LINUXCON

Linux System and Performance Monitoring

Darren Hoch

Director of Professional Services – StrongMail Systems, Inc.

Linux Performance Monitoring

PUBLISHED BY:
Darren Hoch
StrongMail Systems
1300 Island Drive
Suite 200
Redwood City, CA
94065
dhoch@strongmail.com

http://www.strongmailsystems.com

Table of Contents

1.0 Performance Monitoring Introduction

Performance tuning is the process of finding bottlenecks in a system and tuning the operating system to eliminate these bottlenecks. Many administrators believe that performance tuning can be a “cook book” approach, which is to say that setting some parameters in the kernel will simply solve a problem. This is not the case. Performance tuning is about achieving balance between the different sub-systems of an OS. These sub-systems include:

- **CPU**
- **Memory**
- **IO**
- **Network**

These sub-systems are all highly dependent on each other. Any one of them with high utilization can easily cause problems in the other. For example:

- **large amounts of page-in IO requests can fill the memory queues**
- **full gigabit throughput on an Ethernet controller may consume a CPU**
- **a CPU may be consumed attempting to maintain free memory queues**
- **a large number of disk write requests from memory may consume a CPU and IO channels**

In order to apply changes to tune a system, the location of the bottleneck must be located. Although one sub-system appears to be causing the problems, it may be as a result of overload on another sub-system.

1.1 Determining Application Type

In order to understand where to start looking for tuning bottlenecks, it is first important to understand the behavior of the system under analysis. The application stack of any system is often broken down into two types:

- **IO Bound – An IO bound application requires heavy use of memory and the underlying storage system. This is due to the fact that an IO bound application is processing (in memory) large amounts of data. An IO bound application does not require much of the CPU or network (unless the storage system is on a network). IO bound applications use CPU resources to make IO requests and then often go into a sleep state. Database applications are often considered IO bound applications.**
- **CPU Bound – A CPU bound application requires heavy use of the CPU. CPU bound applications require the CPU for batch processing and/or mathematical calculations. High volume web servers, mail servers, and any kind of rendering server are often considered CPU bound applications.**

1.2 Determining Baseline Statistics

System utilization is contingent on administrator expectations and system specifications. The only way to understand if a system is having performance issues is to understand what is expected of the system. What kind of performance should be expected and what do those numbers look like? The only way to establish this is to create a baseline. Statistics must be available for a system under acceptable performance so it can be compared later against unacceptable performance.

In the following example, a baseline snapshot of system performance is compared against a snapshot of the system under heavy utilization.

```
# vmstat 1
procs                      memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 1  0 138592  17932 126272 214244    0    0     1    18  109    19  2  1  1 96
 0  0 138592  17932 126272 214244    0    0     0     0  105    46  0  1  0 99
 0  0 138592  17932 126272 214244    0    0     0     0  198    62 40 14  0 45
 0  0 138592  17932 126272 214244    0    0     0     0  117    49  0  0  0 100
 0  0 138592  17924 126272 214244    0    0     0   176  220   938  3  4 13 80
 0  0 138592  17924 126272 214244    0    0     0     0  358  1522  8 17  0 75
 1  0 138592  17924 126272 214244    0    0     0     0  368  1447  4 24  0 72
 0  0 138592  17924 126272 214244    0    0     0     0  352  1277  9 12  0 79
```

```
# vmstat 1
procs                      memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 2  0 145940  17752 118600 215592    0    1     1    18  109    19  2  1  1 96
 2  0 145940  15856 118604 215652    0    0     0   468  789   108 86 14  0  0
 3  0 146208  13884 118600 214640    0  360     0   360  498    71 91  9  0  0
 2  0 146388  13764 118600 213788    0  340     0   340  672    41 87 13  0  0
 2  0 147092  13788 118600 212452    0  740     0  1324  620    61 92  8  0  0
 2  0 147360  13848 118600 211580    0  720     0   720  690    41 96  4  0  0
 2  0 147912  13744 118192 210592    0  720     0   720  605    44 95  5  0  0
 2  0 148452  13900 118192 209260    0  372     0   372  639    45 81 19  0  0
 2  0 149132  13692 117824 208412    0  372     0   372  457    47 90 10  0  0
```

Just by looking at the numbers in the last column (`id`) which represent idle time, we can see that under baseline conditions, the CPU is idle for 79% - 100% of the time. In the second output, we can see that the system is 100% utilized and not idle. What needs to be determined is whether or not the system at CPU utilization is managing.

2.0 Installing Monitoring Tools

Most *nix systems ship with a series of standard monitoring commands. These monitoring commands have been a part of *nix since its inception. Linux provides these monitoring tools as part of the base installation or add-ons. Ultimately, there are packages available for most distributions with these tools. Although there are multiple open source and 3rd party monitoring tools, the goal of this paper is to use tools included with a Linux distribution.

This paper describes how to monitor performance using the following tools.

Figure 1: Performance Monitoring Tools

Tool	Description	Base	Repository
`vmstat`	all purpose performance tool	yes	yes
`mpstat`	provides statistics per CPU	no	yes
`sar`	all purpose performance monitoring tool	no	yes
`iostat`	provides disk statistics	no	yes
`netstat`	provides network statistics	yes	yes
`dstat`	monitoring statistics aggregator	no	in most distributions
`iptraf`	traffic monitoring dashboard	no	yes
`netperf`	Network bandwidth tool	no	In some distributions
`ethtool`	reports on Ethernet interface configuration	yes	yes
`iperf`	Network bandwidth tool	no	yes
`tcptrace`	Packet analysis tool	no	yes
`iotop`	Displays IO per process	no	yes

3.0 Introducing the CPU

The utilization of a CPU is largely dependent on what resource is attempting to access it. The kernel has a scheduler that is responsible for scheduling two kinds of resources: threads (single or multi) and interrupts. The scheduler gives different priorities to the different resources. The following list outlines the priorities from highest to lowest:

- **Interrupts – Devices tell the kernel that they are done processing. For example, a NIC delivers a packet or a hard drive provides an IO request**
- **Kernel (System) Processes – All kernel processing is handled at this level of priority.**
- **User Processes – This space is often referred to as "userland". All software applications run in the user space. This space has the lowest priority in the kernel scheduling mechanism.**

In order to understand how the kernel manages these different resources, a few key concepts need to be introduced. The following sections introduce context switches, run queues, and utilization.

3.1 Context Switches

Most modern processors can only run one process (single threaded) or thread at time. The n-way Hyper threaded processors have the ability to run n threads at a time. Still, the Linux kernel views each processor core on a dual core chip as an independent processor. For example, a system with one dual core processor is reported as two individual processors by the Linux kernel.

A standard Linux kernel can run anywhere from 50 to 50,000 process threads at once. With only one CPU, the kernel has to schedule and balance these process threads. Each thread has an allotted time quantum to spend on the processor. Once a thread has either passed the time quantum or has been preempted by something with a higher priority (a hardware interrupt, for example), that thread is place back into a queue while the higher priority thread is placed on the processor. This switching of threads is referred to as a context switch.

Every time the kernel conducts a context switch, resources are devoted to moving that thread off of the CPU registers and into a queue. The higher the volume of context switches on a system, the more work the kernel has to do in order to manage the scheduling of processes.

3.2 The Run Queue

Each CPU maintains a run queue of threads. Ideally, the scheduler should be constantly running and executing threads. Process threads are either in a sleep state (blocked and waiting on IO) or they are runnable. If the CPU sub-system is heavily utilized, then it is possible that the kernel scheduler can't keep up with the demand of the system. As a result, runnable processes start to fill up a run queue. The larger the run queue, the longer it will take for process threads to execute.

A very popular term called "load" is often used to describe the state of the run queue. The system load is a combination of the amount of process threads currently executing along with the amount of threads in the CPU run queue. If two threads were executing on a dual core system and 4 were in the run queue, then the load would be 6. Utilities such as top report load averages over the course of 1, 5, and 15 minutes.

3.3 CPU Utilization

CPU utilization is defined as the percentage of usage of a CPU. How a CPU is utilized is an important metric for measuring system. Most performance monitoring tools categorize CPU utilization into the following categories:

- **User Time – The percentage of time a CPU spends executing process threads in the user space.**
- **System Time – The percentage of time the CPU spends executing kernel threads and interrupts.**
- **Wait IO – The percentage of time a CPU spends idle because ALL process threads are blocked waiting for IO requests to complete.**
- **Idle – The percentage of time a processor spends in a completely idle state.**

4.0 CPU Performance Monitoring

Understanding how well a CPU is performing is a matter of interpreting run queue, utilization, and context switching performance. As mentioned earlier, performance is all relative to baseline statistics. There are, however, some general performance expectations on any system. These expectations include:

- **Run Queues – A run queue should have no more than 1-3 threads queued per processor. For example, a dual processor system should not have more than 6 threads in the run queue.**
- **CPU Utilization – If a CPU is fully utilized, then the following balance of utilization should be achieved.**
- **65% – 70% User Time**
- **30% - 35% System Time**
- **0% - 5% Idle Time**
- **Context Switches – The amount of context switches is directly relevant to CPU utilization. A high amount of context switching is acceptable if CPU utilization stays within the previously mentioned balance**

There are many tools that are available for Linux that measure these statistics. The first two tools examined will be `vmstat` and top.

4.1 Using the vmstat Utility

The `vmstat` utility provides a good low-overhead view of system performance. Because `vmstat` is such a low-overhead tool, it is practical to keep it running on a console even under a very heavily loaded server were you need to monitor the health of a system at a glance. The utility runs in two modes: average and sample mode. The sample mode will measure statistics over a specified interval. This mode is the most useful when understanding performance under a sustained load. The following example demonstrates `vmstat` running at 1 second intervals.

```
# vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 0  0 104300  16800  95328  72200    0    0     5    26    7    14  4  1 95  0
 0  0 104300  16800  95328  72200    0    0     0    24 1021    64  1  1 98  0
 0  0 104300  16800  95328  72200    0    0     0     0 1009    59  1  1 98  0
```

The relevant fields in the output are as follows:

Table 1: The vmstat CPU statistics

Field	Description
`r`	The amount of threads in the run queue. These are threads that are runnable, but the CPU is not available to execute them.
`b`	This is the number of processes blocked and waiting on IO requests to finish.
`in`	This is the number of interrupts being processed.
`cs`	This is the number of context switches currently happening on the system.
`us`	This is the percentage of user CPU utilization.
`sys`	This is the percentage of kernel and interrupts utilization.
`wa`	This is the percentage of idle processor time due to the fact that ALL runnable threads are blocked waiting on IO.
`id`	This is the percentage of time that the CPU is completely idle.

4.2 Case Study: Sustained CPU Utilization

In the next example, the system is completely utilized.

```
# vmstat 1
procs                      memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 3  0 206564  15092  80336 176080    0    0     0     0  718    26 81 19  0  0
 2  0 206564  14772  80336 176120    0    0     0     0  758    23 96  4  0  0
 1  0 206564  14208  80336 176136    0    0     0     0  820    20 96  4  0  0
 1  0 206956  13884  79180 175964    0  412     0  2680 1008    80 93  7  0  0
 2  0 207348  14448  78800 175576    0  412     0   412  763    70 84 16  0  0
 2  0 207348  15756  78800 175424    0    0     0     0  874    25 89 11  0  0
 1  0 207348  16368  78800 175596    0    0     0     0  940    24 86 14  0  0
 1  0 207348  16600  78800 175604    0    0     0     0  929    27 95  3  0  2
 3  0 207348  16976  78548 175876    0    0     0  2508  969    35 93  7  0  0
 4  0 207348  16216  78548 175704    0    0     0     0  874    36 93  6  0  1
 4  0 207348  16424  78548 175776    0    0     0     0  850    26 77 23  0  0
 2  0 207348  17496  78556 175840    0    0     0     0  736    23 83 17  0  0
 0  0 207348  17680  78556 175868    0    0     0     0  861    21 91  8  0  1
```

The following observations are made from the output:

- **There are a high amount of interrupts (`in`) and a low amount of context switches. It appears that a single process is making requests to hardware devices.**
- **To further prove the presence of a single application, the user (`us`) time is constantly at 85% and above. Along with the low amount of context switches, the process comes on the processor and stays on the processor.**
- **The run queue is just about at the limits of acceptable performance. On a couple occasions, it goes beyond acceptable limits.**

4.3 Case Study: Overloaded Scheduler

In the following example, the kernel scheduler is saturated with context switches.

```
# vmstat 1
procs                      memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy wa id
 2  1 207740  98476  81344 180972    0    0  2496     0  900  2883  4 12 57 27
 0  1 207740  96448  83304 180984    0    0  1968   328  810  2559  8  9 83  0
 0  1 207740  94404  85348 180984    0    0  2044     0  829  2879  9  6 78  7
 0  1 207740  92576  87176 180984    0    0  1828     0  689  2088  3  9 78 10
 2  0 207740  91300  88452 180984    0    0  1276     0  565  2182  7  6 83  4
 3  1 207740  90124  89628 180984    0    0  1176     0  551  2219  2  7 91  0
 4  2 207740  89240  90512 180984    0    0   880   520  443   907 22 10 67  0
 5  3 207740  88056  91680 180984    0    0  1168     0  628  1248 12 11 77  0
 4  2 207740  86852  92880 180984    0    0  1200     0  654  1505  6  7 87  0
 6  1 207740  85736  93996 180984    0    0  1116     0  526  1512  5 10 85  0
 0  1 207740  84844  94888 180984    0    0   892     0  438  1556  6  4 90  0
```

The following conclusions can be drawn from the output:

- **The amount of context switches is higher than interrupts, suggesting that the kernel has to spend a considerable amount of time context switching threads.**
- **The high volume of context switches is causing an unhealthy balance of CPU utilization. This is evident by the fact that the wait on IO percentage is extremely high and the user percentage is extremely low.**
- **Because the CPU is block waiting for I/O, the run queue starts to fill and the amount of threads blocked waiting on I/O also fills.**

4.4 Using the mpstat Utility

If your system has multiple processor cores, you can use the `mpstat` command to monitor each individual core. The Linux kernel treats a dual core processor as 2 CPU's. So, a dual processor system with dual cores will report 4 CPUs available. The `mpstat` command provides the same CPU utilization statistics as `vmstat`, but `mpstat` breaks the statistics out on a per processor basis.

```
# mpstat –P ALL 1
Linux 2.4.21-20.ELsmp (localhost.localdomain)   05/23/2006

05:17:31 PM  CPU   %user   %nice %system   %idle    intr/s
05:17:32 PM  all    0.00    0.00    3.19   96.53     13.27
05:17:32 PM    0    0.00    0.00    0.00  100.00      0.00
05:17:32 PM    1    1.12    0.00   12.73   86.15     13.27
05:17:32 PM    2    0.00    0.00    0.00  100.00      0.00
05:17:32 PM    3    0.00    0.00    0.00  100.00      0.00
```

4.5 Case Study: Underutilized Process Load

In the following case study, a 4 CPU cores are available. There are two CPU intensive processes running that fully utilize 2 of the cores (CPU 0 and 1). The third core is processing all kernel and other system functions (CPU 3). The fourth core is sitting idle (CPU 2).

The `top` command shows that there are 3 processes consuming almost an entire CPU core:

```
# top -d 1
top - 23:08:53 up  8:34,  3 users,  load average: 0.91, 0.37, 0.13
Tasks: 190 total,   4 running, 186 sleeping,   0 stopped,   0 zombie
Cpu(s): 75.2% us,  0.2% sy,  0.0% ni, 24.5% id,  0.0% wa,  0.0% hi,  0.0%
si
Mem:   2074736k total,   448684k used,  1626052k free,    73756k buffers
Swap:  4192956k total,        0k used,  4192956k free,   259044k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
15957 nobody    25   0  2776  280  224 R  100  20.5  0:25.48 php
15959 mysql     25   0  2256  280  224 R  100  38.2  0:17.78 mysqld
15960 apache    25   0  2416  280  224 R  100  15.7  0:11.20 httpd
15901 root      16   0  2780 1092  800 R    1  0.1   0:01.59 top
    1 root      16   0  1780  660  572 S    0  0.0   0:00.64 init
```



```
# mpstat –P ALL 1
Linux 2.4.21-20.ELsmp (localhost.localdomain)   05/23/2006

05:17:31 PM  CPU   %user   %nice %system   %idle    intr/s
05:17:32 PM  all   81.52    0.00   18.48   21.17    130.58
05:17:32 PM    0   83.67    0.00   17.35    0.00    115.31
05:17:32 PM    1   80.61    0.00   19.39    0.00     13.27
05:17:32 PM    2    0.00    0.00   16.33   84.66      2.01
05:17:32 PM    3   79.59    0.00   21.43    0.00      0.00

05:17:32 PM  CPU   %user   %nice %system   %idle    intr/s
05:17:33 PM  all   85.86    0.00   14.14   25.00    116.49
05:17:33 PM    0   88.66    0.00   12.37    0.00    116.49
05:17:33 PM    1   80.41    0.00   19.59    0.00      0.00
05:17:33 PM    2    0.00    0.00    0.00  100.00      0.00
05:17:33 PM    3   83.51    0.00   16.49    0.00      0.00

05:17:33 PM  CPU   %user   %nice %system   %idle    intr/s
05:17:34 PM  all   82.74    0.00   17.26   25.00    115.31
05:17:34 PM    0   85.71    0.00   13.27    0.00    115.31
05:17:34 PM    1   78.57    0.00   21.43    0.00      0.00
05:17:34 PM    2    0.00    0.00    0.00  100.00      0.00
05:17:34 PM    3   92.86    0.00    9.18    0.00      0.00

05:17:34 PM  CPU   %user   %nice %system   %idle    intr/s
05:17:35 PM  all   87.50    0.00   12.50   25.00    115.31
05:17:35 PM    0   91.84    0.00    8.16    0.00    114.29
05:17:35 PM    1   90.82    0.00   10.20    0.00      1.02
05:17:35 PM    2    0.00    0.00    0.00  100.00      0.00
05:17:35 PM    3   81.63    0.00   15.31    0.00      0.00
```

You can determine which process is taking up which CPU by running the ps command again and monitoring the PSR column.

```
# while :; do  ps -eo pid,ni,pri,pcpu,psr,comm | grep 'mysqld'; sleep 1;
done
  PID  NI PRI %CPU PSR COMMAND
15775   0  15 86.0   3 mysqld
  PID  NI PRI %CPU PSR COMMAND
15775   0  14 94.0   3 mysqld
  PID  NI PRI %CPU PSR COMMAND
15775   0  14 96.6   3 mysqld
  PID  NI PRI %CPU PSR COMMAND
15775   0  14 98.0   3 mysqld
  PID  NI PRI %CPU PSR COMMAND
15775   0  14 98.8   3 mysqld
  PID  NI PRI %CPU PSR COMMAND
15775   0  14 99.3   3 mysqld
```


4.6 Conclusion

Monitoring CPU performance consists of the following actions:

- **Check the system run queue and make sure there are no more than 3 runnable threads per processor**
- **Make sure the CPU utilization is split between 70/30 between user and system**
- **When the CPU spends more time in system mode, it is more than likely overloaded and trying to reschedule priorities**
- **Running CPU bound process always get penalized while I/O process are rewarded**

5.0 Introducing Virtual Memory

Virtual memory uses a disk as an extension of RAM so that the effective size of usable memory grows correspondingly. The kernel will write the contents of a currently unused block of memory to the hard disk so that the memory can be used for another purpose. When the original contents are needed again, they are read back into memory. This is all made completely transparent to the user; programs running under Linux only see the larger amount of memory available and don't notice that parts of them reside on the disk from time to time. Of course, reading and writing the hard disk is slower (on the order of a thousand times slower) than using real memory, so the programs don't run as fast. The part of the hard disk that is used as virtual memory is called the swap space.

5.1 Virtual Memory Pages

Virtual memory is divided into pages. Each virtual memory page on the X86 architecture is 4KB. When the kernel writes memory to and from disk, it writes memory in pages. The kernel writes memory pages to both the swap device and the file system.

5.2 Kernel Memory Paging

Memory paging is a normal activity not to be confused with memory swapping. Memory paging is the process of synching memory back to disk at normal intervals. Over time, applications will grow to consume all of memory. At some point, the kernel must scan memory and reclaim unused pages to be allocated to other applications.

5.3 The Page Frame Reclaim Algorithm (PFRA)

The PFRA is responsible for freeing memory. The PFRA selects which memory pages to free by page type. Page types are listed below:

- **Unreclaimable – locked, kernel, reserved pages**
- **Swappable – anonymous memory pages**
- **Syncable – pages backed by a disk file**
- **Discardable – static pages, discarded pages**

All but the “unreclaimable” pages may be reclaimed by the PFRA.

There are two main functions in the PFRA. These include the kswapd kernel thread and the “Low On Memory Reclaiming” function.

5.4 `kswapd`

The `kswapd` daemon is responsible for ensuring that memory stays free. It monitors the `pages_high` and `pages_low` watermarks in the kernel. If the amount of free memory is below `pages_low,` the `kswapd` process starts a scan to attempt to free 32 pages at a time. It repeats this process until the amount of free memory is above the `pages_high` watermark.

The kswapd thread performs the following actions:

- **If the page is unmodified, it places the page on the free list.**
- **If the page is modified and backed by a filesystem, it writes the contents of the page to disk.**
- **If the page is modified and not backed up by any filesystem (anonymous), it writes the contents of the page to the swap device.**

```
# ps -ef | grep kswapd
root        30     1  0 23:01 ?        00:00:00 [kswapd0]
```

5.5 Kernel Paging with pdflush

The pdflush daemon is responsible for synchronizing any pages associated with a file on a filesystem back to disk. In other words, when a file is modified in memory, the pdflush daemon writes it back to disk.

```
# ps -ef | grep pdflush
root        28     3  0 23:01 ?        00:00:00 [pdflush]
root        29     3  0 23:01 ?        00:00:00 [pdflush]
```

The pdflush daemon starts synchronizing dirty pages back to the filesystem when 10% of the pages in memory are dirty. This is due to a kernel tuning parameter called vm.dirty_background_ratio.

```
# sysctl -n vm.dirty_background_ratio
10
```

The pdflush daemon works independently of the PFRA under most circumstances. When the kernel invokes the LMR algorithm, the LMR specifically forces pdflush to flush dirty pages in addition to other page freeing routines.

Under intense memory pressure in the 2.4 kernel, the system would experience swap thrashing. This would occur when the PFRA would steal a page that an active process was trying to use. As a result, the process would have to reclaim that page only for it to be stolen again, creating a thrashing condition. This was fixed in kernel 2.6 with the "Swap Token", which prevents the PFRA from constantly stealing the same page from a process.

5.6 Case Study: Large Inbound I/O

The `vmstat` utility reports on virtual memory usage in addition to CPU usage. The following fields in the `vmstat` output are relevant to virtual memory:

Table 2: The vmstat Memory Statistics

Field	Description
swapd	The amount of virtual memory in KB currently in use. As free memory reaches low thresholds, more data is paged to the swap device.
free	The amount of physical RAM in kilobytes currently available to running applications.
buff	The amount of physical memory in kilobytes in the buffer cache as a result of read() and write() operations.
cache	The amount of physical memory in kilobytes mapped into process address space.
so	The amount of data in kilobytes written to the swap disk.
si	The amount of data in kilobytes written from the swap disk back into RAM.
bo	The amount of disk blocks paged out from the RAM to the filesystem or swap device.
bi	The amount of disk blocks paged into RAM from the filesystem or swap device.

The following `vmstat` output demonstrates heavy utilization of virtual memory during an I/O application spike.

```
# vmstat 3
 procs           memory               swap          io     system          cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 3  2 809192 261556  79760  886880  416    0  8244   751  426   863 17  3  6 75
 0  3 809188 194916  79820  952900  307    0 21745  1005 1189  2590 34  6 12 48
 0  3 809188 162212  79840  988920   95    0 12107     0 1801  2633  2  2  3 94
 1  3 809268  88756  79924 1061424  260   28 18377   113 1142  1694  3  5  3 88
 1  2 826284  17608  71240 1144180  100 6140 25839 16380 1528  1179 19  9 12 61
 2  1 854780  17688  34140 1208980    1 9535 25557 30967 1764  2238 43 13 16 28
 0  8 867528  17588  32332 1226392   31 4384 16524 27808 1490  1634 41 10  7 43
 4  2 877372  17596  32372 1227532  213 3281 10912  3337  678   932 33  7  3 57
 1  2 885980  17800  32408 1239160  204 2892 12347 12681 1033   982 40 12  2 46
 5  2 900472  17980  32440 1253884   24 4851 17521  4856  934  1730 48 12 13 26
 1  1 904404  17620  32492 1258928   15 1316  7647 15804  919   978 49  9 17 25
 4  1 911192  17944  32540 1266724   37 2263 12907  3547  834  1421 47 14 20 20
 1  1 919292  17876  31824 1275832    1 2745 16327  2747  617  1421 52 11 23 14
 5  0 925216  17812  25008 1289320   12 1975 12760  3181  772  1254 50 10 21 19
 0  5 932860  17736  21760 1300280    8 2556 15469  3873  825  1258 49 13 24 15
```


The following observations are made from this output:

- **A large amount of disk blocks are paged in (`bi`) from the filesystem. This is evident in the fact that the cache of data in process address spaces (`cache`) grows.**
- **During this period, the amount of free memory (`free`) remains steady at 17MB even though data is paging in from the disk to consume free RAM.**
- **To maintain the free list, `kswapd` steals memory from the read/write buffers (`buff`) and assigns it to the free list. This is evident in the gradual decrease of the buffer cache (`buff`).**
- **The `kswapd` process then writes dirty pages to the swap device (`so`). This is evident in the fact that the amount of virtual memory utilized gradually increases (`swpd`).**

5.7 Conclusion

Virtual memory performance monitoring consists of the following actions:

- **The less major page faults on a system, the better response times achieved as the system is leveraging memory caches over disk caches.**
- **Low amounts of free memory are a good sign that caches are effectively used unless there are sustained writes to the swap device and disk.**
- **If a system reports any sustained activity on the swap device, it means there is a memory shortage on the system.**

6.0 Introducing I/O Monitoring

Disk I/O subsystems are the slowest part of any Linux system. This is due mainly to their distance from the CPU and the fact that disks require the physics to work (rotation and seek). If the time taken to access disk as opposed to memory was converted into minutes and seconds, it is the difference between 7 days and 7 minutes. As a result, it is essential that the Linux kernel minimizes the amount of I/O it generates on a disk. The following subsections describe the different ways the kernel processes data I/O from disk to memory and back.

6.1 Reading and Writing Data - Memory Pages

The Linux kernel breaks disk I/O into pages. The default page size on most Linux systems is 4K. It reads and w rites disk blocks in and out of memory in 4K page sizes. You can check the page size of your system by using the `time` command in verbose mode and searching for the page size:

```
# /usr/bin/time -v date

<snip>

Page size (bytes): 4096

<snip>
```

6.2 Major and Minor Page Faults

Linux, like most UNIX systems, uses a virtual memory layer that maps into physical address space. This mapping is "on demand" in the sense that when a process starts, the kernel only maps that which is required. When an application starts, the kernel searches the CPU caches and then physical memory. If the data does not exist in either, the kernel issues a major page fault (MPF). A MPF is a request to the disk subsystem to retrieve pages off disk and buffer them in RAM.

Once memory pages are mapped into the buffer cache, the kernel will attempt to use these pages resulting in a minor page fault (MnPF). A MnPF saves the kernel time by reusing a page in memory as opposed to placing it back on the disk.

In the following example, the time command is used to demonstrate how many MPF and MnPF occurred when an application started. The first time the application starts, there are many MPFs:


```
# /usr/bin/time -v evolution

<snip>

Major (requiring I/O) page faults: 163
Minor (reclaiming a frame) page faults: 5918

<snip>
```

The second time evolution starts, the kernel does not issue any MPFs because the application is in memory already:

```
# /usr/bin/time -v evolution

<snip>

Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 5581

<snip>
```

6.3 The File Buffer Cache

The file buffer cache is used by the kernel to minimize MPFs and maximize MnPFs. As a system generates I/O over time, this buffer cache will continue to grow as the system will leave these pages in memory until memory gets low and the kernel needs to "free" some of these pages for other uses. The end result is that many system administrators see low amounts of free memory and become concerned when in reality, the system is just making good use of its caches.

The following output is taken from the `/proc/meminfo` file:

```
# cat /proc/meminfo
MemTotal: 2075672 kB
MemFree: 52528 kB
Buffers: 24596 kB
Cached: 1766844 kB

<snip>
```

The system has a total of 2 GB (`MemTotal`) of RAM available on it. There is currently 52 MB of RAM "free" (`MemFree`), 24 MB RAM that is allocated to disk write operations (`Buffers`), and 1.7 GB of pages read from disk in RAM (`Cached`).

The kernel is using these via the MnPF mechanism as opposed to pulling all of these pages in from disk. It is impossible to tell from these statistics whether or not the system is under distress as we only have part of the picture.

6.4 Types of Memory Pages

There are 3 types of memory pages in the Linux kernel. These pages are described below:

- **Read Pages – These are pages of data read in via disk (MPF) that are read only and backed on disk. These pages exist in the Buffer Cache and include static files, binaries, and libraries that do not change. The Kernel will continue to page these into memory as it needs them. If memory becomes short, the kernel will "steal" these pages and put them back on the free list causing an application to have to MPF to bring them back in.**
- **Dirty Pages – These are pages of data that have been modified by the kernel while in memory. These pages need to be synced back to disk at some point using the `pdflush` daemon. In the event of a memory shortage, `kswapd` (along with `pdflush`) will write these pages to disk in order to make more room in memory.**
- **Anonymous Pages – These are pages of data that do belong to a process, but do not have any file or backing store associated with them. They can't be synchronized back to disk. In the event of a memory shortage, `kswapd` writes these to the swap device as temporary storage until more RAM is free ("swapping" pages).**

6.5 Writing Data Pages Back to Disk

Applications themselves may choose to write dirty pages back to disk immediately using the `fsync()` or `sync()` system calls. These system calls issue a direct request to the I/O scheduler. If an application does not invoke these system calls, the `pdflush` kernel daemon runs at periodic intervals and writes pages back to disk.

```
# ps -ef | grep pdflush
root 186 6 0 18:04 ? 00:00:00 [pdflush]
```


7.0 Monitoring I/O

Certain conditions occur on a system that may create I/O bottlenecks. These conditions may be identified by using a standard set of system monitoring tools. These tools include `top, vmstat, iostat,` and `sar`. There are some similarities between the output of these commands, but for the most part, each offers a unique set of output that provides a different aspect on performance. The following subsections describe conditions that cause I/O bottlenecks.

7.1 Calculating IO's Per Second

Every I/O request to a disk takes a certain amount of time. This is due primarily to the fact that a disk must spin and a head must seek. The spinning of a disk is often referred to as "rotational delay" (RD) and the moving of the head as a "disk seek" (DS). The time it takes for each I/O request is calculated by adding DS and RD. A disk's RD is fixed based on the RPM of the drive. An RD is considered half a revolution around a disk. To calculate RD for a 10K RPM drive, perform the following:

1. Divide 10000 RPM by 60 seconds (10000/60 = 166 RPS)
2. Convert 1 of 166 to decimal (1/166 = 0.0006 seconds per Rotation)
3. Multiply the seconds per rotation by 1000 milliseconds (6 MS per rotation)
4. Divide the total in half (6/2 = 3 MS) or RD
5. Add an average of 3 MS for seek time (3 MS + 3 MS = 6 MS)
6. Add 2 MS for latency (internal transfer) (6 MS + 2 MS = 8MS)
7. Divide 1000 MS by 8MS per I/O (1000/8 = 125 IOPS)

Each time an application issues an I/O, it takes an average of 8MS to service that I/O on a 10K RPM disk. Since this is a fixed time, it is imperative that the disk be as efficient as possible with the time it will spend reading and writing to the disk. The amount of I/O requests are often measured in I/Os Per Second (IOPS). The 10K RPM disk has the ability to push 120 to 150 (burst) IOPS. To measure the effectiveness of IOPS, divide the amount of IOPS by the amount of data read or written for each I/O.

7.2 Random vs Sequential I/O

The relevance of KB per I/O depends on the workload of the system. There are two different types of workload categories on a system. They are sequential and random.

7.2.1 Sequential I/O

The `iostat` command provides information about IOPS and the amount of data processed during each I/O. Use the `–x` switch with `iostat`. Sequential workloads require large amounts of data to be read sequentially and at once. These include applications like enterprise databases executing large queries and streaming media services capturing data. With sequential workloads, the KB per I/O ratio should be high. Sequential workload performance relies on the ability to move large amounts of data as fast as possible. If each I/O costs time, it is imperative to get as much data out of that I/O as possible.

```
# iostat -x 1

avg-cpu: %user   %nice   %sys    %idle
         0.00     0.00    57.1 4  42.86

Device:  rrqm/s wrqm/s  r/s  w/s   rsec/s   wsec/s   rkB/s   wkB/s   avgrq-sz avgqu-sz await svctm %util
/dev/sda  0.00  12891.43 0.00 105.71 0.00 106080.00 0.00 53040.00 1003.46 1099.43 3442.43 26.49 280.00
/dev/sda1 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
/dev/sda2 0.00  12857.14 0.00 5.71 0.00   105782.86   0.00 52891.43 18512.00  559.14 780.00 490.00 280.00
/dev/sda3 0.00  34.29  0.00  100.00 0.00  297.14     0.00 148.57       2.97  540.29 3594.57 24.00 240.00

avg-cpu: %user %nice %sys %idle
0.00 0.00 23.53 76.47

Device:  rrqm/s wrqm/s  r/s   w/s  rsec/s  wsec/s   rkB/s   wkB/s   avgrq-sz avgqu-sz await svctm %util
/dev/sda  0.00 17320.59 0.00 102.94 0.00 142305.88 0.00  71152.94    1382.40 6975.29 952.29 28.57 294.12
/dev/sda1 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
/dev/sda2 0.00 16844.12 0.00 102.94 0.00 138352.94 0.00 69176.47   1344.00  6809.71  952.29 28.57 294.12
/dev/sda3 0.00 476.47   0.00 0.00   0.00    952.94 0.00  1976.47      0.00    165.59   0.00   0.00 276.47
```

The way to calculate the efficiency of IOPS is to divide the reads per second (`r/s`) and writes per second (`w/s`) by the kilobytes read (`rkB/s`) and written (`wkB/s`) per second. In the above output, the amount of data written per I/O for `/dev/sda` increases during each iteration:

53040/105 = 505KB per I/O

71152/102 = 697KB per I/O

7.2.2 Random I/O

Random access workloads do not depend as much on size of data. They depend primarily on the amount of IOPS a disk can push. Web and mail servers are examples of random access workloads. The I/O requests are rather small. Random access workload relies on how many requests can be processed at once. Therefore, the amount of IOPS the disk can push becomes crucial.

```
# iostat -x 1
```



```
avg-cpu: %user %nice %sys %idle
           2.04 0.00 97.96 0.00

Device:  rrqm/s  wrqm/s r/s w/s rsec/s wsec/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util
/dev/sda 0.00  633.67 3.06 102.31 24.49 5281.63 12.24 2640.82 288.89 73.67 113.89 27.22 50.00
/dev/sda1 0.00   5.10  0.00 2.04  0.00  57.14   0.00   28.57   28.00  1.12  55.00 55.00 11.22
/dev/sda2 0.00 628.57 3.06 100.27 24.49 5224.49 12.24 2612.24 321.50 72.55 121.25 30.63 50.00
/dev/sda3 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 %sys %idle
           2.15 0.00 97.85 0.00

Device: rrqm/s wrqm/s r/s w/s  rsec/s wsec/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util
/dev/sda 0.00  41.94  6.45 130.98 51.61 352.69 25.81 3176.34 19.79  2.90    286.32 7.37 15.05
/dev/sda1 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
/dev/sda2 0.00 41.94  4.30 130.98 34.41 352.69 17.20 3176.34 21.18  2.90    320.00 8.24 15.05
/dev/sda3 0.00 0.00   2.15   0.00 17.20   0.00  8.60    0.00  8.00  0.00      0.00 0.00  0.00
```

The previous output shows that the amount of IOPS for writes stays almost the same as the sequential output. The difference is the actual write size per I/O:

2640/102 = 23KB per I/O

3176/130 = 24KB per I/O

7.3 When Virtual Memory Kills I/O

If the system does not have enough RAM to accommodate all requests, it must start to use the SWAP device. Just like file system I/O, writes to the SWAP device are just as costly. If the system is extremely deprived of RAM, it is possible that it will create a paging storm to the SWAP disk. If the SWAP device is on the same file system as the data trying to be accessed, the system will enter into contention for the I/O paths. This will cause a complete performance breakdown on the system. If pages can't be read or written to disk, they will stay in RAM longer. If they stay in RAM longer, the kernel will need to free the RAM. The problem is that the I/O channels are so clogged that nothing can be done. This inevitably can lead to a kernel panic and crash of the system.

The following `vmstat` output demonstrates a system under memory distress. It is writing data out to the swap device:

```
 procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b    swpd   free  buff  cache   si   so    bi    bo   in cs   us sy id wa
17  0     1250  3248 45820 1488472    30 132   992     0 2437 7657 23 50  0 23
11  0     1376  3256 45820 1488888    57 245   416     0 2391 7173 10 90  0 0
12  0     1582  1688 45828 1490228    63 131  1348    76 2432 7315 10 90  0 10
12  2     3981  1848 45468 1489824   185 56   2300    68 2478 9149 15 12  0 73
14  2     10385 2400 44484 1489732     0 87   1112    20 2515 11620 0 12  0 88
14  2     12671 2280 43644 1488816    76 51   1812   204 2546 11407 20 45 0 35
```

The previous output demonstrates a large amount of read requests into memory (`bi`). The requests are so many that the system is short on memory (`free`). This is causing the system to send blocks to the swap device (`so`) and the size of swap keeps growing (swpd). Also notice a large percentage of WIO time (wa). This indicates that the CPU is starting to slow because of I/O requests.

To see the effect the swapping to disk is having on the system, check the swap partition on the drive using iostat.

```
# iostat -x 1

avg-cpu: %user %nice %sys %idle
         0.00  0.00 100.00 0.00

Device: rrqm/s wrqm/s   r/s w/s rsec/s wsec/s    rkB/s    wkB/s   avgrq-sz  avgqu-sz   await svctm %util
/dev/sda  0.00  1766.67 4866.67 1700.00 38933.33 31200.00 19466.67 15600.00 10.68    6526.67 100.56 5.08
3333.33
/dev/sda1 0.00 933.33   0.00    0.00     0.00  7733.33     0.00  3866.67  0.00 20.00 2145.07 7.37 200.00
/dev/sda2 0.00 0.00  4833.33    0.00 38666.67   533.33  19333.33  266.67  8.11 373.33 8.07   6.90  87.00
/dev/sda3 0.00 833.33  33.33 1700.00   266.67 22933.33    133.33 11466.67   13.38   6133.33 358.46 11.35
1966.67
```

In the previous example, both the swap device (/dev/sda1) and the file system device (/dev/sda3) are contending for I/O. Both have high amounts of write requests per second (w/s) and high wait time (await) to low service time ratios (svctm). This indicates that there is contention between the two partitions, causing both to under perform.

7.4 Determining Application I/O Usage

The iotop command displays I/O usage per process. It is similar to the top command in its output. The iotop command can be used in conjunction with iostat to determine which application is causing the I/O bottleneck.

In the following example, there is both a read (find) and write (smbd) operation contending for the same disk.

The I/O stat shows both sustained read requests (r/s) and write requests (w/s) the same disk (sda1)

```
# iostat –x 1

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           7.14    0.00   35.71   57.14    0.00    0.00

Device:         rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     0.00  123.47   25.51   987.76 21951.02   153.97    27.76  224.29   6.85 102.04
sda1              0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00   0.00   0.00
sda2              0.00     0.00  123.47   25.51   987.76 21951.02   153.97    27.76  224.29   6.85 102.04
```

The following output shows that the find process is generating the read requests and the smbd process is generating the write requests.

```
# iotop –d 5 -P
```



```
Total DISK READ: 981.23 K/s | Total DISK WRITE: 21.43 M/s
  PID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 2574 be/4 root      967.01 K/s    0.00 B/s  0.00 % 39.05 % find /
   64 be/3 root        0.00 B/s   19.94 M/s  0.00 % 13.09 % smbd -D
 2533 be/4 dhoch       3.63 K/s    8.72 K/s  0.00 %  1.82 % [kjournald]
 2442 be/4 root        0.00 B/s    2.91 K/s  0.00 %  0.46 % iostat -x 1
 2217 be/4 dhoch       0.00 B/s 1488.57 B/s  0.00 %  0.00 % mono /usr~-ior-fd=25
 1985 be/4 dhoch       0.00 B/s  255.12 K/s  0.00 %  0.00 % smbd -D
```

The `DISK READ` and `DISK WRITE` columns can be correlated to the `iostat` `rsec/s` and `wsec/s` columns.

7.5 Conclusion

I/O performance monitoring consists of the following actions:

- **Any time the CPU is waiting on I/O, the disks are overloaded.**
- **Calculate the amount of IOPS your disks can sustain.**
- **Determine whether your applications require random or sequential disk access.**
- **Monitor slow disks by comparing wait times and service times.**
- **Monitor the swap and file system partitions to make sure that virtual memory is not contending for filesystem I/O.**

8.0 Introducing Network Monitoring

Out of all the subsyetms to monitor, networking is the hardest to monitor. This is due primarily to the fact that the network is abstract. There are many factors that are beyond a system's control when it comes to monitoring and performance. These factors include latency, collisions, congestion and packet corruption to name a few.

This section focuses on how to check the performance of Ethernet, IP and TCP.

8.1 Ethernet Configuration Settings

Unless explicitly changed, all Ethernet networks are auto negotiated for speed. The benefit of this is largely historical when there were multiple devices on a network that could be different speeds and duplexes.

Most enterprise Ethernet networks run at either 100 or 1000BaseTX. Use `ethtool` to ensure that a specific system is synced at this speed.

In the following example, a system with a 100BaseTX card is running auto negotiated in 10BaseT.

```
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: Yes
        Speed: 10Mb/s
        Duplex: Half
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0x00000007 (7)
        Link detected: yes
```


The following example demonstrates how to force this card into 100BaseTX:

```
# ethtool -s eth0 speed 100 duplex full autoneg off
# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: No
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: off
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0x00000007 (7)
        Link detected: yes
```

8.2 Monitoring Network Throughput

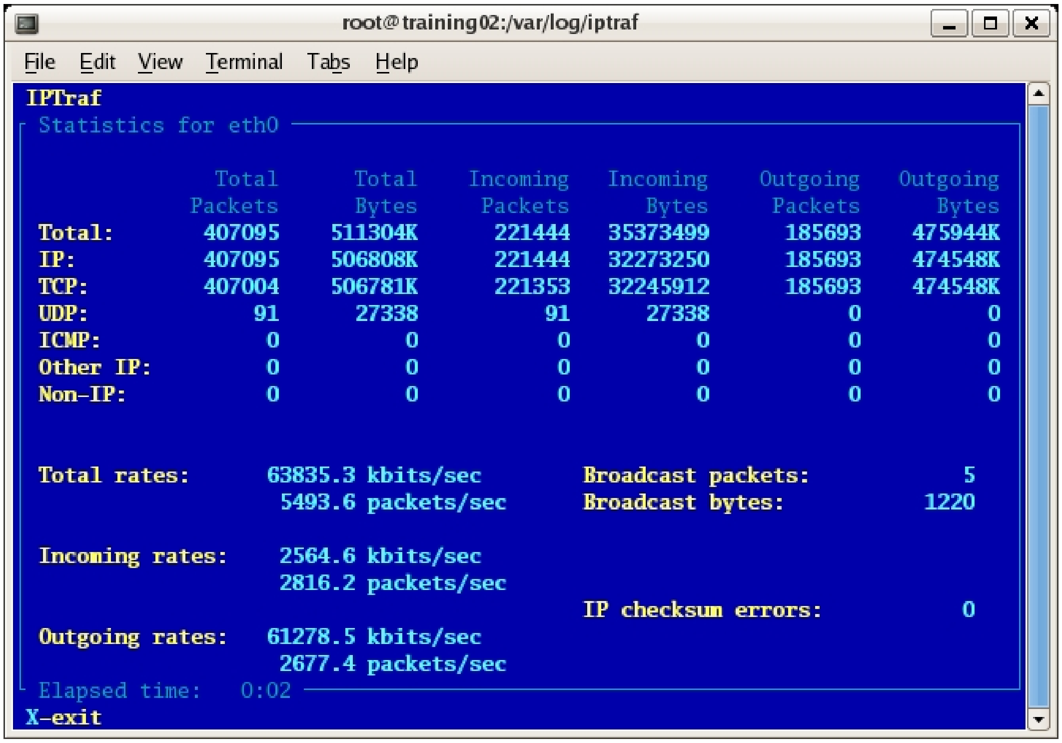
Just because an interface is now synchronized does not mean it is still having bandwidth problems. It is impossible to control or tune the switches, wires, and routers that sit in between two host systems. The best way to test network throughput is to send traffic between two systems and measure statistics like latency and speed.

8.2.0 Using `iptraf` for Local Throughput

The `iptraf` utility (`http://iptraf.seul.org`) provides a dashboard of throughput per Ethernet interface.

```
# iptraf –d eth0
```


Figure 1: Monitoring for Network Throughput



The previous output shows that the system tested above is sending traffic at a rate of 61 mbps (7.65 megabytes). This is rather slow for a 100 mbps network.

8.2.1 Using `netperf for` Endpoint Throughput

Unlike `iptraf` which is a passive interface that monitors traffic, the netperf utility enables a system administrator to perform controlled tests of network throughput. This is extremely helpful in determining the throughput from a client workstation to a heavily utilized server such as a file or web server. The `netperf` utility runs in a client/server mode.

To perform a basic controlled throughput test, the `netperf` server must be running on the server system:

```
server# netserver
Starting netserver at port 12865
Starting netserver at hostname 0.0.0.0 port 12865 and family AF_UNSPEC
```

There are multiple tests that the `netperf` utility may perform. The most basic test is a standard throughput test. The following test initiated from the client performs a 30 second test of TCP based throughput on a LAN:

The output shows that that the throughput on the network is around 89 mbps. The server (192.168.1.215) is on the same LAN. This is exceptional performance for a 100 mbps network.

```
client# netperf -H 192.168.1.215 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) 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    30.02      89.46
```

Moving off of the LAN onto a 54G wireless network within 10 feet of the router. The throughput decreases significantly. Out of a possible 54MBits, the laptop achieves a total throughput of 14 MBits

```
client# netperf -H 192.168.1.215 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.215 (192.168.1.215) 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    30.10      14.09
```

At a distance of 50 feet and down one story in a building, the signal further decreases to 5MBits.

```
# netperf -H 192.168.1.215 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.215 (192.168.1.215) 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    30.64       5.05
```


Moving off the LAN and onto the public Internet, the throughput drops to under 1Mbit.

```
# netperf -H litemail.org -p 1500 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
litemail.org (72.249.104.148) 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    31.58       0.93
```

The last check is the VPN connection, which has the worst throughput of all links on the network.

```
# netperf -H 10.0.1.129 -l 30
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
10.0.1.129 (10.0.1.129) 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    31.99       0.51
```

Another useful test using `netperf` monitors the amount of TCP request and response transactions taking place per second. The test accomplishes this by creating a single TCP connection and then sending multiple request/response sequences over that connection (`ack` packets back and forth with a byte size of 1). This behavior is similar to applications such as RDBMS executing multiple transactions or mail servers piping multiple messages over one connection.

The following example simulates TCP request/response over the duration of 30 seconds:

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to 192.168.1.230 (192.168.1.230) 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  1        1       30.00    4453.80
16384  87380
```

In the previous output, the network supported a transaction rate of 4453 `psh/ack` per second using 1 byte payloads. This is somewhat unrealistic due to the fact that most requests, especially responses, are greater than 1 byte.

In a more realistic example, a `netperf` uses a default size of 2K for requests and 32K for responses:

```
client# netperf -t TCP_RR -H 192.168.1.230 -l 30 -- -r 2048,32768
TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.1.230 (192.168.1.230) 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   30.00     222.37
16384  87380
```

The transaction rate reduces significantly to 222 transactions per second.

8.2.3 Using `iperf` to Measure Network Efficiency

The `iperf` tool is similar to the `netperf` tool in that it checks connections between two endpoints. The difference with `iperf` is that it has more in-depth checks around TCP/UDP efficiency such as window sizes and QoS settings. The tool is designed for administrators who specifically want to tune TCP/IP stacks and then test the effectiveness of those stacks.

The `iperf` tool is a single binary that can run in either server or client mode. The tool runs on port 5001 by default.

To start the server (`192.168.1.215`):

```
server# iperf -s -D
Running Iperf Server as a daemon
The Iperf daemon process ID : 3655
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

In the following example, the `iperf` tool on the client performs an iterative test of network throughput on a wireless network. The wireless network is fully utilized, including multiple hosts downloading ISO image files.

The client connects to the server (`192.168.1.215`) and performs a 60 second bandwidth test, reporting in 5 second iterations.


```
client# iperf -c 192.168.1.215 -t 60 -i 5
------------------------------------------------------------
Client connecting to 192.168.1.215, TCP port 5001
TCP window size: 25.6 KByte (default)
------------------------------------------------------------
[  3] local 192.168.224.150 port 51978 connected with
192.168.1.215 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.0 sec  6.22 MBytes  10.4 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3]  5.0-10.0 sec  6.05 MBytes  10.1 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 10.0-15.0 sec  5.55 MBytes  9.32 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 15.0-20.0 sec  5.19 MBytes  8.70 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 20.0-25.0 sec  4.95 MBytes  8.30 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 25.0-30.0 sec  5.21 MBytes  8.74 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 30.0-35.0 sec  2.55 MBytes  4.29 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 35.0-40.0 sec  5.87 MBytes  9.84 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 40.0-45.0 sec  5.69 MBytes  9.54 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 45.0-50.0 sec  5.64 MBytes  9.46 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 50.0-55.0 sec  4.55 MBytes  7.64 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3] 55.0-60.0 sec  4.47 MBytes  7.50 Mbits/sec
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-60.0 sec  61.9 MBytes  8.66 Mbits/sec
```

The other network traffic did have an effect on the bandwidth for this single host as seen in the fluctuations between 4 – 10 Mbits over a 60 second interval.

In addition to TCP tests, iperf has UDP tests to measure packet loss and jitter. The following `iperf` test was run on the same 54Mbit wireless G network with network load. As demonstrated in the previous example, the network throughput is currently only 9 out of 54 Mbits.


```
# iperf -c 192.168.1.215 -b 10M
WARNING: option -b implies udp testing
------------------------------------------------------------
Client connecting to 192.168.1.215, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   107 KByte (default)
------------------------------------------------------------
[  3] local 192.168.224.150 port 33589 connected with 192.168.1.215 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  11.8 MBytes  9.90 Mbits/sec
[  3] Sent 8420 datagrams
[  3] Server Report:
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  6.50 MBytes  5.45 Mbits/sec  0.480 ms 3784/ 8419 (45%)
[  3]  0.0-10.0 sec  1 datagrams received out-of-order
```

Out of the 10M that was attempted to be transferred, only 5.45M actually made it to the other side with a packet loss of 45%.

8.3 Individual Connections with `tcptrace`

The `tcptrace` utility provides detailed TCP based information about specific connections. The utility uses `libpcap` based files to perform and an analysis of specific TCP sessions. The utility provides information that is sometimes difficult to catch in a TCP stream. This information includes:

- **TCP Retransmissions – the amount of packets that needed to be sent again and the total data size**
- **TCP Window Sizes – identify slow connections with small window sizes**
- **Total throughput of the connection**
- **Connection duration**

8.3.1 Case Study – Using `tcptrace`

The `tcptrace` utility may be available in some Linux software repositories. This paper uses a precompiled package from the following website: `http://dag.wieers.com/rpm/packages/tcptrace.` The `tcptrace` command takes a source `libpcap` based file as an input. Without any options, the utility lists all of the unique connections captured in the file.

The following example uses a `libpcap` based input file called `bigstuff`:

```
# tcptrace bigstuff
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:01.634065, 89413 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
  1: 192.168.1.60:pcanywherestat - 192.168.1.102:2571 (a2b)       404>  450<
  2: 192.168.1.60:3356 - ftp.strongmail.net:21 (c2d)               35>   21<
  3: 192.168.1.60:3825 - ftp.strongmail.net:65023 (e2f)             5>    4<
(complete)
  4: 192.168.1.102:1339 - 205.188.8.194:5190 (g2h)                  6>    6<
```



```
  5: 192.168.1.102:1490 - cs127.msg.mud.yahoo.com:5050 (i2j)        5>    5<
  6: py-in-f111.google.com:993 - 192.168.1.102:3785 (k2l)         13>   14<
```

```
<snip>
```

In the previous output, each connection has a number associated with it and the source and destination host. The most common option to `tcptrace` is the `–l` and `–o` option which provide detailed statistics on a specific connection.

The following example lists all of the statistics for connection #16 in the `bigstuff` file:

```
# tcptrace -l -o1 bigstuff
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:00.529361, 276008 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
32 TCP connections traced:
TCP connection 1:
        host a:        192.168.1.60:pcanywherestat
        host b:        192.168.1.102:2571
        complete conn: no       (SYNs: 0)  (FINs: 0)
        first packet:  Sun Jul 20 15:58:05.472983 2008
        last packet:   Sun Jul 20 16:00:04.564716 2008
        elapsed time:  0:01:59.091733
        total packets: 854
        filename:      bigstuff
   a->b:                              b->a:
     total packets:           404           total packets:           450
     ack pkts sent:           404           ack pkts sent:           450
     pure acks sent:           13           pure acks sent:          320
     sack pkts sent:            0           sack pkts sent:            0
     dsack pkts sent:           0           dsack pkts sent:           0
     max sack blks/ack:         0           max sack blks/ack:         0
     unique bytes sent:     52608           unique bytes sent:     10624
     actual data pkts:        391           actual data pkts:        130
     actual data bytes:     52608           actual data bytes:     10624
     rexmt data pkts:           0           rexmt data pkts:           0
     rexmt data bytes:          0           rexmt data bytes:          0
     zwnd probe pkts:           0           zwnd probe pkts:           0
     zwnd probe bytes:          0           zwnd probe bytes:          0
     outoforder pkts:           0           outoforder pkts:           0
     pushed data pkts:        391           pushed data pkts:        130
     SYN/FIN pkts sent:       0/0           SYN/FIN pkts sent:       0/0
     urgent data pkts:          0 pkts      urgent data pkts:          0 pkts
     urgent data bytes:         0 bytes     urgent data bytes:         0 bytes
     mss requested:             0 bytes     mss requested:             0 bytes
     max segm size:           560 bytes     max segm size:           176 bytes
     min segm size:            48 bytes     min segm size:            80 bytes
     avg segm size:           134 bytes     avg segm size:            81 bytes
     max win adv:           19584 bytes     max win adv:           65535 bytes
     min win adv:           19584 bytes     min win adv:           64287 bytes
     zero win adv:              0 times     zero win adv:              0 times
     avg win adv:           19584 bytes     avg win adv:           64949 bytes
     initial window:          160 bytes     initial window:            0 bytes
     initial window:            2 pkts      initial window:            0 pkts
     ttl stream length:        NA           ttl stream length:        NA
     missed data:              NA           missed data:              NA
     truncated data:        36186 bytes     truncated data:         5164 bytes
     truncated packets:       391 pkts      truncated packets:       130 pkts
```



```
     data xmit time:      119.092 secs      data xmit time:      116.954 secs
     idletime max:       441267.1 ms        idletime max:       441506.3 ms
     throughput:              442 Bps       throughput:               89 Bps
```

8.3.2 Case Study – Calculating Retransmission Percentages

It is almost impossible to identify which connections have severe enough retransmission problems that require analysis. The `tcptrace` utility has the ability to use filters and Boolean expressions to locate problem connections. On a saturated network with multiple connections, it is possible that all connections may experience retransmissions. The key is to locate which ones are experiencing the most.

In the following example, the `tcptrace` command uses a filter to locate connections that retransmitted more than 100 segments:

```
# tcptrace -f'rexmit_segs>100' bigstuff
Output filter: ((c_rexmit_segs>100)OR(s_rexmit_segs>100))
1 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:00.687788, 212431 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
 16: ftp.strongmail.net:65014 - 192.168.1.60:2158 (ae2af) 18695> 9817<
```

In the previous output, connection #16 experienced had more than 100 retransmissions. From here, the `tcptrace` utility provides statistics on just that connection:

```
# tcptrace -l -o16 bigstuff
 arg remaining, starting with 'bigstuff'
Ostermann's tcptrace -- version 6.6.7 -- Thu Nov  4, 2004

146108 packets seen, 145992 TCP packets traced
elapsed wallclock time: 0:00:01.355964, 107752 pkts/sec analyzed
trace file elapsed time: 0:09:20.358860
TCP connection info:
32 TCP connections traced:
================================
TCP connection 16:
        host ae:       ftp.strongmail.net:65014
        host af:       192.168.1.60:2158
        complete conn: no       (SYNs: 0)  (FINs: 1)
        first packet:  Sun Jul 20 16:04:33.257606 2008
        last packet:   Sun Jul 20 16:07:22.317987 2008
        elapsed time:  0:02:49.060381
        total packets: 28512
        filename:      bigstuff
   ae->af:                             af->ae:

<snip>

     unique bytes sent:  25534744          unique bytes sent:         0
     actual data pkts:      18695          actual data pkts:          0
     actual data bytes:  25556632          actual data bytes:         0
```



```
rexmt  data pkts:       1605           rexmt data pkts:          0
rexmt  data bytes:   2188780           rexmt data bytes:         0
```

To calculate the retransmission rate:

```
rexmt/actual * 100 = Retransmission rate
```

Or

```
1605/18695* 100 = 8.5%
```

The previous connection had a retransmission rate of 8.5% which is the cause of the slow connection.

8.2.3 Case Study – Calculating Retransmits By Time

The tcptrace utility comes with a series of modules that present data by different dimensions (protocol, port, time, etc). The slice module enables you to view TCP performance over an elapsed time. Specifically, you can identify when exactly a series of retransmits occurred and tie that back to other performance data to locate a bottleneck.

The following example demonstrates how to create the time slice output file using tcptrace:

```
# tcptrace –xslice bigfile
```

This command creates a file called slice.dat in the present working directory. This specific file contains the information about retransmissions at 15 second intervals:

```
# ls -l slice.dat
-rw-r--r-- 1 root root 3430 Jul 10 22:50 slice.dat
# more slice.dat
date                segs    bytes  rexsegs rexbytes      new   active
--------------- -------- -------- -------- -------- -------- --------
22:19:41.913288       46     5672        0        0        1        1
22:19:56.913288      131    25688        0        0        0        1
22:20:11.913288        0        0        0        0        0        0
22:20:26.913288     5975  4871128        0        0        0        1
22:20:41.913288    31049 25307256        0        0        0        1
22:20:56.913288    23077 19123956       40    59452        0        1
22:21:11.913288    26357 21624373        5     7500        0        1
22:21:26.913288    20975 17248491        3     4500       12       13
22:21:41.913288    24234 19849503       10    15000        3        5
22:21:56.913288    27090 22269230       36    53999        0        2
22:22:11.913288    22295 18315923        9    12856        0        2
22:22:26.913288     8858  7304603        3     4500        0        1
```


8.4 Conclusion

To monitor network performance, perform the following actions:

- **Check to make sure all Ethernet interfaces are running at proper rates.**
- **Check total throughput per network interface and be sure it is inline with network speeds.**
- **Monitor network traffic types to ensure that the appropriate traffic has precedence on the system.**

Appendix A: Performance Monitoring Step by Step – Case Study

In the following scenario, an end user calls support and complains that the reporting module of a web user interface is taking 20 minutes to generate a report when it should take 15 seconds.

System Configuration

- **RedHat Enterprise Linux 3 update 7**
- **Dell 1850 Dual Core Xenon Processors, 2 GB RAM, 75GB 15K Drives**
- **Custom LAMP software stack**

Performance Analysis Procedure

1. Start with the output of `vmstat` for a dashboard of system performance.

```
# vmstat 1 10
procs                      memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 1  0 249844  19144  18532 1221212    0    0     7     3   22    17 25  8 17 18
 0  1 249844  17828  18528 1222696    0    0 40448     8 1384  1138 13  7 65 14
 0  1 249844  18004  18528 1222756    0    0 13568     4  623   534  3  4 56 37
 2  0 249844  17840  18528 1223200    0    0 35200     0 1285  1017 17  7 56 20
 1  0 249844  22488  18528 1218608    0    0 38656     0 1294  1034 17  7 58 18
 0  1 249844  21228  18544 1219908    0    0 13696   484  609   559  5  3 54 38
 0  1 249844  17752  18544 1223376    0    0 36224     4 1469  1035 10  6 67 17
 1  1 249844  17856  18544 1208520    0    0 28724     0  950   941 33 12 49  7
 1  0 249844  17748  18544 1222468    0    0 40968     8 1266  1164 17  9 59 16
 1  0 249844  17912  18544 1222572    0    0 41344    12 1237  1080 13  8 65 13
```

Key Data Points

- **There are no issues with memory shortages because there is no sustained swapping activity (`si` and `so`). Although the `free` memory is shrinking the `swpd` column does not change.**
- **There are no serious issues with the CPU. Although there is a bit of a run queue, the processor is still over 50% idle.**
- **There are a high amount of context switches (`cs`) and blocks being read in (`bo`).**
- **The CPU is stalled at an average of 20% waiting on I/O (`wa`).**

Conclusion: A preliminary analysis points to an I/O bottleneck.

2. Use `iostat` to determine from where the read requests are being generated.

```
# iostat -x 1
Linux 2.4.21-40.ELsmp (mail.example.com)  03/26/2007

avg-cpu:  %user   %nice    %sys   %idle
          30.00    0.00    9.33   60.67

Device:    rrqm/s wrqm/s   r/s   w/s  rsec/s  wsec/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svctm  %util
/dev/sda  7929.01  30.34 1180.91 14.23 7929.01  357.84  3964.50   178.92     6.93     0.39    0.03   0.06   6.69
/dev/sda1    2.67   5.46  0.40  1.76   24.62   57.77    12.31    28.88    38.11     0.06    2.78   1.77   0.38
/dev/sda2    0.00   0.30  0.07  0.02    0.57    2.57     0.29     1.28    32.86     0.00    3.81   2.64   0.03
/dev/sda3 7929.01  24.58 1180.44 12.45 7929.01  297.50  3964.50   148.75     6.90     0.32    0.03   0.06   6.68

avg-cpu:  %user   %nice    %sys   %idle
           9.50    0.00   10.68   79.82

Device:    rrqm/s wrqm/s   r/s   w/s  rsec/s  wsec/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svctm  %util
/dev/sda     0.00   0.00 1195.24  0.00    0.00    0.00     0.00     0.00     0.00    43.69    3.60   0.99 117.86
/dev/sda1    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
/dev/sda2    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
/dev/sda3    0.00   0.00 1195.24  0.00    0.00    0.00     0.00     0.00     0.00    43.69    3.60   0.99 117.86

avg-cpu:  %user   %nice    %sys   %idle
           9.23    0.00   10.55   79.22

Device:    rrqm/s wrqm/s   r/s   w/s  rsec/s  wsec/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svctm  %util
/dev/sda     0.00   0.00 1200.37  0.00    0.00    0.00     0.00     0.00     0.00    41.65    2.12   0.99 112.51
/dev/sda1    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
/dev/sda2    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
/dev/sda3    0.00   0.00 1200.37  0.00    0.00    0.00     0.00     0.00     0.00    41.65    2.12   0.99 112.51
```

Key Data Points

- **The only active partition is the /dev/sda3 partition. All other partitions are completely idle.**
- **There are roughly 1200 read IOPS (`r/s`) on a disk that supports around 200 IOPS.**
- **Over the course of two seconds, nothing was actually read to disk (`rkB/s`). This correlates with the high amount of wait I/O from the `vmstat`.**
- **The high amount of read IOPS correlates with the high amount of context switches in the `vmstat`. There are multiple read system calls issued.**

Conclusion: An application is inundating the system with more read requests than the I/O subsystem can handle.

3. Using `top`, determine what application is most active on the system

```
# top -d 1
 11:46:11  up 3 days, 19:13,  1 user,  load average: 1.72, 1.87, 1.80
176 processes: 174 sleeping, 2 running, 0 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total   12.8%    0.0%    4.6%   0.2%     0.2%   18.7%   63.2%
           cpu00   23.3%    0.0%    7.7%   0.0%     0.0%   36.8%   32.0%
           cpu01   28.4%    0.0%   10.7%   0.0%     0.0%   38.2%   22.5%
           cpu02    0.0%    0.0%    0.0%   0.9%     0.9%    0.0%   98.0%
           cpu03    0.0%    0.0%    0.0%   0.0%     0.0%    0.0%  100.0%
Mem:  2055244k av, 2032692k used,   22552k free,  0k shrd,   18256k buff
                   1216212k actv,  513216k in_d,   25520k in_c
Swap: 4192956k av,  249844k used, 3943112k free                 1218304k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14939 mysql     25   0  379M 224M 1117 R  38.2 25.7% 15:17.78 mysqld
 4023 root      15   0  2120  972  784 R  2.0  0.3   0:00.06 top
    1 root      15   0  2008  688  592 S  0.0  0.2   0:01.30 init
    2 root      34  19     0    0    0 S  0.0  0.0   0:22.59 ksoftirqd/0
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    4 root      10  -5     0    0    0 S  0.0  0.0   0:00.05 events/0
```

Key Data Points

- **The `mysql` process seems to be consuming the most resources. The rest of the system is completely idle.**
- **There is a wait on I/O reported by `top` (wa) which can be correlated with the `wio` field in `vmstat`.**

Conclusion: It appears the mysql is the only process that is requesting resources from the system, therefore it is probably the one generating the requests.

4. Now that MySQL has been identified as generating the read requests, use `strace` to determine what is the nature of the read requests.

```
# strace -p 14939

Process 14939 attached - interrupt to quit
read(29, "\3\1\237\1\366\337\1\222%\4\2\0\0\0\0\0012P/d", 20) = 20
read(29, "ata1/strongmail/log/strongmail-d"..., 399) = 399
_llseek(29, 2877621036, [2877621036], SEEK_SET) = 0
read(29, "\1\1\241\366\337\1\223%\4\2\0\0\0\0\0012P/da", 20) = 20
read(29, "ta1/strongmail/log/strongmail-de"..., 400) = 400
_llseek(29, 2877621456, [2877621456], SEEK_SET) = 0
read(29, "\1\1\235\366\337\1\224%\4\2\0\0\0\0\0012P/da", 20) = 20
read(29, "ta1/strongmail/log/strongmail-de"..., 396) = 396
_llseek(29, 2877621872, [2877621872], SEEK_SET) = 0
read(29, "\1\1\245\366\337\1\225%\4\2\0\0\0\0\0012P/da", 20) = 20
read(29, "ta1/strongmail/log/strongmail-de"..., 404) = 404
_llseek(29, 2877622296, [2877622296], SEEK_SET) = 0
read(29, "\3\1\236\2\366\337\1\226%\4\2\0\0\0\0\0012P/d", 20) = 20
```


Key Data Points

- **There are a large amount of reads followed by seeks indicating that the mysql application is generating random I/O.**
- **There appears to be a specific query that is requesting the read operations.**

Conclusion: The mysql application is executing some kind of read query that is generating all of the read IOPS.

5. Using the `mysqladmin` command, report on which queries are both dominating the system and taking the longest to run.

```
# ./mysqladmin -pstrongmail processlist

+----+------+-----------+------------+---------+------+----------+---------------------------------------
| Id | User | Host      | db         | Command | Time | State    | Info
+----+------+-----------+------------+---------+------+----------+---------------------------------------
| 1  | root | localhost | strongmail | Sleep   | 10   |          |
| 2  | root | localhost | strongmail | Sleep   | 8    |          |
| 3  | root | localhost | root       | Query   | 94   | Updating | update `failures` set
`update_datasource`='Y' where database_id='32' and  update_datasource='N' and  |
| 14 | root | localhost |            | Query   | 0    |          | show processlist
```

Key Data Points

- **The MySQL database seems to be constantly running an `update` query to a table called `failures`.**
- **In order to conduct the update, the database must index the entire table.**

Conclusion: An update query issued by MySQL is attempting to index an entire table of data. The amount of read requests generated by this query is devastating system performance.

Performance Follow-up

The performance information was handed to an application developer who analyzed the PHP code. The developer found a sub-optimal implementation in the code. The specific query assumed that the failures database would only scale to 100K records. The specific database in question contained 4 million records. As a result, the query could not scale to the database size. Any other query (such as report generation) was stuck behind the `update` query.

References

- **Ezlot, Phillip – Optimizing Linux Performance, Prentice Hall, Princeton NJ 2005 ISBN – 0131486829**
- **Johnson, Sandra K., Huizenga, Gerrit – Performance Tuning for Linux Servers, IBM Press, Upper Saddle River NJ 2005 ISBN 013144753X**
- **Bovet, Daniel Cesati, Marco – Understanding the Linux Kernel, O'Reilly Media, Sebastoppl CA 2006, ISBN 0596005652**
- **Blum, Richard – Network Performance Open Source Toolkit, Wiley, Indianapolis IN 2003, ISBN 0-471-43301-2**
- **Neil Horman - Understanding Virtual Memory in RedHat 4, , 12/05 http://people.redhat.com/nhorman/papers/rhel4_vm.pdf**
- **IBM, Inside the Linux Scheduler, http://www.ibm.com/developerworks/linux/library/l-scheduler/**
- **Aas, Josh - Understanding the Linux 2.6.8.1 CPU Scheduler, http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf**
- **Chen, Tim, Shi, Alex, Zhang, Yanim – Tuning Toolbox, Linux Magazine Pro March 2009**
- **Feilner, Markus, Spreitzer, Sascha – Top (tools) Contenders, Linux Magazine Pro December 2008**
- **Wieers, Dag, Dstat: Versatile Resource Statistics Tool,** `http://dag.wieers.com/home-made/dstat/`

