Linux Performance Monitoring

Darren Hoch

Services Architect – StrongMail Systems, Inc.

Linux Performance Monitoring

PUBLISHED BY:
Darren Hoch
dhoch@strongmail.com
http://www.ufsdump.org

http://www.strongmailsystems.com

Table of Contents

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.

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.

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.

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
`ethtool`	reports on Ethernet interface configuration	yes	yes

Installing the Monitoring Packages

The previously mentioned tools ship with most distributions, but are not installed as part of the base. The tools can be selected as an additional package during install or added later. The following example demonstrates how to install all the previously mentioned tools for the Fedora™ and CentOS distributions.

```
# yum install sysstat
# yum install iptraf
# yum install dstat
```

With the exception of `dstat` and `iptraf`, the other tools have periodically contained bugs in their counters. The type of bug and patch fix is distro dependent.

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:

- **Hardware Interrupts** – These are requests made by hardware on the system to process data. For example, a disk may signal an interrupt when it has completed and IO transaction or a NIC may signal that a packet has been received.
- **Soft Interrupts** – These are kernel software interrupts that have to do with maintenance of the kernel. For example, the kernel clock tick thread is a soft interrupt. It checks to make sure a process has not passed its allotted time on a processor.
- **Real Time Threads** – Real time threads have more priority than the kernel itself. A real time process may come on the CPU and preempt (or "kick off) the kernel. The Linux 2.4 kernel is NOT a fully preemptable kernel, making it not ideal for real time application programming.
- **Kernel Threads** – All kernel processing is handled at this level of priority.
- **User Threads** – 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.

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.

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.

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.

Time Slicing

The timeslice is the numeric value that represents how long a task can run until it is preempted. The scheduler policy must dictate a default timeslice, which is not simple. A timeslice that is too long will cause the system to have poor interactive performance; the system will no longer feel as if applications are being concurrently executed. A timeslice that is too short will cause significant amounts of processor time to be wasted on the overhead of switching processes, as a significant percentage of the system's time will be spent switching from one process with a short timeslice to the next. Furthermore, the conflicting goals of I/O-bound versus processor-bound processes again arise; I/O-bound processes do not need longer timeslices, whereas processor-bound processes crave long timeslices (to keep their caches hot, for example).

Static and Dynamic Priorities

The kernel scheduler assigns a default priority to each process. Processes with higher priorities run before processes with lower priorities. The nice command is used to modify the kernel scheduler assigned priorities by either favoring a processor more or less. The nice command takes a range of -20 (highest) to 19 (lowest).

The kernel scheduler employs a reward and penalty system to processes and how they use their allowed time slice. In addition to the default priorities, the kernel will either raise (reward) or lower (punish) the priority by 5. The bonus is calculated based on the amount of time a process spends in sleep state.

I/O bound processes spend most of their time in sleep state and as a result, they are rewarded by the scheduler.

CPU bound processes constantly use their time slice and are often penalized by the scheduler.

In the event that both a CPU bound and I/O bound process run on the same system, the I/O bound performance will stay the same while the CPU bound process will lose performance due to constant preemption by the I/O process.

The `ps` command displays the priorities of a given process:

```
# ps -eo pid,class,ni,pri,psr,comm | more
  PID CLS  NI PRI PSR COMMAND
    1 TS    0  24   0 init
    2 TS   19   5   0 ksoftirqd/0
    3 FF    - 139   0 watchdog/0
    4 TS   -5  29   0 events/0
    5 TS   -5  29   0 khelper
    6 TS   -5  28   0 kthread
    9 TS   -5  27   0 kblockd/0
   10 TS   -5  19   0 kacpid
   85 TS   -5  19   0 cqueue/0
   88 TS   -5  29   0 khubd
   90 TS   -5  29   0 kseriod
```

Static and dynamic prioritization is new to the kernel 2.6.

The following ps output shows the penalization in priority of a CPU intensive process called cpu-hog.

```
term1# ./cpu-hog

term2# while :; do  ps -eo pid,ni,pri,pcpu,comm | egrep
'hog|PRI'; sleep 1; done
    PID  NI PRI %CPU COMMAND
22855   0  20 84.5 cpu-hog
  PID  NI PRI %CPU COMMAND
22855   0  18 89.6 cpu-hog
  PID  NI PRI %CPU COMMAND
22855   0  15 92.2 cpu-hog
  PID  NI PRI %CPU COMMAND
22855   0  15 93.8 cpu-hog
```

By applying a nice value, we can further lower (show less favor) to the cpu-hog process. The kernel scheduler penalizes 5 and an additional 7 points via the nice value.

```
term1# nice –n 7 ./cpu-hog
# while :; do  ps -eo pid,ni,pri,pcpu,comm | egrep 'hog|PRI';
sleep 1; done
  PID  NI PRI %CPU COMMAND
22917  20   7  0.0 cpu-hog
  PID  NI PRI %CPU COMMAND
22917  15   5  98 cpu-hog
  PID  NI PRI %CPU COMMAND
22917  12   3  87.2 cpu-hog
  PID  NI PRI %CPU COMMAND
22917   9   0  98.8 cpu-hog
  PID  NI PRI %CPU COMMAND
22917  8   0  100 cpu-hog
  PID  NI PRI %CPU COMMAND
22917  8   0  97.6 cpu-hog
```

The following ps output tracks the find command. The find command is a heavy I/O bound process. It does not consume all of its timeslice, but rather it often goes into a sleep state. As a result, it is rewarded over time.


```
term1# find /

term2# while :; do  ps -eo pid,ni,pri,pcpu,comm | egrep
'find|PRI'; sleep 1; done
  PID  NI PRI %CPU COMMAND
23101   0  20  0.0 find
  PID  NI PRI %CPU COMMAND
23101   0  21  4.0 find
  PID  NI PRI %CPU COMMAND
23101   0  23  3.5 find
  PID  NI PRI %CPU COMMAND
23101   0  23  4.3 find
  PID  NI PRI %CPU COMMAND
23101   0  23  4.2 find
  PID  NI PRI %CPU COMMAND
23101   0  23  4.4 find
```

When run together, the I/O process is gradually rewarded and the CPU process penalized. The processor is also preempted more frequently, resulting in less available CPU cycles.

```
# while :; do  ps -eo pid,ni,pri,pcpu,comm | egrep 'find|hog';
sleep 1; done
23675   0  20 70.9 cpu-hog
23676   0  20  5.6 find
23675   0  20 69.9 cpu-hog
23676   0  21  5.6 find
23675   0  20 70.6 cpu-hog
23676   0  23  5.8 find
23675   0  19 71.2 cpu-hog
23676   0  23  6.0 find
23675   0  19 71.8 cpu-hog
23676   0  23  6.1 find
23675   0  18 72.8 cpu-hog
23676   0  23  6.2 find
23675   0  16 73.2 cpu-hog
23676   0  23  6.6 find
23675   0  14 73.9 cpu-hog
```

The kernel scheduling algorithm was completely rewritten in kernel 2.6 to be much more effective. Dubbed the "0(1)" scheduler, it has significant performance enhancements over the kernel 2.4 scheduler.

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.

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.

Using the dstat Utility

The `dstat` utility provides the ability to condense the output of `vmstat` down to specific CPU related fields.

```
# dstat -cip 1
----total-cpu-usage---- ----interrupts--- ---procs---
usr sys idl wai hiq siq|  15   169   185 |run blk new
  6   1  91   2   0   0|  12     0    13 |  0   0   0
  1   0  99   0   0   0|   0     0     6 |  0   0   0
  0   0 100   0   0   0|  18     0     2 |  0   0   0
  0   0 100   0   0   0|   0     0     3 |  0   0   0
```

One off the benefits of `dstat` over `vmstat` is that `dstat` provides interrupts per device. The first line of the `dstat` output for interrupts displays an interrupt number associated with a device. The interrupt number may be reconciled via the `/proc/interrupts` file.

```
# cat /proc/interrupts
           CPU0
  0: 1277238713    IO-APIC-edge  timer
  6:          5    IO-APIC-edge  floppy
  7:          0    IO-APIC-edge  parport0
  8:          1    IO-APIC-edge  rtc
  9:          1   IO-APIC-level  acpi
 14:    6011913    IO-APIC-edge  ide0
 15:   15761438    IO-APIC-edge  ide1
169:         26   IO-APIC-level  Intel 82801BA-ICH2
185:   16785489   IO-APIC-level  eth1
193:          0   IO-APIC-level  uhci_hcd:usb1
```

The following example demonstrates the difference between a system with an idle NIC card and a utilized NIC card (`eth1`).


```
# dstat -cip 1
----total-cpu-usage---- ----interrupts--- ---procs---
usr sys idl wai hiq siq|  15   169   185 |run blk new
  6   1  91   2   0   0|  12     0    13 |  0   0   0
  0   0 100   0   0   0|  15     0     6 |  0   0   0
  0   0 100   0   0   0|   3     0     3 |  0   0   0
  1   0  99   0   0   0|   0     0     3 |  0   0   0
  0   0 100   0   0   0|  18     0     2 |  0   0   0
  0   0 100   0   0   0|   0     0     4 |  0   0   0

# while :
> do wget http://www.litemail.org/index.html
> done

# dstat –cip 1
----total-cpu-usage---- ----interrupts--- ---procs---
usr sys idl wai hiq siq|  15   169   185 |run blk new
  1   2  97   0   0   0|  18     0    67 |  0   0   2
  2   3  95   0   0   0|   0     0    91 |  0   0   3
  5   3  90   0   0   2|  18     0  1064 |  0   0   4
  5   3  91   0   0   1|   0     0   400 |  0   0   5
  3   3  93   0   0   1|  18     0   515 |  0   0   5
  2   3  94   1   0   0|   0     0   103 |  0   0   4
```

The `eth1` device has an id of `185`. The amount of interrupts generated by the `wget` command utilized an average of 8% of the CPU as the idle time decreased from an average of 99% idle to 92% idle.

Case Study: Application Spike

In the following example, a system is experiencing CPU performance spikes, going from completely idle to 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
 4  0 200560  91656  88596 176092    0    0     0     0  103    12  0  0  0 100
 0  0 200560  91660  88600 176092    0    0     0     0  104    12  0  0  0 100
 0  0 200560  91660  88600 176092    0    0     0     0  103    16  1  0  0 99
 0  0 200560  91660  88600 176092    0    0     0     0  103    12  0  0  0 100
 0  0 200560  91660  88604 176092    0    0     0    80  108    28  0  0  6 94
 0  0 200560  91660  88604 176092    0    0     0     0  103    12  0  0  0 100
 1  0 200560  91660  88604 176092    0    0     0     0  103    12  0  0  0 100
 1  0 200560  91652  88604 176092    0    0     0     0  113    27 14  3  0 83
 1  0 200560  84176  88604 176092    0    0     0     0  104    14 95  5  0  0
 2  0 200560  87216  88604 176092    0    0     0   324  137    96 86  9  1  4
 2  0 200560  78592  88604 176092    0    0     0     0  104    23 97  3  0  0
 2  0 200560  90940  88604 176092    0    0     0     0  149    63 92  8  0  0
 2  0 200560  83036  88604 176092    0    0     0     0  104    32 97  3  0  0
 2  0 200560  74916  88604 176092    0    0     0     0  103    22 93  7  0  0
 2  0 200560  80188  88608 176092    0    0     0   376  130   104 70 30  0  0
 3  0 200560  74028  88608 176092    0    0     0     0  103    69 70 30  0  0
 2  0 200560  81560  88608 176092    0    0     0     0  219   213 38 62  0  0
 1  0 200560  90200  88608 176100    0    0     8     0  153   118 56 31  0 13
 0  0 200560  88692  88612 179036    0    0  2940     0  249   249 44  4 24 28
 2  0 200560  88708  88612 179036    0    0     0   484  254    94 39 22  1 38
 0  0 200560  88708  88612 179036    0    0     0     0  121    22  0  0  0 100
 0  0 200560  88708  88612 179036    0    0     0     0  103    12  0  0  0 100
```


The following observations are made from the output:

- The run queue (r) during the spike goes as high as 3, almost passing the threshold.
- The percentage of CPU time in the user space (us)goes to almost 90%, but then levels off.
- During this time, the amount of context switches (cs) does not increase significantly, this could suggest that a single threaded application used a large amount of processor for a short period of time.
- It appears that the application batches all disk writes in one action. For one second, the CPU experiences a disk usage spike (wa = 24%)

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.

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.

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
```


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
```

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

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.

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.

Virtual Size (VSZ) and Resident Set Size (RSS)

When an application starts, it requests virtual memory (VSZ). The kernel either grants or denies the virtual memory request. As the application uses the requested memory, that memory is mapped into physical memory. The RSS is the amount of virtual memory that is physically mapped into memory. In most cases, an application uses less resident memory (RSS) than it requested (VSZ).

The following output from the ps command displays the VSZ and RSS values. In all cases, VSZ is greater than RSS. This means that although an application requested virtual memory, not all of it is allocated in RAM (RSS).

```
# ps –aux
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND

<snip>

daemon    2177  0.0  0.2  3352  648 ?        Ss   23:03   0:00 /usr/sbin/atd
dbus      2196  0.0  0.5 13180 1320 ?        Ssl  23:03   0:00 dbus-daemon-1 --sys
root      2210  0.0  0.4  2740 1044 ?        Ss   23:03   0:00 cups-config-daemon
root      2221  0.3  1.5  6108 4036 ?        Ss   23:03   0:02 hald
root      2231  0.0  0.1  2464  408 tty1     Ss+  23:03   0:00 /sbin/mingetty tty1
root      2280  0.0  0.1  3232  404 tty2     Ss+  23:03   0:00 /sbin/mingetty tty2
root      2343  0.0  0.1  1692  408 tty3     Ss+  23:03   0:00 /sbin/mingetty tty3
root      2344  0.0  0.1  2116  404 tty4     Ss+  23:03   0:00 /sbin/mingetty tty4
root      2416  0.0  0.1  1476  408 tty5     Ss+  23:03   0:00 /sbin/mingetty tty5
root      2485  0.0  0.1  1976  408 tty6     Ss+  23:03   0:00 /sbin/mingetty tty6
root      2545  0.0  0.9 10920 2336 ?        Ss   23:03   0:00 /usr/bin/gdm-binary
```


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.

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.

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]
```


Low on Memory Reclaiming (LMR)

The LMR attempts to reclaim pages when a page allocation fails. Page allocations fail when `kswapd` can't maintain the enough free memory. It attempts to free 1024 dirty pages per iteration until memory allocation is successful.

Out of Memory Killer (OMK)

The kernel implements OMK when the LMR can't reclaim pages fast enough. The OMK uses a selective algorithm (`select_bad_process()`) to determine which processes to kill. Once OMK has selected a process, it will send it a `SIGKILL.` This will immediately free pages. The OMK selects a process to kill based on the following criteria:

- The process owns a large number of page frames.
- The process should only lose a small amount of work.
- The process should have a low static priority process.
- The process should not be owned by `root`.

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.

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`).

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.

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.

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>
```

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>
```

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.

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).

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]
```


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.

Condition 1: Too Much I/O at Once

In an ideal environment, a CPU splits a percentage of its time between user (65%), kernel (30%) and idle (5%). If I/O starts to cause a bottleneck on the system, a new condition called "Wait on I/O" (WIO) appears in CPU performance statistics. A WIO condition occurs when a CPU is completely idle because all runnable processes are waiting on I/O. This means that all applications are in a sleep state because they are waiting for requests to complete in the I/O subsystem.

The `vmstat` command provides WIO statistics in the last 4 fields of output under the `"cpu"` header.

```
# vmstat 1
procs -----memory----- ---swap---io---- --system--cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
3 2 0 55452 9236 1739020 0 0 9352 0 2580 8771 20 24 0 57
2 3 0 53888 9232 1740836 0 0 14860 0 2642 8954 23 25 0 52
2 2 0 51856 9212 1742928 0 0 12688 0 2636 8487 23 25 0 52
```

These last 4 columns provide percentages of CPU utilization for user (`us`), kernel (`sys`), idle (`id`), and WIO (`wa`). In the previous output, the CPU averages 50% idle waiting on I/O requests to complete. This means that there is 50% of the processor that is usable for executing applications, but no applications can execute because the kernel is waiting on I/O requests to complete. You can observe this in the blocked threads column (`b`).

It is also worth noting that the major cause of the I/O bottleneck is disk reads due to the large amount of disk blocks read into memory (`bi`). There is no data being written out to disk as the blocks out (`bo`) column has a zero value. From this output alone, it appears that the system is processing a large I/O request.

The `sar` command without any options also provides CPU percentages that include WIO (`%iowait`) percentages:

```
# sar 1 100
07:25:55 PM       CPU %user %nice   %system     %iowait %idle
07:25:56 PM       all 74.26 0.00    25.74       0.00    0.00
07:25:57 PM       all 52.00 0.00    31.00       16.00   1.00
07:25:58 PM       all 12.87 0.00    13.86       73.27   0.00
```

The `sar` command with the `-B` option provides statistics on kilobytes read (`pgpgin/s`) and written out (`pgpgout/s`) of memory that may be correlated with the `bi` and `bo`

columns of `vmstat`. The `sar -B` command also shows MnPF (`fault/s`) and MPF statistics (`majflt/s`).

```
# sar -B 1 100
07:28:23 PM pgpgin/s pgpgout/s fault/s majflt/s
07:28:24 PM 6653.47   463.37  1604.95  74.26
07:28:25 PM 7448.00    96.00  2061.00  79.00
07:28:26 PM 4190.10   118.81  723.76   30.69
07:28:27 PM 2966.34   146.53  525.74    9.90
07:28:28 PM 3728.00     0.00  146.00    6.00
07:28:29 PM 5244.00   580.00  927.00   39.00
```

There is no exact tool that can identify which application is causing the I/O read requests. The `top` tool can provide enough insight to make an educated guess. Start the `top` command with a delay of 1 second:

```
# top -d 1
```

Once `top` is running, sort the output by faults (MPF and MnPF) by typing "F" to bring up the sort menu and "u" to sort by faults.

```
# top -d 1
top - 19:45:07 up 1:40, 3 users, load average: 6.36, 5.87, 4.40
Tasks: 119 total, 3 running, 116 sleeping, 0 stopped, 0 zombie
Cpu(s): 5.9% us, 87.1% sy, 0.0% ni, 0.0% id, 5.9% wa, 1.0% hi, 0.0% si
Mem: 2075672k total, 2022668k used, 53004k free, 7156k buffers
Swap: 2031608k total, 132k used, 2031476k free, 1709372k cached
PID USER PR  NI VIRT RES SHR   S %CPU %MEM  TIME+    nFLT COMMAND
3069 root 5 -10 450m 303m 280m S  61.5 15.0 10:56.68 4562 vmware-vmx
3016 root 5 -10 447m 300m 280m S  21.8 14.8 12:22.83 3978 vmware-vmx
3494 root 5 -10 402m 255m 251m S   3.0 12.6 1:08.65  3829 vmware-vmx
3624 root 5 -10 401m 256m 251m S   1.0 12.6 0:29.92  3747 vmware-vmx
<snip>
```

The previous output demonstrates that a series of VMWare process are causing the majority of page faults (`nFLT`) which would contribute to the surge of read requests seen in the previous commands. This surge has caused the WIO condition on the system, rendering the CPU idle and causing the system to appear much slower.

Condition 2: Pipes Too Small

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.

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.

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

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

Condition 3: Slow Disks

Many disk configurations may not be a physical disk on the system. Some may be part of a volume group, NAS, shared drive, or SAN. It is possible to measure the latency between the request time and the actual service time of a device.

Many disk configurations may not be a physical disk on the system. Some may be part of a volume group, NAS, shared drive, or SAN. It is possible to measure the latency between the request time and the actual service time of a device.

The following output was taken from a system with Linux volume groups under extreme sequential I/O write access.

```
# iostat -x 1

<snip>

avg-cpu: %user %nice %sys  %iowait %idle
          0.50   0.00 30.94   8.19  60.37

Device: rrqm/s wrqm/s r/s   w/s  rsec/s wsec/s  rkB/s  wkB/s  avgrq-sz avgqu-sz await svctm %util
hda      0.00 2610.03 0.00 6.02   0.00 20984.62 0.00 10492.31  3485.78  8.59   315.28 56.50 34.01
hdb      0.00 2610.03 0.00 6.02   0.00 20984.62 0.00 10492.31  3485.78  8.40   284.56 56.44 33.98
md0      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
md1      0.00   0.00  0.00 2622.74 0.00 20981.94 0.00 10490.97    8.00  0.00     0.00  0.00  0.00

avg-cpu: %user %nice %sys %iowait %idle
          0.83   0.00 59.27 4.01    35.89

Device: rrqm/s wrqm/s r/s   w/s  rsec/s wsec/s rkB/s  wkB/s  avgrq-sz avgqu-sz await  svctm %util
hda      0.00 13595.64 0.00 10.07 0.00 109197.32 0.00 54598.66 10846.93 95.15 1872.43 100.07
100.74
hdb      0.00 13595.64 0.00 10.40 0.00 109197.32 0.00 54598.66 10497.03 94.64 1854.52 96.84
100.74
md0      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
md1      0.00   0.00   0.00 13649.66 0.00 109197.32 0.00 54598.66 8.00  0.00     0.00  0.00  0.00

avg-cpu: %user %nice %sys %iowait %idle
          0.34 0.00 20.94 62.31 16.42

Device: rrqm/s wrqm/s r/s   w/s  rsec/s wsec/s rkB/s  wkB/s  avgrq-sz avgqu-sz await  svctm %util
hda      0.00 3420.07 0.00 11.37 0.00 27478.26 0.00 13739.13 2416.47  158.53 2997.18 88.24 100.33
hdb      0.00 3420.07 0.00 11.37 0.00 27478.26 0.00 13739.13 2416.47 157.97  2964.79 88.24 100.33
md0      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
md1      0.00 0.00    0.00 3434.78 0.00 27478.26 0.00 13739.13    8.00  0.00    0.00   0.00  0.00
```

The previous `iostat` output monitors a RAID 1 device (`/dev/md1`). Notice the difference between the service time (`svctm`) and the average wait time (`await`). Both of these values are in milliseconds. The actual time it takes to service the request is minimal compared to the time the system spends waiting for the response.

Since a mirror has to sync to another disk, the amount of I/O doubles. Notice in the following output that the disks were completely idle waiting on I/O. Even though requests were still being made (`w/s`), there was no disk activity and a large discrepancy between the service time and average wait time. Also notice that the disks were 100% utilized

even though nothing was writing. This is indicative of a complete stall waiting for the volume group software to catch up.

```
avg-cpu: %user %nice %sys %iowait %idle
          0.00   0.00 1.00  52.68 46.32

Device: rrqm/s wrqm/s r/s  w/s   rsec/s wsec/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util
hda      0.00  0.00   0.00 11.00 0.00   0.00   0.00  0.00   0.00   145.44 5848.03 90.94 100.03
hdb      0.00  0.00   0.00 10.67 0.00   0.00   0.00  0.00   0.00   144.89 5872.97 93.78 100.03
md0      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
md1      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 %iowait %idle
          0.17   0.00 0.84  49.00 50.00

Device: rrqm/s wrqm/s r/s  w/s   rsec/s wsec/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util
hda      0.00  0.00   0.00 10.96 0.00   0.00   0.00  0.00   0.00    111.83 8053.45 90.94 99.70
hdb      0.00  0.00   0.00 10.96 0.00   0.00   0.00  0.00   0.00    111.28 8003.18 90.94 99.70
md0      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
md1      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
```

This behavior can also be observed using the `sar -b` command. This command shows I/O statistics per device node number. To locate the node numbers of your devices, use an `ls` command with a `-lL` switch.

```
# ls -lL /dev/md1
brw-rw---- 1 root disk 9, 1 Dec 30 08:13 /dev/md1
# ls -lL /dev/hda
brw-rw---- 1 root disk 3, 0 Dec 30 08:13 /dev/hda
# ls -lL /dev/hdb
brw-rw---- 1 root disk 3, 64 Dec 30 08:13 /dev/hdb
```

The mirror device has a major number of 9 and a minor number of 1 or 91. The other two disks are 3, 0 and 3, 64. Looking at the sar output, it appears that the RAID device issues a large amount of I/O writes to the underlying drives. The drives end up timing out trying to process the requests.


```
# sar -b 3

<snip>

04:28:14 PM dev3-0 11.11 0.00 106650.51
04:28:14 PM dev3-64 10.10 0.00 106634.34
04:28:14 PM dev9-0 0.00 0.00 0.00
04:28:14 PM dev9-1 13326.26 0.00 106610.10

<snip>

04:28:15 PM dev3-0 9.90 0.00 0.00
04:28:15 PM dev3-64 10.89 0.00 0.00
04:28:15 PM dev9-0 0.00 0.00 0.00
04:28:15 PM dev9-1 0.00 0.00 0.00

<snip>
```

Condition 4: 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.

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.

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.

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
```

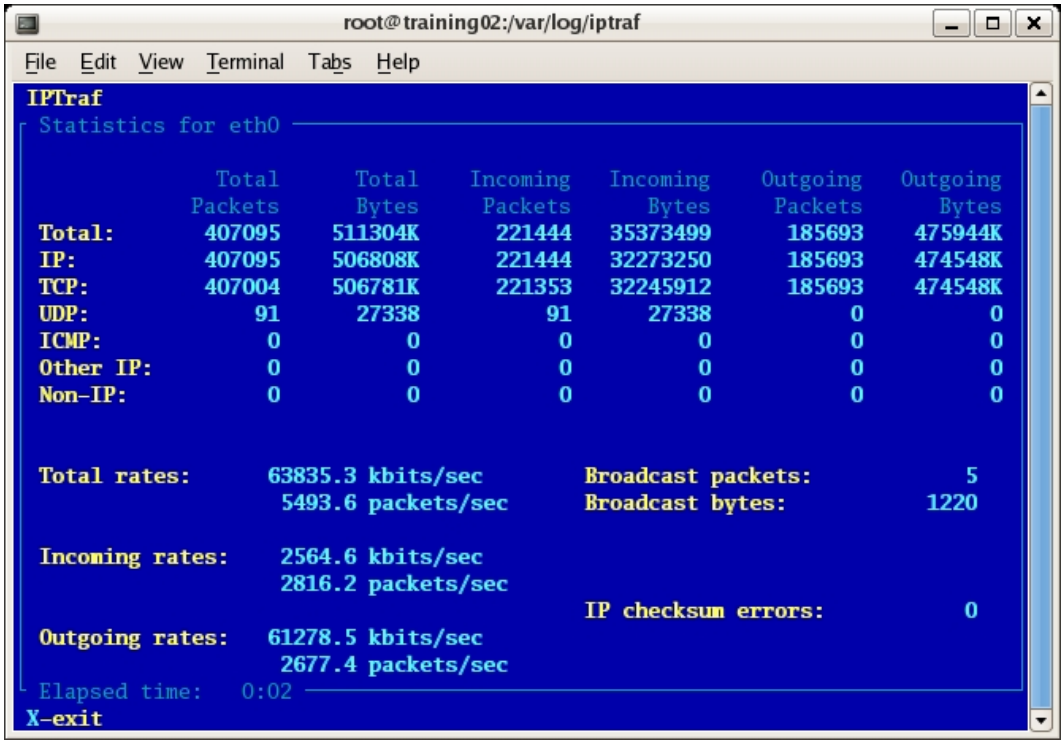

The following example demonstrates how to force this card into 100BaseTX:

```
# 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
```

Just because an interface is now synchronized does not mean it is still having bandwidth problems. The `iptraf` utility (`http://iptraf.seul.org`) provides a dashboard of throughput per Ethernet interface.

```
# iptraf –d eth0
```


Figure 1: Monitoring for Network Throughput



Monitoring for Error Conditions

The most common kind of error condition checked is for is packet collisions. Most enterprise networks are in a switched environment, practically eliminating collisions. However, with the increased usage of networked based services, there are other conditions that may arise. These conditions include dropped frames, backlogged buffers, and overutilized NIC cards.

Under extreme network loads, the `sar` command provides a report on all possible error types on a network.

```
# sar -n FULL  5 100
Linux 2.6.9-55.ELsmp (sapulpa)  06/23/2007

11:44:32 AM     IFACE   rxpck/s   txpck/s   rxbyt/s   txbyt/s   rxcmp/s   txcmp/s  rxmcst/s
11:44:37 AM        lo      6.00      6.00    424.40    424.40      0.00      0.00      0.00
11:44:37 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      sit0      0.00      0.00      0.00      0.00      0.00      0.00      0.00

11:44:32 AM     IFACE   rxerr/s   txerr/s    coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
11:44:37 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
11:44:37 AM      sit0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

11:44:32 AM    totsck    tcpsck    udpsck    rawsck   ip-frag
11:44:37 AM       297        79         8         0         0
```

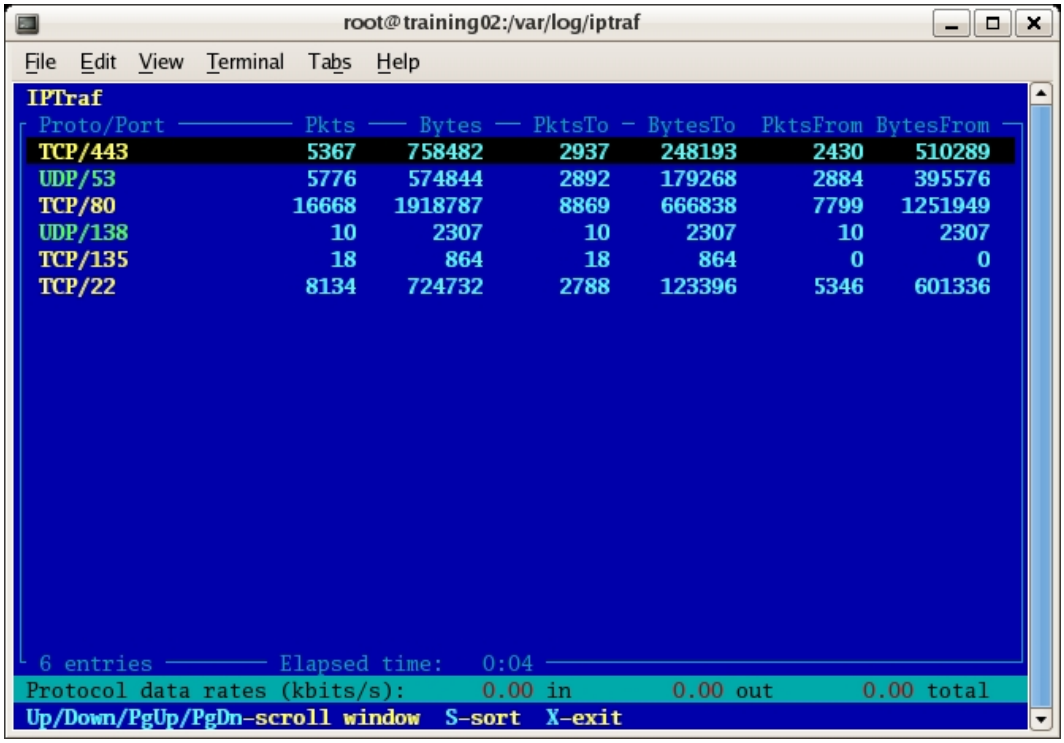
Table 3: Types of Network Errors

Field	Description
`rxerr/s`	rate of receive errors
`tcerr/s`	rate of transmit errors
`coll/s`	rate of collisions
`rxdrop/s`	received frames dropped due to kernel buffer shortage
`txdrop/s`	trasnmitted frames dropped due to kernel buffer shortage
`txcarr/s`	carrier errors
`rxfram/s`	frame alignement errors
`rxfifo/s`	receiving FIFO errors
`tcfifo/s`	trasmitted FIFO errors

Monitoring Traffic Types

Certain systems are designed to serve different traffic. For instance, a web server serves traffic over port 80 and a mail server over port 25. The `iptraf` tool determines displays the highest volume of traffic per TCP port.

Figure 2: Monitoring TCP Traffic per Port



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.

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
- Understanding Virtual Memory in RedHat 4, Neil Horman, 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
- Wieers, Dag, Dstat: Versatile Resource Statistics Tool, `http://dag.wieers.com/home-made/dstat/`

