深入理解iostat

2017-06-28 | 分类 linux | 标签 linux

前言

iostat算是比较重要的查看块设备运行状态的工具，相信大多数使用Linux的同学都用过这个工具，或者听说过这个工具。但是对于这个工具，引起的误解也是最多的，大多数人对这个工具处于朦朦胧胧的状态。现在我们由浅到深地介绍这个工具，它输出的含义什么，介绍它的能力边界，介绍关于这个工具的常见误解。

基本用法和输出的基本含义

iostat的用法比较简单，一般来说用法如下：

```
iostat -mtx 2
```

含义是说，每2秒钟采集一组数据：

```
-m     Display statistics in megabytes per second.

-t     Print the time for each report displayed. The timestamp format may depend on the value of the S_TIME_FORMAT
environment variable (see below).

-x     Display extended statistics.
```

输出的结果如下所示：

```
06/26/2017 09:46:44 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          1.96    0.00    0.96    0.00    0.00   97.07

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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    38.00    0.00   58.50     0.00     0.49    17.23     0.00    0.03    0.00    0.03   0.00   0.00
sdd               0.00     0.00    0.00    2.00     0.00     0.00     4.00     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              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-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
sdg               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
sdc               0.00     0.00    0.00  465.50     0.00     1.82     8.00    63.99  136.49    0.00  136.49   2.15 100.00
sdf               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

06/26/2017 09:46:46 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          4.49    0.00    1.30    0.00    0.00   94.21

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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   151.00    0.00  182.50     0.00    19.28   216.39     0.01    0.09    0.00    0.09   0.04   0.80
sdd               0.00     0.00    0.00    5.50     0.00     0.02     6.55     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              0.00     0.00    0.00    4.00     0.00     0.02     8.00     0.00    0.00    0.00    0.00   0.00   0.00
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
sdg               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
sdc               0.00     0.00    0.00  462.00     0.00     1.80     8.00    63.98  138.39    0.00  138.39   2.16 100.00
sdf               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
```

注意，上图是在对sdc这块单盘（RAID卡上的单盘）做4KB的随机写入测试：

```
fio --name=randwrite --rw=randwrite --bs=4k --size=20G --runtime=1200 --ioengine=libaio --iodepth=64 --numjobs=1 --
rate_iops=5000 --filename=/dev/sdf --direct=1 --group_reporting
```

因此上图中只有sdc在忙。

如何阅读iostat的输出，各个参数都是什么含义，反映了磁盘的什么信息？

第一列Device比较容易理解，就是说这一行描述的是哪一个设备。

- rrqm/s：每秒合并读操作的次数
- wrqm/s: 每秒合并写操作的次数
- r/s ：每秒读操作的次数
- w/s：每秒写操作的次数
- rMB/s :每秒读取的MB字节数
- wMB/s: 每秒写入的MB字节数
- avgrq-sz：每个IO的平均扇区数，即所有请求的平均大小，以扇区（512字节）为单位
- avgqu-sz：平均为完成的IO请求数量，即平均意义山的请求队列长度
- await：平均每个IO所需要的时间，包括在队列等待的时间，也包括磁盘控制器处理本次请求的有效时间。
  - r_wait：每个读操作平均所需要的时间，不仅包括硬盘设备读操作的时间，也包括在内核队列中的时间。
  - w_wait: 每个写操平均所需要的时间，不仅包括硬盘设备写操作的时间，也包括在队列中等待的时间。
- svctm： 表面看是每个IO请求的服务时间，不包括等待时间，但是实际上，这个指标已经废弃。实际上，iostat工具没有任何一输出项表示的是硬盘设备平均每次IO的时间。
- %util： 工作时间或者繁忙时间占总时间的百分比

avgqu-sz 和繁忙程度

首先我们用超市购物来比对iostat的输出。我们在超市结账的时候，一般会有很多队可以排，队列的长度，在一定程度上反应了该收银柜台的繁忙程度。那么这个变量是avgqu-sz这个输出反应的，该值越大，表示排队等待处理的io越多。

我们搞4K的随机IO，但是iodepth=1 ，查看下fio的指令和iostat的输出：

```
 fio --name=randwrite --rw=randwrite --bs=4k --size=20G --runtime=1200 --ioengine=libaio --iodepth=1 --numjobs=1 --
filename=/dev/sdc --direct=1 --group_reporting
```

```
06/26/2017 10:54:02 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           7.79    0.00    2.85    0.04    0.00   89.32

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00     0.00    0.50    1.00     0.00     0.00     0.33     0.00    0.00    0.00    0.00   0.00   0.00
sdb               0.00    30.50    0.00   42.00     0.00     0.34    16.67     0.00    0.00    0.00    0.00   0.00   0.00
sdd               0.00     4.00    1.00   11.50     0.00     0.04     6.16     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              0.00     0.00    0.00    4.50     0.00     0.02     7.11     0.00    0.00    0.00    0.00   0.00   0.00
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    2.50     0.00     0.01     8.00     0.00    0.00    0.00    0.00   0.00   0.00
sdg               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
sdc               0.00     0.00    0.00  450.00     0.00     1.76     8.00     0.98    2.19    0.00    2.19   2.19  98.40
sdf               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
```

同样是4K的随机IO，我们设置iodepth=16， 查看fio的指令和iostat的输出：

```
fio --name=randwrite --rw=randwrite --bs=4k --size=20G --runtime=1200 --ioengine=libaio --iodepth=16 --numjobs=1 --
filename=/dev/sdc --direct=1 --group_reporting
```

```
06/26/2017 10:57:27 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.84    0.00    0.84    0.00    0.00   97.32

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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    25.00    0.00   17.50     0.00     0.25    29.49     0.00    0.00    0.00    0.00   0.00   0.00
sdd               0.00     0.00    0.00    3.00     0.00     0.01     4.00     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              0.00     0.00    0.00    1.00     0.00     0.00     4.00     0.00    0.00    0.00    0.00   0.00   0.00
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
sdg               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
sdc               0.00     0.00    0.00  479.50     0.00     1.87     8.00    15.98   33.31    0.00   33.31   2.09 100.00
sdf               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
```

注意，内核中有I/O Scheduler队列。我们看到因为avgqu-sz大小不一样，所以一个IO时间（await）就不一样。就好像你在超时排队，有一队没有人，而另一队队伍长度达到16 ，那么很明显，队伍长队为16的更繁忙一些。

avgrq-sz

avgrq-sz这个值反应了用户的IO-Pattern。我们经常关心，用户过来的IO是大IO还是小IO，那么avgrq-sz反应了这个要素。它的含义是说，平均下来，这这段时间内，所有请求的平均大小，单位是扇区，即（512字节）。

上面图中，sdc的avgrq-sz总是8，即8个扇区 = 8*512（Byte） = 4KB，这是因为我们用fio打io的时候，用的bs=4k。

下面我们测试当bs=128k时候的fio指令：

```
fio --name=randwrite --rw=randwrite --bs=128k --size=20G --runtime=1200 --ioengine=libaio --iodepth=1 --numjobs=1 --filename=/dev/sdc --direct=1 --group_reporting
```

```
06/26/2017 11:10:20 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.84    0.00    0.84    0.00    0.00   97.32

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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    27.50    0.00   39.50     0.00     0.31    15.90     0.00    0.05    0.00    0.05   0.05   0.20
sdd               0.00     5.50    0.00    3.00     0.00     0.03    20.00     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              0.00     0.00    0.00    4.50     0.00     0.02     8.00     0.00    0.00    0.00    0.00   0.00   0.00
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    3.00     0.00     0.01     8.00     0.00    0.00    0.00    0.00   0.00   0.00
sdg               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
sdc               0.00     0.00    0.00  296.50     0.00    37.06   256.00     0.99    3.33    0.00    3.33   3.33  98.80
sdf               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
```

注意sdc的avgrq-sz这列的值，变成了256，即256 个扇区 = 256* 512 Byte = 128KB，等于我们fio测试时，下达的bs = 128k。

注意，这个值也不是为所欲为的，它受内核参数的控制：

```
root@node-186:~# cat  /sys/block/sdc/queue/max_sectors_kb
256
```

这个值不是最大下发的IO是256KB，即512个扇区。当我们fio对sdc这块盘做测试的时候，如果bs=256k，iostat输出中的avgrq-sz 会变成 512 扇区，但是，如果继续增大bs，比如bs=512k，那么iostat输出中的avgrq-sz不会继续增大，仍然是512，表示512扇区。

```
fio --name=randwrite --rw=randwrite --bs=512k --size=20G --runtime=1200 --ioengine=libaio --iodepth=1 --numjobs=1 --filename=/dev/sdc --direct=1 --group_reporting
```

```
06/26/2017 11:26:06 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           6.15    0.00    2.43    0.04    0.00   91.38

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00     0.00    0.50    1.00     0.00     0.00     0.33     0.00    0.00    0.00    0.00   0.00   0.00
sdb               0.00   103.50    0.00  174.50     0.00    18.27   214.37     0.06    0.32    0.00    0.32   0.06   1.00
sdd               0.00     0.00    1.00    4.50     0.00     0.01     2.36     0.00    0.00    0.00    0.00   0.00   0.00
sde               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-0              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-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
sdg               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
sdc               0.00     0.00    0.00  253.00     0.00    63.25   512.00     1.58    6.26    0.00    6.26   3.93  99.40
sdf               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
```

注意，本来512KB等于1024个扇区，avgrq-sz应该为1204，但是由于内核的max_sectors_kb控制参数，决定了不可能：

另外一个需要注意也不难理解的现象是，io请求越大，需要消耗的时间就会越长。对于块设备而言，时间分成2个部分：

- 寻道
- 读或写操作

注意此处的寻道不能简单地理解成磁盘磁头旋转到指定位置，因为后备块设备可能是RAID，可能是SSD，我们理解写入前的准备动作。准备工作完成之后，写入4K和写入128KB，明显写入128KB的工作量要更大一些，因此很容易理解随机写入128KB给块设备带来的负载要比随机写入4K给块设备带来的负载要高一些。

对比生活中的例子，超时排队的时候，你会首先查看队列的长度来评估下时间，如果队列都差不多长的情况下，你就要关心前面顾客篮子里东西的多少了。如果前面顾客每人手里拿着一两件商品，另一队几乎每一个人都推这满满一车子的商品，你可能知道要排那一队。因为商品越多，处理单个顾客的时间就会越久。IO也是如此。

rrqm/s 和wrqm/s

块设备有相应的调度算法。如果两个IO发生在相邻的数据块时，他们可以合并成1个IO。

这个简单的可以理解为快递员要给一个18层的公司所有员工送快递，每一层都有一些包裹，对于快递员来说，最好的办法是同一楼层相近的位置的包裹一起投递，否则如果不采用这种算法，采用最原始的来一个送一个（即noop算法），那么这个快递员，可能先送了一个包括到18层，又不得不跑到2层送另一个包裹，然后有不得不跑到16层送第三个包裹，然后包到1层送第三个包裹，那么快递员的轨迹是杂乱无章的，也是非常低效的。

Linux常见的调度算法有： noop deadline和cfq。此处不展开了。

```
root@node-186:~# cat   /sys/block/sdc/queue/scheduler
[noop] deadline cfq
```

类比总结

我们还是以超时购物为例，比如一家三口去购物，各人买各人的东西，最终会汇总到收银台，你固然可以每人各自付各自的，但是也可以汇总一下，把所有购买的东西放在一起，由一个人来完成，也就说，三次收银事件merge成了一次。

至此，我们以超时购物收银为例，介绍了avgqu-sz 类比于队伍的长度，avgrq-sz 类比于每个人购物车里物品的多少，rrqm/s 和wrqm/s 类比于将一家购得东西汇总一起，付费一次。还有svctm和%util两个没有介绍。

按照我们的剧情，我们自然而然地可以将svctm类比成收银服务员服务每个客户需要的平均时间，%util类比成收银服务员工作的繁忙程度。

注意这个类比是错误的，就是因为类似的类比，容易让人陷入误区不能自拔。不能简单地将svctm理解成单个IO被块设备处理的有效时间，同时不能理解成%util到了100% ，磁盘工作就饱和了，不能继续提升了，这是两个常见的误区。

svctm和%util是iostat最容易引起误解的两个输出。为了准确地评估块设备的能力，我们希望得到这样一个数值：即一个io从发

给块设备层到完成这个io的时间，不包括其他在队列等待的时间。从表面看，svctm就是这个值。实际上并非如此。

Linux下iostat输出的svctm并不具备这方面的含义，这个指标应该非废弃。iostat和sar的man page都有这方面的警告：

```
svctm
The  average  service time (in milliseconds) for I/O requests that were issued to the device. Warning! Do not trust this
field any more.  This field will be removed in a future sysstat version.
```

那么iostat输出中的svctm到底是怎么来的，%util又是怎么算出来的，进而iostat的输出的各个字段都是从哪里拿到的信息呢？

iostat输出的数据来源diskstats

iostat数据的来源是Linux操作系统的/proc/diskstats:

```
   8       0 sda 441279 3 63361625 9989308 7202098 180461 3159771848 190075536 0 2367832 200113620
   8       1 sda1 181245 3 60611334 9815532 6903540 180429 3159771520 190063040 0 2295532 199901092
   8      16 sdb 46752 3720 2796200 321968 13825145 10875914 932601304 8639276 0 794500 8948212
   8      17 sdb1 65 0 520 156 0 0 0 0 0 156 156
   8      18 sdb2 45933 3713 2788786 320448 10098724 10875661 932598848 8615092 0 770332 8922960
   8      19 sdb3 256 7 2352 224 54 253 2456 216 0 408 440
   8      20 sdb4 256 0 2048 200 0 0 0 0 0 200 200
   8      48 sdd 802835 6159 11714469 1285300 30904844 13476361 5037471304 351602032 0 5499816 352912720
   8      49 sdd1 40625 927 2449632 131888 8300561 2174967 3263379696 300816392 0 4154232 300944072
   8      50 sdd2 476753 5231 5652328 893716 11753899 11301378 1774091448 40421684 0 2818820 41431892
   8      64 sde 381600 1515 6911703 347276 13884376 6491318 4153918128 211671968 0 3754336 212034176
   8      65 sde1 191 0 1528 20 0 0 0 0 0 20 20
   8      66 sde2 223 0 1854 20 0 0 0 0 0 20 20
 252       0 dm-0 389305 0 4212981 576212 16928509 0 910090394 67016608 0 2219732 67844444
 252       1 dm-1 18453 0 150820 375208 840266 0 88202706 3517972 0 820452 4081940
 252       2 dm-2 163891 0 60437372 10102232 5784388 0 2212756766 183475152 0 2317724 193844112
 252       3 dm-3 669 0 5496 64 2293071 0 18475608 45400760 0 1427608 45400980
   8      96 sdg 13532 0 109582 41756 14622176 43331 437926312 101278624 0 4022972 101322964
   8      32 sdc 5248826 0 41991890 1627200 20304863 0 275351704 60124188 0 4233524 61748640
   8      80 sdf 1989 0 16029 3456 665545 0 5324344 89125572 0 1397260 89128868
   8      81 sdf1 443 0 3544 452 0 0 0 0 0 276 452
   8      82 sdf2 14 0 106 60 0 0 0 0 0 60 60
   8      83 sdf3 27 0 202 296 0 0 0 0 0 184 296
```

注意，procfs中的前三个字段：主设备号、从设备号、设备名。这就不多说了。

从第四个字段开始，介绍的是该设备的相关统计：

- (rd_ios)：读操作的次数
- (rd_merges):合并读操作的次数。如果两个读操作读取相邻的数据块，那么可以被合并成1个。
- (rd_sectors): 读取的扇区数量
- (rd_ticks):读操作消耗的时间（以毫秒为单位）。每个读操作从__make_request()开始计时，到end_that_request_last()为止，包括了在队列中等待的时间。
- (wr_ios):写操作的次数
- (wr_merges):合并写操作的次数
- (wr_sectors): 写入的扇区数量
- (wr_ticks): 写操作消耗的时间（以毫秒为单位）
- (in_flight): 当前未完成的I/O数量。在I/O请求进入队列时该值加1，在I/O结束时该值减1。 注意：是I/O请求进入队列时，而不是提交给硬盘设备时
- (io_ticks)该设备用于处理I/O的自然时间(wall-clock time)
- (time_in_queue): 对字段#10(io_ticks)的加权值

这些字段大多来自内核的如下数据：

```
include/linux/genhd.h
struct disk_stats {
        unsigned long sectors[2];       /* READs and WRITEs */
        unsigned long ios[2];
        unsigned long merges[2];
        unsigned long ticks[2];
        unsigned long io_ticks;
        unsigned long time_in_queue;
};
```

除了in_flight来自：

```
part_in_flight(hd),
static inline int part_in_flight(struct hd_struct *part)
{
        return atomic_read(&part->in_flight[0]) + atomic_read(&part->in_flight[1]);
}
```

内核相关的代码如下：

```
while ((hd = disk_part_iter_next(&piter))) {
  cpu = part_stat_lock();
  part_round_stats(cpu, hd);
  part_stat_unlock();
  seq_printf(seqf, "%4d %7d %s %lu %lu %llu "
        "%u %lu %lu %llu %u %u %u %u\n",
        MAJOR(part_devt(hd)), MINOR(part_devt(hd)),
        disk_name(gp, hd->partno, buf),
        part_stat_read(hd, ios[READ]),
        part_stat_read(hd, merges[READ]),
        (unsigned long long)part_stat_read(hd, sectors[READ]),
        jiffies_to_msecs(part_stat_read(hd, ticks[READ])),
        part_stat_read(hd, ios[WRITE]),
        part_stat_read(hd, merges[WRITE]),
        (unsigned long long)part_stat_read(hd, sectors[WRITE]),
        jiffies_to_msecs(part_stat_read(hd, ticks[WRITE])),
        part_in_flight(hd),
        jiffies_to_msecs(part_stat_read(hd, io_ticks)),
        jiffies_to_msecs(part_stat_read(hd, time_in_queue))
     );
```

io_ticks and time_in_queue

这里面大部分字段都是很容易理解的，稍微难理解的在于io_ticks。初看之下，明明已经有了rd_ticks和wr_ticks 为什么还需一个io_ticks。注意rd_ticks和wr_ticks是把每一个IO消耗时间累加起来，但是硬盘设备一般可以并行处理多个IO，因此，rd_ticks和wr_ticks之和一般会比自然时间（wall-clock time）要大。而io_ticks 不关心队列中有多少个IO在排队，它只关心设备有IO的时间。即不考虑IO有多少，只考虑IO有没有。在实际运算中，in_flight不是0的时候保持计时，而in_flight 等于0的时候，时间不累加到io_ticks。

下一个比较难理解的是time_in_queue这个值，它的计算是当前IO数量（即in_flight的值）乘以自然时间间隔。表面看该变量的名字叫time_in_queue，但是实际上，并不只是在队列中等待的时间。

有人不理解time_in_queue，但是我相信读过小学 听过下面这句话的小朋友都会理解time_in_queue：

```
因为你上课讲话， 让老师批评你5分钟，班里有50人，50个人你就浪费了全班250分钟。
```

这段话非常形象地介绍了time_in_queue的计算法则，即自然时间只过去了5分钟，但是对于队列中的所有同学，哦不，所有IO来说，需要加权计算：

```
static void part_round_stats_single(int cpu, struct hd_struct *part,
                unsigned long now)
{
  if (now == part->stamp)
      return;

  /*如果队列不为空，存在in_flight io*/
  if (part_in_flight(part)) {

      /*小学数学老师的算法，now-part->stamp 乘以班级人数，哦不，是乘以队列中等待的io请求个数*/
      __part_stat_add(cpu, part, time_in_queue,
             part_in_flight(part) * (now - part->stamp));

     /*如实的记录，因为批评调皮学生，浪费了5分钟。io不是空的时间增加now - part->stamp*/
      __part_stat_add(cpu, part, io_ticks, (now - part->stamp));
  }
  part->stamp = now;
}
```

这个计算的方法很简单：

- 当请求队列为空的时候：
  - io_ticks不增加
  - time_in_queue不增加
  - part->stamp 更新为now
- 当请求队列不是空的时候：
  - io_ticks增加， 增加量为 now - part->timestamp
  - time_in_queue增加，增加量为 在队列中IO的个数乘以 (now - part->stamp)
  - part->stamp 更新为now

注意调用part_round_stats_single函数的时机在于：

- 在新IO请求插入队列（被merge的不算）
- 完成一个IO请求

空说太过抽象，但是我们还是给出一个例子来介绍io_ticks和time_in_queue的计算：

ID	Time	Ops	in_flight	stamp	stamp_delta	io_ticks	time_in_queue
0	100	新请求入队列	0	0	无需计算	0	0
1	100.10	新请求入队列	1	100	100.10-100 = 0.1	0.1	0.1
2	101.20	完成一个IO请求	2	100.10	101.20-100.10 = 1.1	1.2	0.1+1.1*2 = 2.3
3	103.60	完成一个IO请求	1	101.20	103.60-101.20 = 2.4	3.6	2.3+2.4*1=4.7
4	153.60	新请求入队列	0	103.60	无需计算	3.6	4.7
5	153.90	完成一个IO请求	1	153.60	153.90 - 153.60 = 0.3	3.9	4.7+0.3 * 1= 5

注意上面总时间是53.90时间内，有3.9秒的自然时间内是有IO的，即IO队列的非空时间为3.9秒。

注意，io_ticks这个字段被iostat用来计算%util，而time_in_queue这个字段被iostat用来计算avgqu-sz，即平均队列长度。

其实不难理解了，队列中不为空的时候占总时间的比例即为 %util

/proc/diskstats中其他数据项的更新

既然我们介绍了io_ticks和time_in_queue，我们也简单介绍下其他字段的获取。

在每个IO结束后，都会调用blk_account_io_done函数，这个函数会负责更新rd_ios/wr_ios、rd_ticks/wr_ticks ,包括会更新in_flight。

```
void blk_account_io_done(struct request *req)
{
        /*
         * Account IO completion.  flush_rq isn't accounted as a
         * normal IO on queueing nor completion.  Accounting the
         * containing request is enough.
         */
        if (blk_do_io_stat(req) && !(req->rq_flags & RQF_FLUSH_SEQ)) {
                unsigned long duration = jiffies - req->start_time;
                /*从req获取请求类型: R / W*/
                const int rw = rq_data_dir(req);
                struct hd_struct *part;
                int cpu;

                cpu = part_stat_lock();
                part = req->part;
               /*更新读或写次数，自加*/
                part_stat_inc(cpu, part, ios[rw]);
                /*将io的存活时间，更新到rd_ticks or wr_ticks*/
                part_stat_add(cpu, part, ticks[rw], duration);
                /*更新io_ticks和time_in_queue*/
                part_round_stats(cpu, part);
                /*对应infight 减 1 */
                part_dec_in_flight(part, rw);

                hd_struct_put(part);
                part_stat_unlock();
        }
}
```

注意part_round_stats会调用上一小节介绍的part_round_stats_single函数：

```
void part_round_stats(int cpu, struct hd_struct *part)
{
       /*既要更新分区的统计，也要更新整个块设备的统计*/
        unsigned long now = jiffies;
        if (part->partno)
                part_round_stats_single(cpu, &part_to_disk(part)->part0, now);
        part_round_stats_single(cpu, part, now);
}
```

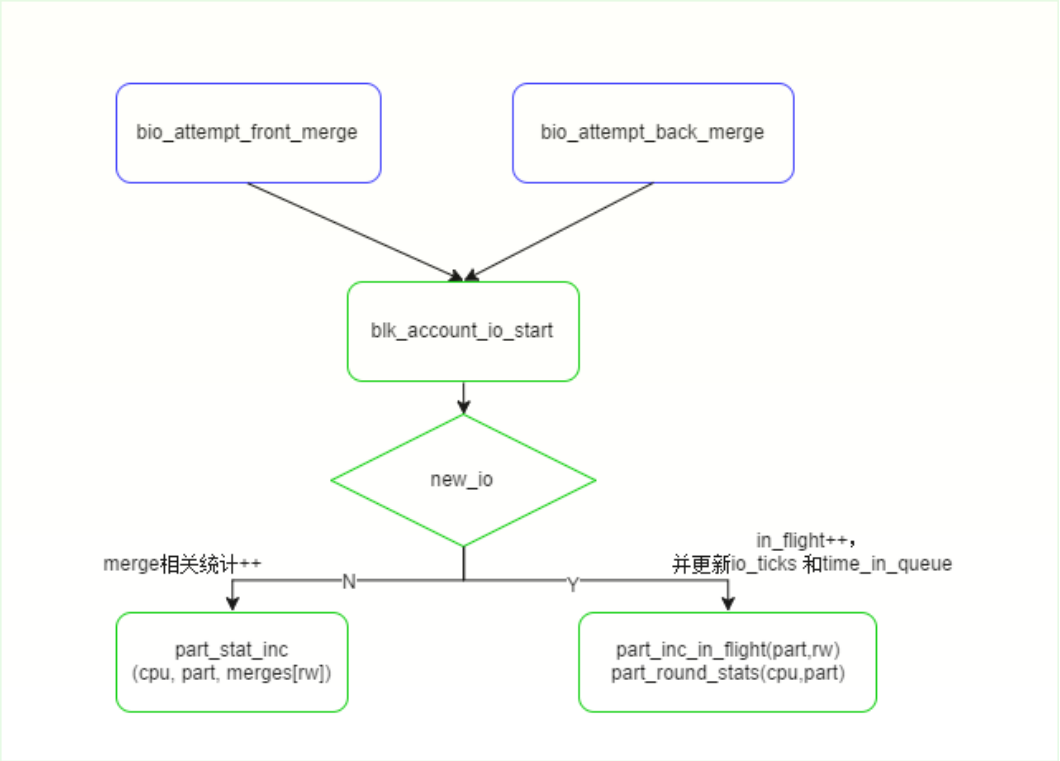
读写扇区的个数统计，是在blk_account_io_completion函数中实现的：

```
void blk_account_io_completion(struct request *req, unsigned int bytes)                                {
        if (blk_do_io_stat(req)) {
                const int rw = rq_data_dir(req);
                struct hd_struct *part;
                int cpu;

                cpu = part_stat_lock();
```

```
                part = req->part;
                /*右移9位，相当于除以512字节，即一个扇区的字节数*/
                part_stat_add(cpu, part, sectors[rw], bytes >> 9);
                part_stat_unlock();
        }
}
```

关于merge部分的统计，在blk_account_io_start函数中统计：




```
void blk_account_io_start(struct request *rq, bool new_io)
{
        struct hd_struct *part;
        int rw = rq_data_dir(rq);
        int cpu;

        if (!blk_do_io_stat(rq))
                return;

        cpu = part_stat_lock();

        if (!new_io) {
                /*注意，merge的IO就不会导致in_flight++*/
                part = rq->part;
                part_stat_inc(cpu, part, merges[rw]);
        } else {
                part = disk_map_sector_rcu(rq->rq_disk, blk_rq_pos(rq));
                if (!hd_struct_try_get(part)) {
                        part = &rq->rq_disk->part0;
                        hd_struct_get(part);
                }
                /*新IO，更新io_ticks and time_in_queue*/
                part_round_stats(cpu, part);
                /*in_flight 加1*/
                part_inc_in_flight(part, rw);
```

```
                rq->part = part;
        }

        part_stat_unlock();
}
```

iostat 输出的计算

注意，/proc/diskstats 已经将所有的素材都准备好了，对于iostat程序来说，就是将处理这些数据，给客户展现出更友好，更有意义的数值。事实上，iostat的源码非常的短，它属于sysstat这个开源软件，整个文件大小1619行。

```
int read_sysfs_file_stat(int curr, char *filename, char *dev_name)
{
        FILE *fp;
        struct io_stats sdev;
        int i;
        unsigned int ios_pgr, tot_ticks, rq_ticks, wr_ticks;
        unsigned long rd_ios, rd_merges_or_rd_sec, wr_ios, wr_merges;
        unsigned long rd_sec_or_wr_ios, wr_sec, rd_ticks_or_wr_sec;

        /* Try to read given stat file */
        if ((fp = fopen(filename, "r")) == NULL)
                return 0;

        i = fscanf(fp, "%lu %lu %lu %lu %lu %lu %lu %u %u %u %u",
                   &rd_ios, &rd_merges_or_rd_sec, &rd_sec_or_wr_ios, &rd_ticks_or_wr_sec,
                   &wr_ios, &wr_merges, &wr_sec, &wr_ticks, &ios_pgr, &tot_ticks, &rq_ticks);

        if (i == 11) {
                /* Device or partition */
                sdev.rd_ios     = rd_ios;
                sdev.rd_merges  = rd_merges_or_rd_sec;
                sdev.rd_sectors = rd_sec_or_wr_ios;
                sdev.rd_ticks   = (unsigned int) rd_ticks_or_wr_sec;
                sdev.wr_ios     = wr_ios;
                sdev.wr_merges  = wr_merges;
                sdev.wr_sectors = wr_sec;
                sdev.wr_ticks   = wr_ticks;
                sdev.ios_pgr    = ios_pgr;
                sdev.tot_ticks  = tot_ticks;
                sdev.rq_ticks   = rq_ticks;
        }
        else if (i == 4) {
                /* Partition without extended statistics */
                sdev.rd_ios     = rd_ios;
                sdev.rd_sectors = rd_merges_or_rd_sec;
                sdev.wr_ios     = rd_sec_or_wr_ios;
                sdev.wr_sectors = rd_ticks_or_wr_sec;
        }
        if ((i == 11) || !DISPLAY_EXTENDED(flags)) {
                /*
                 * In fact, we _don't_ save stats if it's a partition without
                 * extended stats and yet we want to display ext stats.
                 */
                save_stats(dev_name, curr, &sdev, iodev_nr, st_hdr_iodev);
        }

        fclose(fp);
```

```
        return 1;
}
```

数据都采集到了，剩下就是计算了。其中下面几项的计算是非常简单的：

- rrqm/s
- wrqm/s
- r/s
- w/s
- rMB/s
- wMB/s

这几项的计算是非常简单的，就是采样两次，后一次的值减去前一次的值，然后除以时间间隔，得到平均值即可。因为这些/proc/diskstats中对应的值都是累加的，后一次减去前一次，即得到采样时间间隔内的新增量。不赘述。

avgrq-sz的计算

```
    /*       rrq/s wrq/s   r/s   w/s  rsec  wsec  rqsz  qusz await r_await w_await svctm %util */
       cprintf_f(2, 8, 2,
                 S_VALUE(ioj->rd_merges, ioi->rd_merges, itv),
                 S_VALUE(ioj->wr_merges, ioi->wr_merges, itv));
       cprintf_f(2, 7, 2,
                 S_VALUE(ioj->rd_ios, ioi->rd_ios, itv),
                 S_VALUE(ioj->wr_ios, ioi->wr_ios, itv));
       cprintf_f(4, 8, 2,
                 S_VALUE(ioj->rd_sectors, ioi->rd_sectors, itv) / fctr,
                 S_VALUE(ioj->wr_sectors, ioi->wr_sectors, itv) / fctr,
                 xds.arqsz,  //此处是avgrq-sz
                 S_VALUE(ioj->rq_ticks, ioi->rq_ticks, itv) / 1000.0);//此处是avgqu-sz
```

注意avgrq-sz来自xds的argsz变量，该变量是通过该函数计算得到的：

```
/*注意sdc中的c指的是current，sdp中的p指的是previous*/
void compute_ext_disk_stats(struct stats_disk *sdc, struct stats_disk *sdp,
                            unsigned long long itv, struct ext_disk_stats *xds)
{
       double tput
              = ((double) (sdc->nr_ios - sdp->nr_ios)) * HZ / itv;

       xds->util  = S_VALUE(sdp->tot_ticks, sdc->tot_ticks, itv);
       xds->svctm = tput ? xds->util / tput : 0.0;
       xds->await = (sdc->nr_ios - sdp->nr_ios) ?
              ((sdc->rd_ticks - sdp->rd_ticks) + (sdc->wr_ticks - sdp->wr_ticks)) /
              ((double) (sdc->nr_ios - sdp->nr_ios)) : 0.0;

       xds->arqsz = (sdc->nr_ios - sdp->nr_ios) ?
              ((sdc->rd_sect - sdp->rd_sect) + (sdc->wr_sect - sdp->wr_sect)) /
              ((double) (sdc->nr_ios - sdp->nr_ios)) : 0.0;
}
```

注意nr_ios来自如下运算，即读IO和写IO的和

```
       sdc.nr_ios    = ioi->rd_ios + ioi->wr_ios;
       sdp.nr_ios    = ioj->rd_ios + ioj->wr_ios;
```

那么xds->arqsz 的计算就是如下含义：

```
     xds->arqsz = (读扇区总数 + 写扇区总数)/(读IO次数+写IO次数)
     xds->arqsz = (sdc->nr_ios - sdp->nr_ios) ?
              ((sdc->rd_sect - sdp->rd_sect) + (sdc->wr_sect - sdp->wr_sect)) /
              ((double) (sdc->nr_ios - sdp->nr_ios)) : 0.0;
```

OK非常容易理解，而且计算也是很合理的。

avgqu-sz的计算

平均队列长度的计算，这个计算就用到了diskstats中time_in_queue这个值。

这个值的计算来自这句话：

```
S_VALUE(ioj->rq_ticks, ioi->rq_ticks, itv) / 1000.0)
```

其中rq_ticks即diskstats中的time_in_queue。

我们考虑如下的场景，如果IO请求有一个burst，同一时间来了250个IO请求，后续再也没有新的请求到来。这种情况下，每个请求处理时间都是4ms，那么所有IO的平均等待时间为：

```
平均等待时间 = 单个请求处理时间*(1+2+3+4...+(请求总数-1))/请求总数
```

对于我们这个例子而言，平均等待时间是4*125 = 500 ms

那么所有IO花费的总时间为250*500=125000毫秒，这个时间除以1000毫秒：

```
125000/1000 = 125
```

即平均下来，队列的长度是125 ，这个值很明显是符合直观的。排在队列最前端的IO认为，队列的长度是0，第2个IO认为队列的长度是1，第3个IO认为队列的长度是2，最后一个认为队列的长度是249。

我们换一种思路来考虑，即diskstats中time_in_queue的思路。

当第一个IO完成的时候，队列中250个IO，250个IO都等了4ms，即time_in_queue + = (250*4) ，当第二个IO完成的时候，time_in_queue += (249*4)，当所有IO都完成的时候，time_in_queue = 4*(250+249+248....+1)， ...

根据time_in_queue/1000,殊途同归地获得了平均队列长度。

await、r_wait及w_wait的计算

```
void compute_ext_disk_stats(struct stats_disk *sdc, struct stats_disk *sdp,
                            unsigned long long itv, struct ext_disk_stats *xds)
{
        ...
        xds->await = (sdc->nr_ios - sdp->nr_ios) ?
                ((sdc->rd_ticks - sdp->rd_ticks) + (sdc->wr_ticks - sdp->wr_ticks)) /
                ((double) (sdc->nr_ios - sdp->nr_ios)) : 0.0;
        ...
}
```

这个没啥好说的了：

```
await = ((所有读IO的时间)+(所有写IO的时间))/((读请求的个数) + (写请求的个数))
```

注意一点就行了，这个所有读IO的时间和所有写IO的时间，都是包括IO在队列的时间在内的。不能一厢情愿地认为，是磁盘控

制器处理该IO的时间。

注意，能不能说，await比较高，所以武断地判定这块盘的能力很菜？答案是不能。await这个值不能反映硬盘设备的性能。await的这个值不能反映硬盘设备的性能，await这个值不能反映硬盘设备的性能，重要的话讲三遍。

我们考虑两种IO的模型：

- 250个IO请求同时进入等待队列
- 250个IO请求依次发起，待上一个IO完成后，发起下一个IO

第一种情况await高达500ms，第二个情况await只有4ms，但是都是同一块盘。

但是注意await是相当重要的一个参数，它表明了用户发起的IO请求的平均延迟：

```
await  = IO 平均处理时间 + IO在队列的平均等待时间
```

因此，这个指标是比较重要的一个指标。

%util 和磁盘设备饱和度

注意，%util是最容易让人产生误解的一个参数。很多初学者看到%util 等于100%就说硬盘能力到顶了，这种说法是错误的。

%util数据源自diskstats中的io_ticks，这个值并不关心等待在队里里面IO的个数，它只关心队列中有没有IO。

和超时排队结账这个类比最本质的区别在于，现代硬盘都有并行处理多个IO的能力，但是收银员没有。收银员无法做到同时处理10个顾客的结账任务而消耗的总时间与处理一个顾客结账任务相差无几。但是磁盘可以。所以，即使%util到了100%，也并不意味着设备饱和了。

最简单的例子是，某硬盘处理单个IO请求需要0.1秒，有能力同时处理10个。但是当10个请求依次提交的时候，需要1秒钟才能完成这10%的请求，，在1秒的采样周期里，%util达到了100%。但是如果10个请一次性提交的话， 硬盘可以在0.1秒内全部完成，这时候，%util只有10%。

因此，在上面的例子中，一秒中10个IO，即IOPS=10的时候，%util就达到了100%，这并不能表明，该盘的IOPS就只能到10，事实上，纵使%util到了100%，硬盘可能仍然有很大的余力处理更多的请求，即并未达到饱和的状态。

下一小节有4张图，可以看到当IOPS为1000的时候%util为100%，但是并不意味着该盘的IOPS就在1000，实际上2000，3000,5000的IOPS都可以达到。根据%util 100%时的 r/s 或w/s 来推算磁盘的IOPS是不对的。

那么有没有一个指标用来衡量硬盘设备的饱和程度呢。很遗憾，iostat没有一个指标可以衡量磁盘设备的饱和度。

svctm的计算

对于iostat这个功能而言，%util固然会给人带来一定的误解和苦扰，但是svctm给人带来的误解更多。一直以来，人们希望了解块设备处理单个IO的service time，这个指标直接地反应了硬盘的能力。

回到超市收银这个类比中，如果收银员是个老手，操作流，效率很高，那么大家肯定更愿意排这一队。但是如果收银员是个新手，各种操作不熟悉，动作慢，效率很低，那么同样多的任务，就会花费更长的时间。因此IO的平均service time（不包括排队时间）是非常有意义的。

但是service time和iostat无关，iostat没有任何一个参数能够提供这方面的信息。而svctm这个输出给了人们这种美好的期待，却只能让人空欢喜。

从现在起，我们记住，我们不能从svctm中得到自己期待的service time这个值，这个值其实并没有什么意义，事实上，这个值不是独立的，它是根据其他值计算出来的。

```
void compute_ext_disk_stats(struct stats_disk *sdc, struct stats_disk *sdp,
                            unsigned long long itv, struct ext_disk_stats *xds)
```

```
{
        double tput
                = ((double) (sdc->nr_ios - sdp->nr_ios)) * HZ / itv;

        xds->util  = S_VALUE(sdp->tot_ticks, sdc->tot_ticks, itv);
        xds->svctm = tput ? xds->util / tput : 0.0;
        ...
}
```

如果一个盘的能力很强悍，随机小IO（4K）fio测试中我们会看到如下现象：当IOPS为1000的时候，iosta输出的svctm为1(ms)，当IOPS为2000的时候，iostat输出的svctm为0.5(ms),当IOPS为3000的时候，iostat输出的svctm为0.33。原因其实无他，因为这种情况下%util都是100%，即当采样周期是1秒的时候，用满了1秒，tput就是fio指定的–rate-iops 即1000、2000、3000，因此算出来svctm为对应的1、0.5、0.33。

```
06/29/2017 11:13:56 AM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.13    0.00    0.42    0.00    0.00   99.46

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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     3.50    0.00    1.00     0.00     0.02    36.00     0.00    0.00    0.00    0.00   0.00   0.00
sdc               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
sdd               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
sde               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
sdf               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
sdg               0.00     0.00    0.00 1000.50     0.00     3.91     8.00     1.47    1.47    0.00    1.47   1.00 100.00
```

```
06/29/2017 11:12:23 AM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.29    0.00    0.67    0.00    0.00   99.04

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
sdc               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
sdd               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
sde               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
sdf               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
sdg               0.00     0.00    0.00 2001.00     0.00     7.82     8.00     2.48    1.24    0.00    1.24   0.50 100.00
```

```
06/29/2017 11:15:24 AM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.13    0.00    0.71    0.00    0.00   99.16

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
sdc               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
sdd               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
sde               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
sdf               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
sdg               0.00     0.00    0.00 3014.50     0.00    11.78     8.00     3.76    1.24    0.00    1.24   0.33 100.00
```

```
06/29/2017 11:17:18 AM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.34    0.00    0.96    0.00    0.00   98.70

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/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     3.50    0.00    1.00     0.00     0.02    36.00     0.00    0.00    0.00    0.00   0.00   0.00
sdc               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
sdd               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
sde               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
sdf               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
sdg               0.00     0.00    0.00 5087.00     0.00    19.87     8.00     8.17    1.61    0.00    1.61   0.20 100.00
```

（注意上面的盘sdg是iSCSI，存储空间是由分布式存储提供，不要问我为什么单个盘随机IOPS能无压力的到5000）

因此从这个例子看，把iostat的输出中的svctm看作是IO的处理时间是相当不靠谱的。为了防止带来的误解，可以直接忽略这个参数。

既然svctm不能反映IO处理时间，那么有没有一个参数可以测量块设备的IO平均处理时间呢？很遗憾iostat是做不到的。但是只要思想不滑坡，办法总比困难多，blktrace这个神器可能得到这个设备的IO平均处理时间。

接下来我们就可以进入另一个天地。

尾声

iostat能够提供给我们的信息就这么多了，通过分析我们期待能够得到块设备处理IO的时间，这就要靠blocktrace这个工具了。blktrace可以讲IO路径分段，分别统计各段的消耗的时间。

本文大量参考vmunix的容易被误读的IOSTAT，以及深入分析diskstats，其中第二篇文章给出了一个很详细的IO PATH的流程图，非常有用。第二篇文章中随着代码演进有一些变化，本文采用的比较新的Linux Kernel code做介绍，同时演算io_ticks和time_in_queue部分第二篇文章也有错误，也一并修正了。不过瑕不掩瑜，这两篇都是非常棒的文章。向前辈致敬。

上一篇　下一篇