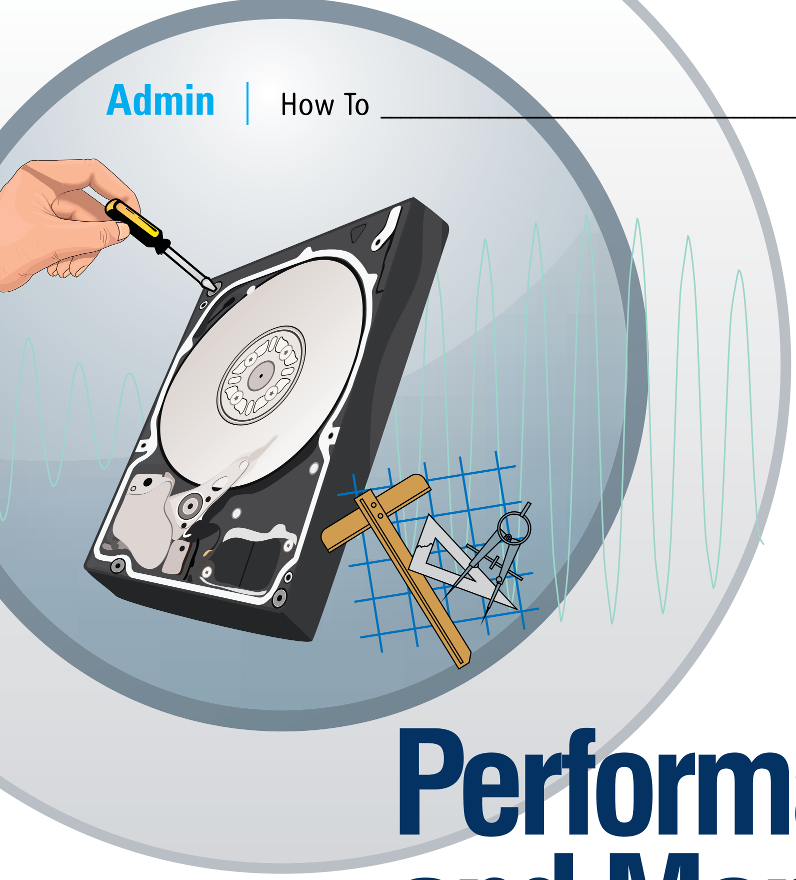

Part 2

Learn how to tune your hard disk I/O using read_ahead, elevators, queuing theory and the utilisation law.

Performance Tuning and Monitoring

Last month we learned how to check the capabilities of our existing hardware, and how to ascertain the current load on the system using tools like *iostat, dmidecode, sar, mrtg, gnuplot,* etc. Well, it's the right time to tune your system to your needs, after monitoring the data now available to you.

First, tune your HDD I/O.

Tuning disk I/O using elevators

Hard disks are electro-mechanical devices with two kinds of components—internal and external. While the internal component is essentially the sealed chamber called the hard drive assembly (HDA), the external components are located on a printed circuit board (PCB), often referred as the logic board. Refer to Figures 1 and 2.

Today's hard disk drives are made of platters coated with magnetic material. An electromechanically actuated armature contains the 'read/write' heads that move between these platters. The armature is normally composed of multiple arms. All the arms are bound together to work as a single unit. So when the armature moves to a specific track on the platter, all the read/write heads are placed on the same track on each platter. This is known as a 'cylinder'.

Remember that whenever there is a moving part involved in I/O, there will be latency. As our HDD also has moving parts—the platters and the armature—we cannot expect a 'wire speed' I/O. So whenever the disk controller requests a read/write from a particular sector, the armature has to move from its current position to the new requested position. This delay is known as the seek time. Add the time the motor takes to make that particular sector appear under the read/write head to the seek time—this time is called the rotational delay, or 'rotational latency'.

The good news is that all modern HDDs (more than 90 per cent of all known brands) use a concept called ZCAV (Zonal Constant Angular Velocity). This takes advantage of the fact that more linear space is available on the outer tracks of the disk platter rather than on the inside tracks. Now since the disk spins at a constant speed, which is also known as CAV (Constant Angular Velocity), the read/write I/O speed will be greater at the outer tracks as compared to the inner tracks. You can use a program called Bonnie++ [*www.coker.com.au/bonnie++*] to check your hard disk.

So the general rule is that you should always create the locations that need frequent I/O—for example, */home* and swap—on the outer tracks. Since generally all HDDs allot partitions from the outside, the easiest way to achieve this is to create these partitions first when partitioning your hard disk.

BUS considerations

Most PC-based systems use the PCI (Peripheral Component Interconnect) bus to connect devices to the processor. The performance of the PCI is determined by the size of data path (bus width) and the speed at which it transfers the data (clock rate). Currently, the PCI supports the 32/64-bit bus width at a speed of 66 MHz.

There is also a parallel interface standard called SCSI (Small Computer System Interface). A typical problem that arises with the SCSI occurs when you place devices of different speeds on the same bus. So a device with a faster throughput may get 'stepped down' to the speed of the slower device on the bus. You can use the *sginfo* tool, part of the *sg3_utils* package, to query or change the parameters of SCSI devices on your machine.

Tuning the sequential read access

The kernel, when reading a file, always tries to take advantage of sequential disk access. 'Read-ahead' is based on the same assumption that if an application is accessing data from Block A, then the chances are more likely that the next Blocks—B, C and D will also need to be read. Therefore, by actually reading ahead and then caching the pages in memory, the kernel improves the I/O and reduces the response time. However, the read-ahead algorithm is designed to get turned off automatically whenever a random read request is detected.

The read-ahead function is based on two values:

1. The current window—the application reads pages from the buffer cache that is its current window.
2. The 'ahead' window—while the application reads pages from the current window, IO happens on the 'ahead' window.

When the application has finished reading from the current window, the 'ahead' window becomes the current window and a new 'ahead' window is started.

To view the current window size, issue *cat > /sys/block/sda/queue/read_ahead_kb*. The following is the command output for my machine:

```
# cat /sys/block/sda/queue/read_ahead_kb
128
```

As you can see, the current and default read-ahead is 128 KB. I can tune it to 256 KB, for example, as follows:

```
# echo 256 > /sys/block/sda/queue/read_ahead_kb
# cat /sys/block/sda/queue/read_ahead_kb
256
```

As is evident, the current window has now become 256 KB. However, note that rebooting the machine will change the default value to 128 KB. Please feel free to use the */etc/rc.local* file to make your changes permanent.

We can also use the *blockdev* command to report read-ahead in sectors. A sector is always 512 bytes in kernel 2.6.x.

```
# blockdev --report /dev/sda
```

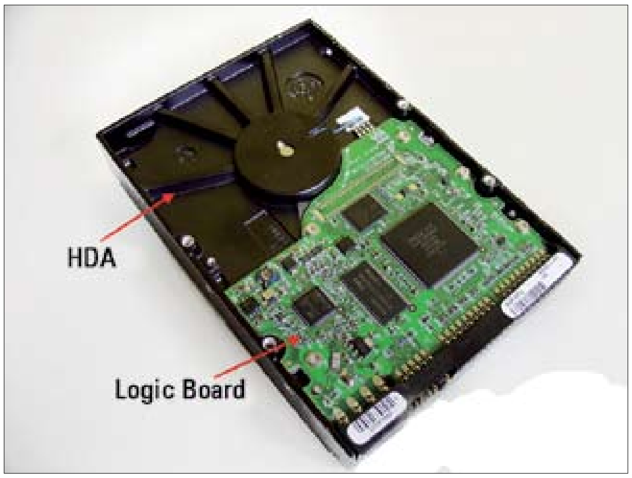

Figure 1: A typical hard disk drive

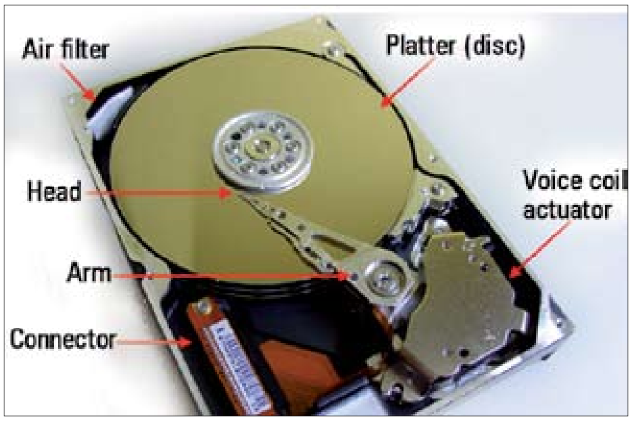

Figure 2: Components of the hard disk

```
RO   RA   SSZ  BSZ  StartSec        Size  Device
rw  256  512  4096         0  320072933376  /dev/sda
```

Remember that if page access occurs in the current window then the size of the new read-ahead windows increases by two pages. So because the default current window's size is 128 KB, the current read-ahead appears as 256. We can use the *blockdev* command with the following options to view and then set our read-ahead.

```
# blockdev --getra /dev/sda
256
# blockdev --setra 512 /dev/sda
# blockdev --getra /dev/sda
512
```

Tuning the disk queue

The I/O subsystem is a series of processes with the responsibility to move blocks of data between the hard disk and the RAM. Generally, every computer task consists of a utility performing either one or both of the following:

- Read a block of data off the disk and move it to RAM
- Write a new block of data from the RAM to disk

These read/write requests are transformed into 'block device requests' that go into a queue. When adding an entry to the queue, the kernel first tries to enlarge an existing request by merging the new request with one that is already in the queue. If the request cannot be merged with an existing one, then the new request will be assigned a position in the queue based on several factors including the elevator algorithm (which we will configure later).

First, let's check what the current queue length is and which current IO scheduler the OS is using. To view the current queue length, issue the following command:

```
# cat /sys/block/sda/queue/nr_requests
128
```

You can see that my current queue length is 128 requests. Now I want to increase it to 170—let's do it using the *echo*:

```
# echo 170 > /sys/block/sda/queue/nr_requests
# cat /sys/block/sda/queue/nr_requests
170
```

To make it permanent, add the entry to the */etc/rc.local* file. [As a rule, keep in mind that anything under */sys* can be made permanent by using */etc/rc.local*.]

Now, let's check which elevator (or I/O scheduler) algorithm the OS is using by default:

```
# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

On my system, it's cfg. Now the question is...

What is an I/O scheduler?

I/O schedulers control the way Linux handles reads and writes to disks. Linux 2.6 includes a number of I/O schedulers—the intention of providing these choices is to allow better optimisation for different classes of workload.

Without an I/O scheduler, Linux follows FIFO (First In, First Out). This could result in massive HDD thrashing—for example, if one process is reading from one part of the disk and one writing to another, the heads would have to seek back and forth across the disk for every operation. The scheduler's main goal is to optimise disk access times.

This is where I/O schedulers come into the picture. Quoting the WlugWiki [*www.wlug.org.nz/LinuxIoScheduler*] an I/O scheduler can use the following techniques to improve performance:

- Request merging—The scheduler merges adjacent requests together to reduce disk seeking.
- Elevator—The scheduler orders requests based on their physical location on the block device, and it basically tries to seek in one direction as much as possible.
- Prioritisation—The scheduler has complete control over how it prioritises requests, and can do so in a number of ways

In addition to these, all I/O schedulers also take into account resource starvation, which ensures requests to be serviced.

Scheduler algorithms operate like elevators in buildings, and are therefore also called elevators. As per *The Red Hat Enterprise Linux 5 IO Tuning Guide:* "The algorithms used to operate real-life building elevators make sure that it services requests per floor, efficiently. To be efficient, the elevator does not travel to each floor depending on which one issued a request to go up or down, first. Instead, it moves in one direction at a time, taking as many requests as it can until it reaches the highest or lowest floor, then does the same in the opposite direction. Simply put, these algorithms schedule disk I/O requests according to which logical block address on the disk they are targeted to. This is because the most efficient way to access the disk is to keep the access pattern as sequential (i.e., moving in one direction) as possible. Sequential, in this case, means 'by increasing the logical block address number'. As such, a disk I/O request targeted for disk block 100 will normally be scheduled before a disk I/O request targeted for disk block 200. This is typically the case, even if the disk I/O request for disk block 200 was issued first."

There are currently four types of schedulers available:

1. The no-op scheduler (noop)
2. The anticipatory IO scheduler (anticipatory)
3. The deadline scheduler (deadline)
4. The Complete Fair Queueing scheduler (CFQ)

The no-op scheduler

As the name suggests, 'no-op' actually does nothing—it simply works in the FIFO manner. No configurable options are available here. The I/O requests are sent in the same manner as they are received, and it is left to the hardware to work on it. The main goal is to conserve CPU cycles.

You can make no-op your IO scheduler as follows:

```
echo noop > /sys/block/sda/queue/scheduler
```

To make it your choice of I/O scheduler across reboots, edit */boot/grub/grub.conf* and append *elevator=noop* at the end of the kernel line:

```
root (hd0,0)
kernel /vmlinuz-2.6.29.6-217.2.16.fc11.i686.PAE ro root=LABEL=Fedora rhgb quiet
elevator=noop
```

The anticipatory scheduler

I personally think this is the most optimistic of the schedulers. It waits in anticipation that the next I/O request might be for the next disk block, before moving to another location. The basic idea is that the IO scheduler will wait for a certain short period of time after catering to one I/O request and before going on to the next I/O request. So if the next I/O request is for the next disk block, it saves time in moving the head back to the same location again. But this may also result in additional latency, if the next I/O is not for the next disk block.

Switch to this scheduler as follows:

```
# echo anticipatory > /sys/block/sda/queue/scheduler
```

Add *elevator=anticipatory* in the kernel line of */boot/grub/grub.conf*, if you want to make it your default.

But how can you tune it as per your needs? Well, The primary tunables are under the */sys/block/sda/queue/iosched/* directory. Some common values that can be tuned are (the values appear automatically under */sys/block/sda/queue/iosched/* as soon as you change your elevator):

- *antic_expire* – Here you can set the maximum amount of time (in milliseconds) you anticipate a good read (one with a short seek distance from the most recently completed request) before giving up. For example, I can change the *antic_expire* from the default 6 ms to 60 ms, as follows:

  ```
  # cat /sys/block/sda/queue/iosched/antic_expire
  6
  # echo 60 > /sys/block/sda/queue/iosched/antic_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  60
  ```

  You can make it permanent by adding the echo line in your */etc/rc.local* file.
- *read_expire* – All the read and write IO requests are processed in batches. So this parameter specifies the time within which the read request must be fulfilled or completed before it expires. The time is specified in milliseconds.

  ```
  # cat /sys/block/sda/queue/iosched/read_expire
  125
  # echo 250 > /sys/block/sda/queue/iosched/read_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  250
  ```
- *write_expire* – they are equivalent to the above, for writes.

  ```
  # cat /sys/block/sda/queue/iosched/write_expire
  250
  # echo 300 > /sys/block/sda/queue/iosched/read_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  300
  ```

The deadline scheduler

As the name suggests, here all the I/O requests have to be fulfilled in a specified amount of time—before they expire. So when an I/O request enters the queue, it is assigned some time (in milliseconds), within which it has to be fulfilled before the time expires, regardless of its targeted block device. The same applies for the write requests. It will also specify how the maximum number of read requests will be fulfilled before switching to a write request.

In order to switch to this scheduler, use the following command:

```
# echo deadline > /sys/block/sda/queue/scheduler
```

To make it permanent, add *elevator=deadline* in the kernel line of */boot/grub/grub.conf*.

And here are its tunables (located under the same */sys/block/sda/queue/iosched/* directory):

- *read_expire* – Here, when a read request first enters the I/O scheduler, it is assigned a deadline that is the current time plus the *read_expire* value, in milliseconds. And, like before, I can change the default value for *read_expire* from 500 ms to 444 ms:

  ```
  # cat /sys/block/sda/queue/iosched/read_expire
  500
  # echo 444 > /sys/block/sda/queue/iosched/read_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  444
  ```
- *write_expire* – As you might have already guessed, its function is the same as *read_expire*, but it is for writes. The time is again in milliseconds.

  ```
  # cat /sys/block/sda/queue/iosched/write_expire
  5000
  # echo 4000 > /sys/block/sda/queue/iosched/read_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  4000
  ```
- *front_merges* – Normally, I/O requests are merged at the bottom of the request queue. This Boolean value controls whether an attempt should be made to merge the request to the top of the request queue. A value of 0 indicates that the front merges are disabled.

  ```
  # cat /sys/block/sda/queue/iosched/front_merges
  1
  ```

The Completely Fair Queue scheduler (CFQ)

The CFS scheduler works in a pure, democratic way. It equally divides all the available bandwidth between all the I/O requests. CFQ uses 64 internal queues to maintain and keep I/O requests. It fills internal queues in a round-robin manner and pulls the I/O requests from these 64 internal queues and places them into a dispatch queue, where they are catered to. Internally, CFQ also changes the order of I/O requests to minimise the head movement.

Most Linux distros come with CFQ as the default I/O scheduler. If not, make it the default as follows:

```
# echo cfq > /sys/block/sda/queue/scheduler
```

Append *elevator=cfq* to the kernel line of */boot/grub/grub.conf* to make the change permanent.

As for the tunables, they are:

- *quantum* – This is the total number of requests to be moved from internal queues to the dispatch queue in each cycle.

  ```
  # cat /sys/block/sda/queue/iosched/quantum
  4
  # echo 8 > /sys/block/sda/queue/iosched/read_expire
  # cat /sys/block/sda/queue/iosched/antic_expire
  8
  ```

  Here I just changed the number of requests to 8 from the default value of 4. Feel free to use */etc/rc.local* to make the changes permanent.
- *queued* – This is the maximum number of requests allowed per internal queue. You can view and change the current value using the *cat* and *echo* commands, respectively, as above.

An introduction to queuing theory

Quoting Wikipedia on the queueing theory, Little's Law states: "The long-term average number of customers in a stable system L is equal to the long-term average arrival rate, λ, multiplied by the long-term average time a customer spends in the system, W, or: L = λ W. Here:

- L = queue length, i.e., the average number of requests waiting in the system
- A = arrival rate, which is the rate at which requests enter a system
- W = wait time, which is the average time to satisfy a request

At a constant arrival rate (λ), if the queue length (L) increases, so will the wait time (W), and if the queue length (L) decreases, so will the wait time (W). So at a constant (λ), L is directly proportional to W.

Wait time (W) consists of queue time (time waiting for a resource to become available, Q) and service time (time for a resource to process a request, S). So we can say, W = Q + S. So Little's Law can be stated as: L = λ (Q + S)

The Utilisation Law is a derivative of Little's Law. It simply states that: U = (service time) x (arrival rate). For a saturated resource U is 1. This means, if the value of U reaches near 1 (or becomes 1), the resource cannot take any more requests.

Now let's check the utilisation of my server hard disk. Run *iostat -x -d /dev/sda 1* to get the values of the average reads per second, the average writes per second and the average service time. I get the following output:

```
Device:  rrqm/s  wrqm/s  r/s  w/s  rsec/s     wsec/s  avgrq-sz  avgqu-
sz  await  svctm  %util
sda          48.00  0.00  86.00  0.00  17760.00  0.00  206.51      0.27
3.12     2.84    24.40
```

From this, I get:

- Average read request (*avgrq-sz*) = 206.51
- Average write request (*wsec/s*) = 0
- Average service time (*svctm*) = 2.84 ms

Now we know that U = S x A (can be remembered as USA, :-)). Hence, U = [(206.51 + 0) x (2.84)] / 1000 = 0.586. So my utilisation is just half way to the saturation level of 1. If this number gets closer to 1 then it's time to change the disk.

We can also find out how many requests per second this hard disk can take. Just make U=1 and find out the A (arrival rate). That is, if 1 = 2.84ms x A, then A = 1000/2.84 = 352.112 requests per second. So this hard disk can take a maximum of 353 requests per second. Anything over this number can cause the machine to fall over.

Finding hot spots in the code

We can use the *strace* tool to trace system calls and signals. *strace* runs a specified command until it exits or ends. It intercepts and records the system calls made by the process and the signals that are received by the process.

For example, let's find out the system calls made by the *who* command:

```
# strace -c who
root    pts/1      2009-09-13 10:37   (172.24.0.123)
alok    :0       2009-09-13 10:38
alok    pts/0      2009-09-13 10:38
alok    pts/1      2009-09-14 00:16
% time    seconds  usecs/call    calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  nan   0.000000        0       20        read
  nan   0.000000        0       4        write
  nan   0.000000        0       20       1 open
  nan   0.000000        0       21        close
  nan   0.000000        0       1        execve
  nan   0.000000        0       1        time
  nan   0.000000        0       45        alarm
  nan   0.000000        0       2       2 access
  nan   0.000000        0       4        kill
  nan   0.000000        0       3        brk
  nan   0.000000        0       4        munmap
  nan   0.000000        0       3        mprotect
  nan   0.000000        0       2        _llseek
  nan   0.000000        0       30        rt_sigaction
  nan   0.000000        0       22        mmap2
  nan   0.000000        0       7       1 stat64
  nan   0.000000        0       20        fstat64
  nan   0.000000        0       31        fcntl64
  nan   0.000000        0       1        set_thread_area
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                241       4 total
```

As you can see, it clearly tells you the name of the system call and how many times it had been called.

Here I conclude for this month. Next time we will learn how to configure a tuneable RAID, using the external journal, and learn about processor scheduling using a locality of reference. END

References and attribution

- RHEL 5 IO Tuning Guide: *http://www.redhat.com/docs/wp/performancetuning/iotuning/index.html*
- Linux I/O Scheduler [WlugWiki]: *www.wlug.org.nz/LinuxIoScheduler*

By: Alok Srivastava

The author is the founder of Network NUTS and holds MCP, MCSE, MCDBA, MCT, CCNA, CCNP, RHCE and RHCSS certifications. Under his leadership, Network NUTS has been a winner of the "Best Red Hat Training Partner in North India" for the last three years in a row. He has also been a trainer for the Indian Air Force, NIC, LIC, IFFCO, Wartsila India Ltd, the government of Rajasthan, Bajaj Allianz, etc. You can reach him at *alok at networknuts dot net.*