系统优化是一项复杂、繁琐、长期的工作，优化前需要监测、采集、测试、评估，优化后也需要测试、采集、评估、监测，而且是一个长期和持续的过程，不 是说现在优化了，测试了，以后就可以一劳永逸了，也不是说书本上的优化就适合眼下正在运行的系统，不同的系统、不同的硬件、不同的应用优化的重点也不同、 优化的方法也不同、优化的参数也不同。性能监测是系统优化过程中重要的一环，如果没有监测、不清楚性能瓶颈在哪里，优化什么呢、怎么优化呢？所以找到性能 瓶颈是性能监测的目的，也是系统优化的关键。系统由若干子系统构成，通常修改一个子系统有可能影响到另外一个子系统，甚至会导致整个系统不稳定、崩溃。所 以说优化、监测、测试通常是连在一起的，而且是一个循环而且长期的过程，通常监测的子系统有以下这些：

* CPU
* Memory
* IO
* Network

这些子系统互相依赖，了解这些子系统的特性，监测这些子系统的性能参数以及及时发现可能会出现的瓶颈对系统优化很有帮助。

**应用类型**

不同的系统用途也不同，要找到性能瓶颈需要知道系统跑的是什么应用、有些什么特点，比如 web server 对系统的要求肯定和 file server 不一样，所以分清不同系统的应用类型很重要，通常应用可以分为两种类型：

* IO 相关，IO 相关的应用通常用来处理大量数据，需要大量内存和存储，频繁 IO 操作读写数据，而对 CPU 的要求则较少，大部分时候 CPU 都在等待硬盘，比如，数据库服务器、文件服务器等。
* CPU 相关，CPU 相关的应用需要使用大量 CPU，比如高并发的 web/mail 服务器、图像/视频处理、科学计算等都可被视作 CPU 相关的应用。

看看实际中的例子，第1个是文件服务器拷贝一个大文件时表现出来的特征，第2个是 CPU 做大量计算时表现出来的特征：

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

0 4 140 1962724 335516 4852308 0 0 388 65024 1442 563 0 2 47 52 0

0 4 140 1961816 335516 4853868 0 0 768 65536 1434 522 0 1 50 48 0

0 4 140 1960788 335516 4855300 0 0 768 48640 1412 573 0 1 50 49 0

0 4 140 1958528 335516 4857280 0 0 1024 65536 1415 521 0 1 41 57 0

0 5 140 1957488 335516 4858884 0 0 768 81412 1504 609 0 2 50 49 0

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

4 0 140 3625096 334256 3266584 0 0 0 16 1054 470 100 0 0 0 0

4 0 140 3625220 334264 3266576 0 0 0 12 1037 448 100 0 0 0 0

4 0 140 3624468 334264 3266580 0 0 0 148 1160 632 100 0 0 0 0

4 0 140 3624468 334264 3266580 0 0 0 0 1078 527 100 0 0 0 0

4 0 140 3624712 334264 3266580 0 0 0 80 1053 501 100 0 0 0 0

上面两个例子最明显的差别就是 id 一栏，代表 CPU 的空闲率，拷贝文件时候 id 维持在 50％ 左右，CPU 大量计算的时候 id 基本为 0。

**底线**

我们如何知道系统性能是好还是差呢？这需要事先建立一个底线，如果性能监测得到的统计数据跨过这条线，我们就可以说这个系统性能差，如果数据能保持 在线内我们就说性能好。建立这样底线需要知道一些理论、额外的负载测试和系统管理员多年的经验。如果自己没有多年的经验，有一个简单划底线的办法就是：把 这个底线建立在自己对系统的期望上。自己期望这个系统有个什么样的性能，这是一个底线，如果没有达到这个要求就是性能差。比如，VPSee 上个月有个 RAID0 的测试，期望的测试结果应该是 RAID0 的 IO 性能比单硬盘有显著提高，底线是 RAID0 的 IO 至少要比单硬盘要好（好多少不重要，底线是至少要好），测试结果却发现 RAID0 性能还不如单硬盘，说明性能差，这个时候需要问个为什么，这往往是性能瓶颈所在，经过排查发现是原硬盘有硬件瑕疵造成性能测试结果错误。

**监测工具**

我们只需要简单的工具就可以对 Linux 的性能进行监测，以下是 VPSee 常用的工具：

| **工具** | **简单介绍** |
| --- | --- |
| top | 查看进程活动状态以及一些系统状况 |
| vmstat | 查看系统状态、硬件和系统信息等 |
| iostat | 查看CPU 负载，硬盘状况 |
| sar | 综合工具，查看系统状况 |
| mpstat | 查看多处理器状况 |
| netstat | 查看网络状况 |
| iptraf | 实时网络状况监测 |
| tcpdump | 抓取网络数据包，详细分析 |
| mpstat | 查看多处理器状况 |
| tcptrace | 数据包分析工具 |
| netperf | 网络带宽工具 |
| dstat | 综合工具，综合了 vmstat, iostat, ifstat, netstat 等多个信息 |

本系列将按照CPU、内存、磁盘IO、网络这几个方面分别介绍。

# Linux性能监测：CPU篇

CPU 的占用主要取决于什么样的资源正在 CPU 上面运行，比如拷贝一个文件通常占用较少 CPU，因为大部分工作是由 DMA（Direct Memory Access）完成，只是在完成拷贝以后给一个中断让 CPU 知道拷贝已经完成；科学计算通常占用较多的 CPU，大部分计算工作都需要在 CPU 上完成，内存、硬盘等子系统只做暂时的数据存储工作。要想监测和理解 CPU 的性能需要知道一些的操作系统的基本知识，比如：中断、进程调度、进程上下文切换、可运行队列等。这里 VPSee 用个例子来简单介绍一下这些概念和他们的关系，CPU 很无辜，是个任劳任怨的打工仔，每时每刻都有工作在做（进程、线程）并且自己有一张工作清单（可运行队列），由老板（进程调度）来决定他该干什么，他需要 和老板沟通以便得到老板的想法并及时调整自己的工作（上下文切换），部分工作做完以后还需要及时向老板汇报（中断），所以打工仔（CPU）除了做自己该做 的工作以外，还有大量时间和精力花在沟通和汇报上。

CPU 也是一种硬件资源，和任何其他硬件设备一样也需要驱动和管理程序才能使用，我们可以把内核的进程调度看作是 CPU 的管理程序，用来管理和分配 CPU 资源，合理安排进程抢占 CPU，并决定哪个进程该使用 CPU、哪个进程该等待。操作系统内核里的进程调度主要用来调度两类资源：进程（或线程）和中断，进程调度给不同的资源分配了不同的优先级，优先级最高的 是硬件中断，其次是内核（系统）进程，最后是用户进程。每个 CPU 都维护着一个可运行队列，用来存放那些可运行的线程。线程要么在睡眠状态（blocked 正在等待 IO）要么在可运行状态，如果 CPU 当前负载太高而新的请求不断，就会出现进程调度暂时应付不过来的情况，这个时候就不得不把线程暂时放到可运行队列里。VPSee 在这里要讨论的是性能监测，上面谈了一堆都没提到性能，那么这些概念和性能监测有什么关系呢？关系重大。如果你是老板，你如何检查打工仔的效率（性能） 呢？我们一般会通过以下这些信息来判断打工仔是否偷懒：

* 打工仔接受和完成多少任务并向老板汇报了（中断）；
* 打工仔和老板沟通、协商每项工作的工作进度（上下文切换）；
* 打工仔的工作列表是不是都有排满（可运行队列）；
* 打工仔工作效率如何，是不是在偷懒（CPU 利用率）。

现在把打工仔换成 CPU，我们可以通过查看这些重要参数：中断、上下文切换、可运行队列、CPU 利用率来监测 CPU 的性能。

**底线**

上一篇 Linux 性能监测：介绍 提到了性能监测前需要知道底线，那么监测 CPU 性能的底线是什么呢？通常我们期望我们的系统能到达以下目标：

* CPU 利用率，如果 CPU 有 100％ 利用率，那么应该到达这样一个平衡：65％－70％ User Time，30％－35％ System Time，0％－5％ Idle Time；
* 上下文切换，上下文切换应该和 CPU 利用率联系起来看，如果能保持上面的 CPU 利用率平衡，大量的上下文切换是可以接受的；
* 可运行队列，每个可运行队列不应该有超过1－3个线程（每处理器），比如：双处理器系统的可运行队列里不应该超过6个线程。

**vmstat**

vmstat 是个查看系统整体性能的小工具，小巧、即使在很 heavy 的情况下也运行良好，并且可以用时间间隔采集得到连续的性能数据。

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

2 1 140 2787980 336304 3531996 0 0 0 128 1166 5033 3 3 70 25 0

0 1 140 2788296 336304 3531996 0 0 0 0 1194 5605 3 3 69 25 0

0 1 140 2788436 336304 3531996 0 0 0 0 1249 8036 5 4 67 25 0

0 1 140 2782688 336304 3531996 0 0 0 0 1333 7792 6 6 64 25 0

3 1 140 2779292 336304 3531992 0 0 0 28 1323 7087 4 5 67 25 0

参数介绍：

* r，可运行队列的线程数，这些线程都是可运行状态，只不过 CPU 暂时不可用；
* b，被 blocked 的进程数，正在等待 IO 请求；
* in，被处理过的中断数
* cs，系统上正在做上下文切换的数目
* us，用户占用 CPU 的百分比
* sys，内核和中断占用 CPU 的百分比
* wa，所有可运行的线程被 blocked 以后都在等待 IO，这时候 CPU 空闲的百分比
* id，CPU 完全空闲的百分比

举两个现实中的例子来实际分析一下：

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

4 0 140 2915476 341288 3951700 0 0 0 0 1057 523 19 81 0 0 0

4 0 140 2915724 341296 3951700 0 0 0 0 1048 546 19 81 0 0 0

4 0 140 2915848 341296 3951700 0 0 0 0 1044 514 18 82 0 0 0

4 0 140 2915848 341296 3951700 0 0 0 24 1044 564 20 80 0 0 0

4 0 140 2915848 341296 3951700 0 0 0 0 1060 546 18 82 0 0 0

从上面的数据可以看出几点：

1. interrupts（in）非常高，context switch（cs）比较低，说明这个 CPU 一直在不停的请求资源；
2. user time（us）一直保持在 80％ 以上，而且上下文切换较低（cs），说明某个进程可能一直霸占着 CPU；
3. run queue（r）刚好在4个。

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

14 0 140 2904316 341912 3952308 0 0 0 460 1106 9593 36 64 1 0 0

17 0 140 2903492 341912 3951780 0 0 0 0 1037 9614 35 65 1 0 0

20 0 140 2902016 341912 3952000 0 0 0 0 1046 9739 35 64 1 0 0

17 0 140 2903904 341912 3951888 0 0 0 76 1044 9879 37 63 0 0 0

16 0 140 2904580 341912 3952108 0 0 0 0 1055 9808 34 65 1 0 0

从上面的数据可以看出几点：

1. context switch（cs）比 interrupts（in）要高得多，说明内核不得不来回切换进程；
2. 进一步观察发现 system time（sy）很高而 user time（us）很低，而且加上高频度的上下文切换（cs），说明正在运行的应用程序调用了大量的系统调用（system call）；
3. run queue（r）在14个线程以上，按照这个测试机器的硬件配置（四核），应该保持在12个以内。

**mpstat**

mpstat 和 vmstat 类似，不同的是 mpstat 可以输出多个处理器的数据，下面的输出显示 CPU1 和 CPU2 基本上没有派上用场，系统有足够的能力处理更多的任务。

$ mpstat -P ALL 1

Linux 2.6.18-164.el5 (vpsee) 11/13/2009

02:24:33 PM CPU %user %nice %sys %iowait %irq %soft %steal %idle intr/s

02:24:34 PM all 5.26 0.00 4.01 25.06 0.00 0.00 0.00 65.66 1446.00

02:24:34 PM 0 7.00 0.00 8.00 0.00 0.00 0.00 0.00 85.00 1001.00

02:24:34 PM 1 13.00 0.00 8.00 0.00 0.00 0.00 0.00 79.00 444.00

02:24:34 PM 2 0.00 0.00 0.00 100.00 0.00 0.00 0.00 0.00 0.00

02:24:34 PM 3 0.99 0.00 0.99 0.00 0.00 0.00 0.00 98.02 0.00

**ps**

如何查看某个程序、进程占用了多少 CPU 资源呢？下面是 Firefox 在 VPSee 的一台 Sunray 服务器上的运行情况，当前只有2个用户在使用 Firefox：

$ while :; do ps -eo pid,ni,pri,pcpu,psr,comm | grep 'firefox'; sleep 1; done

PID NI PRI %CPU PSR COMMAND

7252 0 24 3.2 3 firefox

9846 0 24 8.8 0 firefox

7252 0 24 3.2 2 firefox

9846 0 24 8.8 0 firefox

7252 0 24 3.2 2 firefox

# Linux性能监测：内存篇

这里的讲到的 “内存” 包括物理内存和虚拟内存，虚拟内存（Virtual Memory）把计算机的内存空间扩展到硬盘，物理内存（RAM）和硬盘的一部分空间（SWAP）组合在一起作为虚拟内存为计算机提供了一个连贯的虚拟内 存空间，好处是我们拥有的内存 ”变多了“，可以运行更多、更大的程序，坏处是把部分硬盘当内存用整体性能受到影响，硬盘读写速度要比内存慢几个数量级，并且 RAM 和 SWAP 之间的交换增加了系统的负担。

在操作系统里，虚拟内存被分成页，在 x86 系统上每个页大小是 4KB。Linux 内核读写虚拟内存是以 “页” 为单位操作的，把内存转移到硬盘交换空间（SWAP）和从交换空间读取到内存的时候都是按页来读写的。内存和 SWAP 的这种交换过程称为页面交换（Paging），值得注意的是 paging 和 swapping 是两个完全不同的概念，国内很多参考书把这两个概念混为一谈，swapping 也翻译成交换，在操作系统里是指把某程序完全交换到硬盘以腾出内存给新程序使用，和 paging 只交换程序的部分（页面）是两个不同的概念。纯粹的 swapping 在现代操作系统中已经很难看到了，因为把整个程序交换到硬盘的办法既耗时又费力而且没必要，现代操作系统基本都是 paging 或者 paging/swapping 混合，swapping 最初是在 Unix system V 上实现的。

虚拟内存管理是 Linux 内核里面最复杂的部分，要弄懂这部分内容可能需要一整本书的讲解。VPSee 在这里只介绍和性能监测有关的两个内核进程：kswapd 和 pdflush。

* kswapd daemon 用来检查 pages\_high 和 pages\_low，如果可用内存少于 pages\_low，kswapd 就开始扫描并试图释放 32个页面，并且重复扫描释放的过程直到可用内存大于 pages\_high 为止。扫描的时候检查3件事：1）如果页面没有修改，把页放到可用内存列表里；2）如果页面被文件系统修改，把页面内容写到磁盘上；3）如果页面被修改 了，但不是被文件系统修改的，把页面写到交换空间。
* pdflush daemon 用来同步文件相关的内存页面，把内存页面及时同步到硬盘上。比如打开一个文件，文件被导入到内存里，对文件做了修改后并保存后，内核并不马上保存文件到硬 盘，由 pdflush 决定什么时候把相应页面写入硬盘，这由一个内核参数 vm.dirty\_background\_ratio 来控制，比如下面的参数显示脏页面（dirty pages）达到所有内存页面10％的时候开始写入硬盘。

# /sbin/sysctl -n vm.dirty\_background\_ratio

10

**vmstat**

继续 vmstat 一些参数的介绍，上一篇 Linux 性能监测：CPU 介绍了 vmstat 的部分参数，这里介绍另外一部分。以下数据来自 VPSee 的一个 256MB RAM，512MB SWAP 的 Xen VPS：

# vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

0 3 252696 2432 268 7148 3604 2368 3608 2372 288 288 0 0 21 78 1

0 2 253484 2216 228 7104 5368 2976 5372 3036 930 519 0 0 0 100 0

0 1 259252 2616 128 6148 19784 18712 19784 18712 3821 1853 0 1 3 95 1

1 2 260008 2188 144 6824 11824 2584 12664 2584 1347 1174 14 0 0 86 0

2 1 262140 2964 128 5852 24912 17304 24952 17304 4737 2341 86 10 0 0 4

* swpd，已使用的 SWAP 空间大小，KB 为单位；
* free，可用的物理内存大小，KB 为单位；
* buff，物理内存用来缓存读写操作的 buffer 大小，KB 为单位；
* cache，物理内存用来缓存进程地址空间的 cache 大小，KB 为单位；
* si，数据从 SWAP 读取到 RAM（swap in）的大小，KB 为单位；
* so，数据从 RAM 写到 SWAP（swap out）的大小，KB 为单位；
* bi，磁盘块从文件系统或 SWAP 读取到 RAM（blocks in）的大小，block 为单位；
* bo，磁盘块从 RAM 写到文件系统或 SWAP（blocks out）的大小，block 为单位；

上面是一个频繁读写交换区的例子，可以观察到以下几点：

* 物理可用内存 free 基本没什么显著变化，swapd 逐步增加，说明最小可用的内存始终保持在 256MB X 10％ = 2.56MB 左右，当脏页达到10％的时候（vm.dirty\_background\_ratio ＝ 10）就开始大量使用 swap；
* buff 稳步减少说明系统知道内存不够了，kwapd 正在从 buff 那里借用部分内存；
* kswapd 持续把脏页面写到 swap 交换区（so），并且从 swapd 逐渐增加看出确实如此。根据上面讲的 kswapd 扫描时检查的三件事，如果页面被修改了，但不是被文件系统修改的，把页面写到 swap，所以这里 swapd 持续增加。

# Linux性能监测：磁盘IO篇

磁盘通常是计算机最慢的子系统，也是最容易出现性能瓶颈的地方，因为磁盘离 CPU 距离最远而且 CPU 访问磁盘要涉及到机械操作，比如转轴、寻轨等。访问硬盘和访问内存之间的速度差别是以数量级来计算的，就像1天和1分钟的差别一样。要监测 IO 性能，有必要了解一下基本原理和 Linux 是如何处理硬盘和内存之间的 IO 的。

**内存页**

上一篇 Linux 性能监测：Memory 提到了内存和硬盘之间的 IO 是以页为单位来进行的，在 Linux 系统上1页的大小为 4K。可以用以下命令查看系统默认的页面大小：

$ /usr/bin/time -v date

...

Page size (bytes): 4096

...

**缺页中断**

Linux 利用虚拟内存极大的扩展了程序地址空间，使得原来物理内存不能容下的程序也可以通过内存和硬盘之间的不断交换（把暂时不用的内存页交换到硬盘，把需要的内 存页从硬盘读到内存）来赢得更多的内存，看起来就像物理内存被扩大了一样。事实上这个过程对程序是完全透明的，程序完全不用理会自己哪一部分、什么时候被 交换进内存，一切都有内核的虚拟内存管理来完成。当程序启动的时候，Linux 内核首先检查 CPU 的缓存和物理内存，如果数据已经在内存里就忽略，如果数据不在内存里就引起一个缺页中断（Page Fault），然后从硬盘读取缺页，并把缺页缓存到物理内存里。缺页中断可分为主缺页中断（Major Page Fault）和次缺页中断（Minor Page Fault），要从磁盘读取数据而产生的中断是主缺页中断；数据已经被读入内存并被缓存起来，从内存缓存区中而不是直接从硬盘中读取数据而产生的中断是次 缺页中断。

上面的内存缓存区起到了预读硬盘的作用，内核先在物理内存里寻找缺页，没有的话产生次缺页中断从内存缓存里找，如果还没有发现的话就从硬盘读取。很 显然，把多余的内存拿出来做成内存缓存区提高了访问速度，这里还有一个命中率的问题，运气好的话如果每次缺页都能从内存缓存区读取的话将会极大提高性能。 要提高命中率的一个简单方法就是增大内存缓存区面积，缓存区越大预存的页面就越多，命中率也会越高。下面的 time 命令可以用来查看某程序第一次启动的时候产生了多少主缺页中断和次缺页中断：

$ /usr/bin/time -v date

...

Major (requiring I/O) page faults: 1

Minor (reclaiming a frame) page faults: 260

...

**File Buffer Cache**

从上面的内存缓存区（也叫文件缓存区 File Buffer Cache）读取页比从硬盘读取页要快得多，所以 Linux 内核希望能尽可能产生次缺页中断（从文件缓存区读），并且能尽可能避免主缺页中断（从硬盘读），这样随着次缺页中断的增多，文件缓存区也逐步增大，直到系 统只有少量可用物理内存的时候 Linux 才开始释放一些不用的页。我们运行 Linux 一段时间后会发现虽然系统上运行的程序不多，但是可用内存总是很少，这样给大家造成了 Linux 对内存管理很低效的假象，事实上 Linux 把那些暂时不用的物理内存高效的利用起来做预存（内存缓存区）呢。下面打印的是 VPSee 的一台 Sun 服务器上的物理内存和文件缓存区的情况：

$ cat /proc/meminfo

MemTotal: 8182776 kB

MemFree: 3053808 kB

Buffers: 342704 kB

Cached: 3972748 kB

这台服务器总共有 8GB 物理内存（MemTotal），3GB 左右可用内存（MemFree），343MB 左右用来做磁盘缓存（Buffers），4GB 左右用来做文件缓存区（Cached），可见 Linux 真的用了很多物理内存做 Cache，而且这个缓存区还可以不断增长。

**页面类型**

Linux 中内存页面有三种类型：

* Read pages，只读页（或代码页），那些通过主缺页中断从硬盘读取的页面，包括不能修改的静态文件、可执行文件、库文件等。当内核需要它们的时候把它们读到 内存中，当内存不足的时候，内核就释放它们到空闲列表，当程序再次需要它们的时候需要通过缺页中断再次读到内存。
* Dirty pages，脏页，指那些在内存中被修改过的数据页，比如文本文件等。这些文件由 pdflush 负责同步到硬盘，内存不足的时候由 kswapd 和 pdflush 把数据写回硬盘并释放内存。
* Anonymous pages，匿名页，那些属于某个进程但是又和任何文件无关联，不能被同步到硬盘上，内存不足的时候由 kswapd 负责将它们写到交换分区并释放内存。

**IO’s Per Second（IOPS）**

每次磁盘 IO 请求都需要一定的时间，和访问内存比起来这个等待时间简直难以忍受。在一台 2001 年的典型 1GHz PC 上，磁盘随机访问一个 word 需要 8,000,000 nanosec = 8 millisec，顺序访问一个 word 需要 200 nanosec；而从内存访问一个 word 只需要 10 nanosec.（数据来自：Teach Yourself Programming in Ten Years）这个硬盘可以提供 125 次 IOPS（1000 ms / 8 ms）。

**顺序 IO 和 随机 IO**

IO 可分为顺序 IO 和 随机 IO 两种，性能监测前需要弄清楚系统偏向顺序 IO 的应用还是随机 IO 应用。顺序 IO 是指同时顺序请求大量数据，比如数据库执行大量的查询、流媒体服务等，顺序 IO 可以同时很快的移动大量数据。可以这样来评估 IOPS 的性能，用每秒读写 IO 字节数除以每秒读写 IOPS 数，rkB/s 除以 r/s，wkB/s 除以 w/s. 下面显示的是连续2秒的 IO 情况，可见每次 IO 写的数据是增加的（45060.00 / 99.00 = 455.15 KB per IO，54272.00 / 112.00 = 484.57 KB per IO）。相对随机 IO 而言，顺序 IO 更应该重视每次 IO 的吞吐能力（KB per IO）：

$ iostat -kx 1

avg-cpu: %user %nice %system %iowait %steal %idle

0.00 0.00 2.50 25.25 0.00 72.25

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util

sdb 24.00 19995.00 29.00 99.00 4228.00 45060.00 770.12 45.01 539.65 7.80 99.80

avg-cpu: %user %nice %system %iowait %steal %idle

0.00 0.00 1.00 30.67 0.00 68.33

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util

sdb 3.00 12235.00 3.00 112.00 768.00 54272.00 957.22 144.85 576.44 8.70 100.10

随机 IO 是指随机请求数据，其 IO 速度不依赖于数据的大小和排列，依赖于磁盘的每秒能 IO 的次数，比如 Web 服务、Mail 服务等每次请求的数据都很小，随机 IO 每秒同时会有更多的请求数产生，所以磁盘的每秒能 IO 多少次是关键。

$ iostat -kx 1

avg-cpu: %user %nice %system %iowait %steal %idle

1.75 0.00 0.75 0.25 0.00 97.26

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util

sdb 0.00 52.00 0.00 57.00 0.00 436.00 15.30 0.03 0.54 0.23 1.30

avg-cpu: %user %nice %system %iowait %steal %idle

1.75 0.00 0.75 0.25 0.00 97.24

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util

sdb 0.00 56.44 0.00 66.34 0.00 491.09 14.81 0.04 0.54 0.19 1.29

按照上面的公式得出：436.00 / 57.00 = 7.65 KB per IO，491.09 / 66.34 = 7.40 KB per IO. 与顺序 IO 比较发现，随机 IO 的 KB per IO 小到可以忽略不计，可见对于随机 IO 而言重要的是每秒能 IOPS 的次数，而不是每次 IO 的吞吐能力（KB per IO）。

**SWAP**

当系统没有足够物理内存来应付所有请求的时候就会用到 swap 设备，swap 设备可以是一个文件，也可以是一个磁盘分区。不过要小心的是，使用 swap 的代价非常大。如果系统没有物理内存可用，就会频繁 swapping，如果 swap 设备和程序正要访问的数据在同一个文件系统上，那会碰到严重的 IO 问题，最终导致整个系统迟缓，甚至崩溃。swap 设备和内存之间的 swapping 状况是判断 Linux 系统性能的重要参考，我们已经有很多工具可以用来监测 swap 和 swapping 情况，比如：top、cat /proc/meminfo、vmstat 等：

$ cat /proc/meminfo

MemTotal: 8182776 kB

MemFree: 2125476 kB

Buffers: 347952 kB

Cached: 4892024 kB

SwapCached: 112 kB

...

SwapTotal: 4096564 kB

SwapFree: 4096424 kB

...

$ vmstat 1

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------

r b swpd free buff cache si so bi bo in cs us sy id wa st

1 2 260008 2188 144 6824 11824 2584 12664 2584 1347 1174 14 0 0 86 0

2 1 262140 2964 128 5852 24912 17304 24952 17304 4737 2341 86 10 0 0 4

# Linux性能监测：网络篇

网络的监测是所有 Linux 子系统里面最复杂的，有太多的因素在里面，比如：延迟、阻塞、冲突、丢包等，更糟的是与 Linux 主机相连的路由器、交换机、无线信号都会影响到整体网络并且很难判断是因为 Linux 网络子系统的问题还是别的设备的问题，增加了监测和判断的复杂度。现在我们使用的所有网卡都称为自适应网卡，意思是说能根据网络上的不同网络设备导致的不 同网络速度和工作模式进行自动调整。我们可以通过 ethtool 工具来查看网卡的配置和工作模式：

# /sbin/ethtool eth0

Settings for eth0:

Supported ports: [ TP ]

Supported link modes: 10baseT/Half 10baseT/Full

100baseT/Half 100baseT/Full

1000baseT/Half 1000baseT/Full

Supports auto-negotiation: Yes

Advertised link modes: 10baseT/Half 10baseT/Full

100baseT/Half 100baseT/Full

1000baseT/Half 1000baseT/Full

Advertised auto-negotiation: Yes

Speed: 100Mb/s

Duplex: Full

Port: Twisted Pair

PHYAD: 1

Transceiver: internal

Auto-negotiation: on

Supports Wake-on: g

Wake-on: g

Current message level: 0x000000ff (255)

Link detected: yes

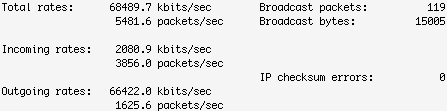
上面给出的例子说明网卡有 10baseT，100baseT 和 1000baseT 三种选择，目前正自适应为 100baseT（Speed: 100Mb/s）。可以通过 ethtool 工具强制网卡工作在 1000baseT 下：

# /sbin/ethtool -s eth0 speed 1000 duplex full autoneg off

## iptraf

两台主机之间有网线（或无线）、路由器、交换机等设备，测试两台主机之间的网络性能的一个办法就是在这两个系统之间互发数据并统计结果，看看吞吐 量、延迟、速率如何。iptraf 就是一个很好的查看本机网络吞吐量的好工具，支持文字图形界面，很直观。下面图片显示在 100 mbps 速率的网络下这个 Linux 系统的发送传输率有点慢，Outgoing rates 只有 66 mbps.

# iptraf -d eth0



## netperf

netperf 运行在 client/server 模式下，比 iptraf 能更多样化的测试终端的吞吐量。先在服务器端启动 netserver：

# netserver

Starting netserver at port 12865

Starting netserver at hostname 0.0.0.0 port 12865 and family AF\_UNSPEC

然后在客户端测试服务器，执行一次持续10秒的 TCP 测试：

# netperf -H 172.16.38.36 -l 10

TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF\_INET to 172.16.38.36 (172.16.38.36) port 0 AF\_INET

Recv Send Send

Socket Socket Message Elapsed

Size Size Size Time Throughput

bytes bytes bytes secs. 10^6bits/sec

87380 16384 16384 10.32 93.68

从以上输出可以看出，网络吞吐量在 94mbps 左右，对于 100mbps 的网络来说这个性能算的上很不错。上面的测试是在服务器和客户端位于同一个局域网，并且局域网是有线网的情况，你也可以试试不同结构、不同速率的网络，比 如：网络之间中间多几个路由器、客户端在 wi-fi、VPN 等情况。

netperf 还可以通过建立一个 TCP 连接并顺序地发送数据包来测试每秒有多少 TCP 请求和响应。下面的输出显示在 TCP requests 使用 2K 大小，responses 使用 32K 的情况下处理速率为每秒243：

# netperf -t TCP\_RR -H 172.16.38.36 -l 10 -- -r 2048,32768

TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF\_INET to 172.16.38.36 (172.16.38.36) port 0 AF\_INET

Local /Remote

Socket Size Request Resp. Elapsed Trans.

Send Recv Size Size Time Rate

bytes Bytes bytes bytes secs. per sec

16384 87380 2048 32768 10.00 243.03

16384 87380

## iperf

iperf 和 netperf 运行方式类似，也是 server/client 模式，先在服务器端启动 iperf：

# iperf -s -D

------------------------------------------------------------

Server listening on TCP port 5001

TCP window size: 85.3 KByte (default)

------------------------------------------------------------

Running Iperf Server as a daemon

The Iperf daemon process ID : 5695

然后在客户端对服务器进行测试，客户端先连接到服务器端（172.16.38.36），并在30秒内每隔5秒对服务器和客户端之间的网络进行一次带宽测试和采样：

# iperf -c 172.16.38.36 -t 30 -i 5

------------------------------------------------------------

Client connecting to 172.16.38.36, TCP port 5001

TCP window size: 16.0 KByte (default)

------------------------------------------------------------

[ 3] local 172.16.39.100 port 49515 connected with 172.16.38.36 port 5001

[ ID] Interval Transfer Bandwidth

[ 3] 0.0- 5.0 sec 58.8 MBytes 98.6 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 5.0-10.0 sec 55.0 MBytes 92.3 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 10.0-15.0 sec 55.1 MBytes 92.4 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 15.0-20.0 sec 55.9 MBytes 93.8 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 20.0-25.0 sec 55.4 MBytes 92.9 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 25.0-30.0 sec 55.3 MBytes 92.8 Mbits/sec

[ ID] Interval Transfer Bandwidth

[ 3] 0.0-30.0 sec 335 MBytes 93.7 Mbits/sec

## tcpdump 和 tcptrace

tcmdump 和 tcptrace 提供了一种更细致的分析方法，先用 tcpdump 按要求捕获数据包把结果输出到某一文件，然后再用 tcptrace 分析其文件格式。这个工具组合可以提供一些难以用其他工具发现的信息：

# /usr/sbin/tcpdump -w network.dmp

tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes

511942 packets captured

511942 packets received by filter

0 packets dropped by kernel

# tcptrace network.dmp

1 arg remaining, starting with 'network.dmp'

Ostermann's tcptrace -- version 6.6.7 -- Thu Nov 4, 2004

511677 packets seen, 511487 TCP packets traced

elapsed wallclock time: 0:00:00.510291, 1002714 pkts/sec analyzed

trace file elapsed time: 0:02:35.836372

TCP connection info:

1: zaber:54581 - boulder:111 (a2b) 6> 5< (complete)

2: zaber:833 - boulder:32774 (c2d) 6> 5< (complete)

3: zaber:pcanywherestat - 172.16.39.5:53086 (e2f) 2> 3<

4: zaber:716 - boulder:2049 (g2h) 347> 257<

5: 172.16.39.100:58029 - zaber:12865 (i2j) 7> 5< (complete)

6: 172.16.39.100:47592 - zaber:36814 (k2l) 255380> 255378< (reset)

7: breakpoint:45510 - zaber:7012 (m2n) 9> 5< (complete)

8: zaber:35813 - boulder:111 (o2p) 6> 5< (complete)

9: zaber:837 - boulder:32774 (q2r) 6> 5< (complete)

10: breakpoint:45511 - zaber:7012 (s2t) 9> 5< (complete)

11: zaber:59362 - boulder:111 (u2v) 6> 5< (complete)

12: zaber:841 - boulder:32774 (w2x) 6> 5< (complete)

13: breakpoint:45512 - zaber:7012 (y2z) 9> 5< (complete)

tcptrace 功能很强大，还可以通过过滤和布尔表达式来找出有问题的连接，比如，找出转播大于100 segments 的连接：

# tcptrace -f'rexmit\_segs>100' network.dmp

如果发现连接 ＃10 有问题，可以查看关于这个连接的其他信息：

# tcptrace -o10 network.dmp

下面的命令使用 tcptrace 的 slice 模式，程序自动在当前目录创建了一个 slice.dat 文件，这个文件包含了每隔15秒的转播信息:

# tcptrace -xslice network.dmp

# cat slice.dat

date segs bytes rexsegs rexbytes new active

--------------- -------- -------- -------- -------- -------- --------

16:58:50.244708 85055 4513418 0 0 6 6

16:59:05.244708 110921 5882896 0 0 0 2

16:59:20.244708 126107 6697827 0 0 1 3

16:59:35.244708 151719 8043597 0 0 0 2

16:59:50.244708 37296 1980557 0 0 0 3

17:00:05.244708 67 8828 0 0 2 3

17:00:20.244708 149 22053 0 0 1 2

17:00:35.244708 30 4080 0 0 0 1

17:00:50.244708 39 5688 0 0 0 1

17:01:05.244708 67 8828 0 0 2 3

17:01:11.081080 37 4121 0 0 1 3