The material for this tutorial was taken from Darren Hoch's "Linux System and Performance Monitoring". You can access it at: http://ufsdump.org/papers/oscon2009-linux-monitoring.pdf.

00 Tutorial objectives

- Offer an introduction to Performance Monitoring
- Get you acquainted with the *vmstat* command and its output
- Get you to use simple commands for monitoring the performance of the CPU and Memory

01 Introduction to Performance Monitoring

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 and Network.

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

In order to choose where to start looking for tuning bottlenecks, it is 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

System utilisation 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. When dealing with a system under heavy utilisation, it needs to be determined whether the system is managing or not.

Ex 00

- Run ***vmstat*** on your machine with a 1 second delay between updates. Let it run for approximately 10 seconds. Notice the CPU utilisation by looking at the ***id*** column.
- Run ***ex00.py*** script. Then run ***vmstat*** as before. What do you notice in this case?
- Without opening ex00.py, is there anything you can deduce regarding the script?
- Have a look inside ex00.py.
- How would you get the system to be 100% utilised with the CPU no longer being idle? Try it.

The ***vmstat*** utility provides a good low-overhead view of system performance. Since ***vmstat*** is such a low-overhead tool, it is practical to have it running even on heavily loaded servers when it is needed to monitor the system's health.

02 Introducing the CPU

The CPU utilisation is dependent on what resources are 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 assigns different priorities to different resources. The following list outlines the priorities from the highest to the 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 utilisation.

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 (hardware interrupts, for example), that thread is placed 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 utilised, 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 Utilisation

The CPU utilisation is an important metric for measuring a system's behaviour. Most performance monitoring tools divide the CPU utilisation 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.

03 CPU Performance Monitoring

Understanding how well a CPU is performing is a matter of interpreting the run queue, its utilisation, and the amount of context switching performed. Although performance is

relative to baseline statistics, in the absence of these statistics, the following general performance expectations of a system can be used as a guideline:

- Run Queues – A run queue should not have more than 3 threads queued per processor. For example, a dual processor system should not have more than 6 threads in the run queue.
- CPU Utilisation – A fully utilised CPU should have the following utilisation distribution:
    - 65% – 70% User Time
    - 30% - 35% System Time
    - 0% - 5% Idle Time
- Context Switches – The amount of context switches is directly relevant to CPU utilisation. As long as the CPU sustains the previously presented utilisation distribution, it is acceptable to have a high amount of context switches.

The following two examples give interpretations of the outputs generated by vmstat.

Example – Sustained CPU Utilisation

```
# 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 can be made based on 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.

Example – Overloaded Scheduler

```
# 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 observations can be made based on 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.

Ex 01

- Open ***ex01.py***. What does it do?
- Try running it firstly for a number up to 998, and then for any larger number. What is the problem?
- Run ***ex01b.py*** instead, for a large number (e.g. 10,000). Use the ***vmstat*** command to check the CPU usage. Try to explain what is happening.

  Hint: Use the ***mpstat*** command to monitor each individual core. The ***mpstat*** command provides the same CPU utilization statistics as ***vmstat***, but ***mpstat*** breaks the statistics out on a per processor basis.
- What do you think is causing this behaviour?
- Comment line 14 (the print call) and the run the script again noticing its behaviour.
- Run both ***ex00.py*** and ***ex01b.py*** simultaneously, and use ***vmstat*** and ***mpstat*** to monitor what is going on.

Takeaway actions for monitoring the CPU performance

- Checking the system run queue and making sure there are no more than 3 runnable threads per processor
- Making sure the CPU utilization is split between 70/30 between user and system
- Knowing that when the CPU spends more time in system mode, it is more than likely overloaded and trying to reschedule priorities
- CPU bound process always get penalized while I/O process are rewarded

04 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.

Kernel Memory Paging

Memory paging is a normal activity not to be confused with memory swapping. Memory paging is the process of syncing 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.

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.

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***.

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

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: Swapd, Free, Buff, Cache, So, Si, Bo, Bi (use ***man vmstat*** to read their description).

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 can be made based on 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).

Ex02

- Use the ***stress*** command to stress the RAM module using 2 processes, with each about 300MiB in size (command: *stress -m 2 --vm-bytes 300M*).
  sUse the ***vmstat*** utility to see the effects on the memory and CPU.
- For stressing the RAM, the CPU also has to do a lot of work. And as a result, if the used processes were more or equal to the available cores, then they will use 100% of your CPU power. Test this.
- Stressing both the CPU and memory in the same time and setting a timeout (e.g. stress -c 2 -m 2 -t 20s)

  Read more about stress here: http://www.hecticgeek.com/2012/11/stress-test-your-ubuntu-computer-with-stress/

Ex03

- Generate an array of random integers, having a length of 10,000,000. Write a function for sorting this array without using built-in sorting functions.
  Monitor the behaviour of the system, in terms of CPU and memory utilisation, while running your code.

Conclusions on monitoring the virtual memory performance:

• 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