, 0);
    if ((void *)sem == MAP_FAILED) {
        perror("mmap()");
        exit(1);
    }

    mylock_init();

    shmfd = shm_open(SHMPATH, O_RDWR|O_CREAT|O_TRUNC, 0600);
    if (shmfd < 0) {
        perror("shm_open()");
        exit(1);
    }

    ret = ftruncate(shmfd, sizeof(int));
    if (ret < 0) {
```

```
        perror("ftruncate()");
        exit(1);
    }

    shm_p = (int *)mmap(NULL, sizeof(int), PROT_WRITE|PROT_READ, MAP_SHARED, shmfd, 0)
;
    if (MAP_FAILED == shm_p) {
        perror("mmap()");
        exit(1);
    }

    *shm_p = 0;

    for (count=0;count<COUNT;count++) {
        pid = fork();
        if (pid < 0) {
            perror("fork()");
            exit(1);
        }

        if (pid == 0) {
            do_child(SHMPATH);
        }
    }

    for (count=0;count<COUNT;count++) {
        wait(NULL);
    }

    printf("shm_p: %d\n", *shm_p);
    munmap(shm_p, sizeof(int));
    close(shmfd);
    shm_unlink(SHMPATH);
    sleep(3000);
    mylock_destroy(sem);
    exit(0);
}
```

以上程序没有仔细考究，只是简单列出了用法。另外要注意的是，这些程序在编译的时候需要加额外的编译参数-lrt和-lpthread。

最后

希望这些内容对大家进一步深入了解Linux的信号量。如果有相关问题，可以在我的微博、微信或者博客上联系我。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux**系统技术 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码：