4

Performance

As a general discussion, performance is much too broad for a single book, let alone a single chapter. However, in this chapter we narrow the focus of performance to a single subject: I/O on a SCSI bus within a storage area network (SAN). SANs are growing in popularity because they assist with storage consolidation and simplification. The main discussion point within the computing industry with regards to storage consolidation is, as it has always been, performance.

In this chapter, we cover basic concepts of SCSI over Fibre Channel Protocol (FCP) using raw/block device files and volume managers. In addition, we cover block size, multipath I/O drivers, and striping with a volume manager, and we conclude our discussion with filesystem performance and CPU loading. We include examples of each topic throughout the chapter.

Start Troubleshooting at the Lowest Layer Possible

A majority of the time, performance issues are related to I/O. However, assuming that a given performance problem is I/O-based is grossly oversimplifying the problem. With any filesystem I/O, there are middle-layer tasks that require resources which may be the source of an I/O contention, such as the volume manager, the volume manager's striping, the filesystem, a multipath I/O driver, or something similar. When troubleshooting a performance problem, always try to simplify the problem by removing as many middle layers as possible. For example, if a particular filesystem is slow, focus your attention first on the disk block or character device performance before considering the volume manager and filesystem performance.

Dissecting a volume with respect to physical device (aka LUN) or `lvol` into its simplest form is absolutely required when preparing to run any performance test or find a performance concern. In this section, we test the raw speed of a storage device by bypassing the filesystem and volume management layers. We bypass as many layers as possible by using a raw device, better known as a character device. A character device must be bound to a block device through the `raw` command. To describe "raw" with more detail would include the physical access to a block device bypassing the kernel's block buffer cache. Our first test performs a simple sequential read of a Logical Unit Number (LUN), which resides on a set of spindles, through a single path after we bind the block device to the character. We create a (LUN) character device because we want to test the speed of the disk, not the buffer cache.

note

> Today's large arrays define a data storage device in many ways. However, the best description is Logical Device (LDEV). When an LDEV is presented to a host, the device changes names and is referred to as a Logical Unit Number (LUN).

The components used throughout this chapter for examples and scenarios include:

- HP IA64 Superdome (hostname is `atlorca2` in this chapter) running SUSE Linux Enterprise Server 9.0 (SLES 9.0)
- 2 Gig Fibre Channel Emulex LP9802 Host Bus Adapter (HBA)
- McData 6064 Fibre Switch with 2Gbps UPMs
- HP XP128 Storage array with 10K RPM RAID 5

The tools for examining the hardware layout and adding and removing LUNs are discussed in Chapter 5, "Adding New Storage via SAN with Reference to PCMCIA and USB." Performance tools were fully discussed in Chapter 3, "Performance Tools," and are used in examples but not explained in detail in this chapter. As stated previously, this chapter's focus is strictly on performance through a system's I/O SCSI bus connected to SAN. Let's look at how to find and bind a block device to a character device using the `raw` command.

Binding a Raw Device to a Block Device Using the raw Command

The LUN, hereafter called disk, used throughout this example is `/dev/sdj`, also referred to as `/dev/scsi/sdh6-0c0i0l2`. Determine the capacity of the disk through the `fdisk` command:

```
atlorca2:~ # fdisk -l
Disk /dev/sdj: 250.2 GB, 250219069440 bytes
255 heads, 63 sectors/track, 30420 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdj1               1       30421   244354559+  ee  EFI GPT
```

Use `lshw` (an open source tool explained in more detail in Chapter 5) to show the device detail:

```
atlorca2:~ # lshw
~~Focus only on single disk test run~~~~
        *-disk:2
             description: SCSI Disk
             product: OPEN-V*4
             vendor: HP
             physical id: 0.0.2
             bus info: scsi@6.0:0.2
             logical name: /dev/sdj
             version: 2111
             size: 233GB
             capacity: 233GB
             capabilities: 5400rpm ### <- just the drivers attempt to
             guess the speed via the standard scsi lun interface…
             Take this at face value.
             configuration: ansiversion=2
```

Linux does not allow raw access to a storage device by default. To remedy this problem, bind the device's block device file to a /dev/raw/rawX character device file to enable I/O to bypass the host buffer cache and achieve a true measurement of device speed through the host's PCI bus (or other bus architecture). This binding can be done by using the raw command. Look at the block device for /dev/sdj:

```
atlorca2:~ # ls -al /dev/sdj
brw-rw----  1 root disk 8, 144 Jun 30  2004 /dev/sdj
```

Take note of the permissions set on the device file brw-rw----. The b means block device, which has a major number 8, which refers to the particular driver in control. The minor number 144 represents the device's location in the scan plus the partition number. Refer to man pages on sd for more info. Continuing with our example, the next step is to bind the /dev/sdj to a raw character device, as depicted in the following:

```
atlorca2:~ # raw /dev/raw/raw8 /dev/sdj
/dev/raw/raw8:  bound to major 8, minor 144
```

Now, issue one of the following commands to view the binding parameters:

```
atlorca2:~ # raw -qa
/dev/raw/raw8:  bound to major 8, minor 144
```

or

```
atlorca2:~ # raw -q /dev/raw/raw8
/dev/raw/raw8:  bound to major 8, minor 144
```

Raw Device Performance

Now that we have bound a block device to a character device, we can measure a read by bypassing the block device, which in turn bypasses the host buffer cache. Recall that our primary objective is to measure performance from the storage device, not from our host cache.

disclaimer

Throughout this chapter, we use a sequential read test provided by the `dd` command. By no means are we implying that this is the best performance benchmark tool. Every array has certain strengths with regards to I/O size and patterns of sequential versus random, so for benchmarks, we like IOzone (refer to http://www.iozone.org) for a nice benchmark tool. One must be aware that some arrays suffer during heavy, large sequential reads, whereas other arrays thrive in these situations, and the same is true for particular HBAs. In addition, those arrays that suffer on sequential read/writes usually excel at random read/writes, whereas the reverse can be said about the arrays that perform well under sequential read/write operations. In addition, `hdparm` impacts performance with respect to direct memory access (DMA) and read-ahead along with other parameters. In this chapter, we use the default settings of `hdparm`. So to reiterate, this chapter uses a simple sequential read/write test to help isolate a performance problem by comparison, and many examples are illustrated.

Our next step requires that we measure a sequential read and calculate time required for the predetermined data allotment. Throughout this chapter, our goal is to determine what factors dictate proper performance of a given device. We focus on average service time, reads per second, writes per second, read sectors per second, average request size, average queue size, and average wait time to evaluate performance. In addition, we discuss the I/Os per second with regard to payload "block" size. For now, we start with a simple sequential read to get our baseline.

Though a filesystem may reside on the device in question, as shown previously in the `fdisk-l` output, the filesystem cannot be mounted for the test run. If mounted, raw access is denied. Proceed with the following action as illustrated in the next section.

Using the dd Command to Determine Sequential I/O Speed

The `dd` command provides a simple way to measure sequential I/O performance. The following shows a sequential read of 1GB (1024MB). There are 1024 1MB (1024KB) reads:

```
atlorca2:~ # time -p dd if=/dev/raw/raw8 of=/dev/null bs=1024k
count=1024
```

```
1024+0 records in
1024+0 records out
real 6.77
user 0.00
sys  0.04
```

The megabytes per second can be calculated as follows:

```
1GB/6.77 sec = 151.25MBps
```

For those who are unfamiliar with high-speed enterprise servers and disk storage arrays, 151MBps may seem extremely fast. However, higher speeds can be achieved with proper striping and tuning of the filesystem across multiple spindles. Though we discuss some of those tuning options later, we first need to reduce the previous test to its simplest form. Let us begin with calculating MBps, proceeding with the blocking factors on each I/O frame and discussing service time for each round trip for a given I/O.

In the previous example, we saw 1024 I/Os, where each I/O is defined to have a boundary set to a block size of 1MB, thanks to the `bs` option on the `dd` command. Calculating MBps simply takes an arithmetic quotient of 1024MB/6.77 seconds, providing a speedy 151MB/sec. In our testing, cache on the array is clear, providing a nice 151MBps, which is not bad for a single LUN/LDEV on a single path. However, determining whether the bus was saturated and whether the service time for each I/O was within specifications are valid concerns. Each question requires more scrutiny.

note

> Different arrays require special tools to confirm that cache within the array is flushed so that a true spindle read is measured. For example, HP's largest storage arrays can have well over 100GB of cache on the controller in which a read/write may be responding, thereby appearing to provide higher average reads/writes than the spindle can truly provide. Minimum cache space should be configured when running performance measurements with respect to design layout.

Using sar and iostat to Measure Disk Performance

Continuing with the `dd` command, we repeat the test but focus only on the data yielded by the `sar` command to depict service time and other traits, as per the following.

```
atlorca2:~ # sar -d 1 100
Linux 2.6.5-7.97-default (atlorca2)     05/09/05

14:19:23          DEV       tps  rd_sec/s  wr_sec/s
14:19:48     dev8-144      0.00      0.00      0.00
14:19:49     dev8-144      0.00      0.00      0.00
14:19:50     dev8-144    178.00 182272.00      0.00
14:19:51     dev8-144    303.00 311296.00      0.00
14:19:52     dev8-144    300.00 307200.00      0.00
14:19:53     dev8-144    303.00 309248.00      0.00
14:19:54     dev8-144    301.00 309248.00      0.00
14:19:55     dev8-144    303.00 311296.00      0.00
14:19:56     dev8-144    302.00 309248.00      0.00
```

This `sar` output shows that the total number of transfers per second (`TPS`) holds around 300. `rd_sec/s` measures the number of read sectors per second, and each sector is 512 bytes. Divide the `rd_sec/s` by the `tps`, and you have the number of sectors in each transfer. In this case, the average is 1024 sectors at 512 bytes each. This puts the average SCSI block size at 512KB. This is a very important discovery; because the `dd` command requests a block size of 1MB, the SCSI driver blocks the request into 512 byte blocks, so two physical I/Os complete for every logical I/O requested. Different operating systems have this value hard coded at the SCSI driver at different block sizes, so be aware of this issue when troubleshooting.

As always, more than one way exists to capture I/O stats. In this case, `iostat` may suit your needs. This example uses `iostat` rather than `sar` to evaluate the `dd` run.

```
atlorca2:~ # iostat
Linux 2.6.5-7.97-default (atlorca2)     05/09/05

avg-cpu:  %user   %nice    %sys %iowait   %idle
           0.00    0.01    0.02    0.08   99.89

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
```

```
sdj              0.06        57.76        0.00   15222123       72
sdj              0.00         0.00        0.00          0        0
sdj              0.00         0.00        0.00          0        0
sdj             98.00    102400.00        0.00     102400        0
sdj            298.00    305152.00        0.00     305152        0
sdj            303.00    309248.00        0.00     309248        0
sdj            303.00    311296.00        0.00     311296        0
sdj            301.00    307200.00        0.00     307200        0
sdj            302.00    309248.00        0.00     309248        0
sdj            302.00    309248.00        0.00     309248        0
sdj            141.00    143360.00        0.00     143360        0
```

Calculating MBps from `iostat` can be achieved by calculating KB from blocks read per second (`Blk_read/s`) and multiplying them by the transactions per second (TPS). In the previous example, 311296 `Blk_read/s` / (303 `tps`) = 1027.3 blocks × 512 bytes/block = 526018 bytes / 1024 bytes/KB = 513KB avg.

Before we explain the importance of the blocking size based on a given driver, let us demonstrate the same test results with a different block size. Again, we move 1GB of data through a raw character device using a much smaller block size. It is very important to understand that the exact same 1GB of data is being read by `dd` and written to `/dev/null`.

Understanding the Importance of I/O Block Size When Testing Performance

The I/O block size can impact performance. By reducing the `dd` read block size from 1024k to 2k, the FCP payload of 2k and the SCSI disk (`sd`) driver can deliver about 1/16 of the performance. Additionally, the I/O rate increases dramatically as the block size of each request drops to that of the FCP limit. In the first example, the `sd` driver was blocking on 512k, which put the I/O rate around 300 per second. In the world of speed, 300 I/O per second is rather dismal; however, we must keep that number in perspective because we were moving large data files at over 100 MBps. Though the I/O rate was low, the MBps was enormous.

Most applications use an 8K block size. In the following demonstration, we use a 2K block size to illustrate the impact of I/O payload (I/O size).

```
atlorca2:~ # time -p dd if=/dev/raw/raw8 of=/dev/null bs=2k \
 count=524288
524288+0 records in
524288+0 records out
real 95.98
user 0.29
sys 8.78
```

You can easily see that by simply changing the block size of a data stream from 1024k to 2k, the time it takes to move large amounts of data changes drastically. The time to transfer 1GB of data has increased 13 times from less than 7 seconds to almost 96 seconds, which should highlight the importance of block size to any bean counter.

We can use `sar` to determine the average I/O size (payload).

```
atlorca2:~ # sar -d 1 100¦ grep dev8-144
14:46:50     dev8-144   5458.00  21832.00      0.00
14:46:51     dev8-144   5478.00  21912.00      0.00
14:46:52     dev8-144   5446.00  21784.00      0.00
14:46:53     dev8-144   5445.00  21780.00      0.00
14:46:54     dev8-144   5464.00  21856.00      0.00
14:46:55     dev8-144   5475.00  21900.00      0.00
14:46:56     dev8-144   5481.00  21924.00      0.00
14:46:57     dev8-144   5467.00  21868.00      0.00
```

From the `sar` output, we can determine that 21868 `rd_sec/s` transpires, while we incur a `tps` of 5467. The quotient of 21868/5467 provides four sectors in a transaction, which equates to 2048 bytes, or 2K. This calculation shows that we are moving much smaller chunks of data but at an extremely high I/O rate of 5500 I/O per second. Circumstances do exist where I/O rates are the sole concern, as with the access rates of a company's Web site. However, changing perspective from something as simple as a Web transaction to backing up the entire corporate database puts sharp focus on the fact that block size matters. Remember, backup utilities use large block I/O, usually 64k.

With the understanding that small block I/O impedes large data movements, note that filesystem fragmentation and sparse file fragmentation can cause an application's request to be broken into very small I/O. In other words, even though a `dd if=/file system/file_name of=/tmp/out_file bs=128k` is requesting a read with 128k block

I/O, sparse file or filesystem fragmentation can force the read to be broken into much smaller block sizes. So, as we continue to dive into performance troubleshooting throughout this chapter, always stay focused on the type of measurement needed: I/O, payload, or block size. In addition to considering I/O, payload, and block size, time is an important factor.

Importance of Time

Continuing with our example, we must focus on I/O round-trip time and bus saturation using the same performance test as earlier. In the next few examples, we use `iostat` to illustrate average wait time, service time, and percent of utilization of our test device.

The following `iostat` display is from the previous sequential read test but with block size set to 4096k, or 4MBs, illustrating time usage and device saturation.

```
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=4096k &

atlorca2:~ # iostat -t -d -x 1 100

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

sdj          0.00   0.00 308.00  0.00 319488.00    0.00 159744.00     \
0.00  1037.30     4.48   14.49   3.24  99.80
sdj          0.00   0.00 311.00  0.00 319488.00    0.00 159744.00     \
0.00  1027.29     4.53   14.66   3.21  99.70
```

In this `iostat` output, we see that the device utilization is pegged at 100%. When device utilization reaches 100%, device saturation has been achieved. This value indicates not only saturation but also the percentage of CPU time for which an I/O request was issued. In addition to eating up CPU cycles with pending I/O waits, notice that the round-trip time (service time) required for each I/O request increased.

Service time, the time required for a request to be completed on any given device, holds around 3.2ms. Before we go into detail about all the items that must be completed within that 3.2ms, which are discussed later in this chapter, we need to recap the initial test parameters. Recall that the previous `iostat` data was collected while using the `dd` command with a block size of 4096k. Running the same test with block size set to 1024k yields identical block counts in `iostat`, as you can see in this example:

```
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=1024k &

atlorca2:~ # iostat -t -d -x 1 100

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

sdj          0.00   0.00 303.00  0.00 309248.00    0.00 154624.00     \
0.00  1020.62     1.51    4.96   3.29  99.80
sdj          0.00   0.00 303.00  0.00 311296.00    0.00 155648.00     \
0.00  1027.38     1.51    4.98   3.29  99.70
sdj          0.00   0.00 304.00  0.00 311296.00    0.00 155648.00     \
0.00  1024.00     1.50    4.95   3.26  99.00
sdj          0.00   0.00 303.00  0.00 309248.00    0.00 154624.00     \
0.00  1020.62     1.50    4.93   3.28  99.40
sdj          0.00   0.00 304.00  0.00 311296.00    0.00 155648.00     \
0.00  1024.00     1.50    4.93   3.28  99.60
```

Determining Block Size

As we have illustrated earlier in this chapter, block size greatly impacts an application's overall performance. However, there are limits that must be understood concerning who has control over the I/O boundary. Every application has the capability to set its own I/O block request size, but the key is to understand the limits and locations. In Linux, excluding applications and filesystems, the `sd` driver blocks all I/O on the largest block depending on medium (such as SCSI LVD or FCP). An I/O operation on the SCSI bus with any typical SCSI RAID controller (not passing any other port drivers, such as Qlogic, or Emulex FCP HBA) holds around 128KB. However, in our case, through FCP, the largest block size is set to 512KB, as shown in the previous example when doing a raw sequential read access through `dd`. However, it goes without saying that other factors have influence, as shown later in this chapter when additional middle layer drivers are installed for I/O manipulation.

To determine the maximum blocking factor, or max I/O size, of a request at the SD/FCP layer through a raw sequential read access, we must focus on the following items captured by the previous `dd` request in `iostat` examples.

The following example explains how to calculate block size.

```
Device:          tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn

sdj           303.00    311296.00         0.00     311296          0
```

As the output shows, the number of blocks read per second is 311296.00, and the number of transactions per second is 303.00.

(sectors read/sec)/(read/sec) =~ 1024 sectors

note

=~ means approximation.

Recall that a sector has 512 bytes.

(1024 sectors) × (512 bytes/sector) = 524288 bytes

Now convert the value to KB.

(524288 bytes) / (1024 bytes/KB) = 512KB

Another way to calculate the block size of an I/O request is to simply look at the `avgrq-sz data` from `iostat`. This field depicts the average number of sectors requested in a given I/O request, which in turn only needs to be multiplied by 512 bytes to yield the block I/O request size in bytes.

Now that we have demonstrated how to calculate the in-route block size on any given I/O request, we need to return to our previous discussion about round-trip time and follow up with queue length.

Importance of a Queue

Service time only includes the amount of time required for a device to complete the request given to it. It is important to keep an eye on `svctm` so that any latency with respect to the end device can be noted quickly and separated from the average wait time. The average wait time (await) is not only the amount of time required to service the I/O at the device but also the amount of wait time spent in the dispatch queue and the round-trip time. It is important to keep track of both times because the difference between the two can help identify problems with the local host.

To wrap things up with I/O time and queues, we need to touch on queue length.. If you are familiar with C programming, you may find it useful to look at how these values are calculated. The following depicts the calculation for average queue length and wait time found in iostat source code.

```
nr_ios = sdev.rd_ios + sdev.wr_ios;
tput   = ((double) nr_ios) * HZ / itv;
util   = ((double) sdev.tot_ticks) / itv * HZ;
svctm  = tput ? util / tput : 0.0;
/*
 * kernel gives ticks already in milliseconds for all platforms
 * => no need for further scaling.
 */
await  = nr_ios ?
   (sdev.rd_ticks + sdev.wr_ticks) / nr_ios : 0.0;
arqsz  = nr_ios ?
   (sdev.rd_sectors + sdev.wr_sectors) / nr_ios : 0.0;

printf("%-10s", st_hdr_iodev_i->name);
if (strlen(st_hdr_iodev_i->name) > 10)
   printf("\n          ");
/*       rrq/s wrq/s   r/s   w/s  rsec  wsec   rkB   wkB  \
         rqsz  qusz await svctm %util */
printf(" %6.2f %6.2f %5.2f %5.2f %7.2f %7.2f %8.2f %8.2f \
         %8.2f %8.2f %7.2f %6.2f %6.2f\n",
       ((double) sdev.rd_merges) / itv * HZ,
       ((double) sdev.wr_merges) / itv * HZ,
       ((double) sdev.rd_ios) / itv * HZ,
       ((double) sdev.wr_ios) / itv * HZ,
       ((double) sdev.rd_sectors) / itv * HZ,
```

```
        ((double) sdev.wr_sectors) / itv * HZ,
        ((double) sdev.rd_sectors) / itv * HZ / 2,
        ((double) sdev.wr_sectors) / itv * HZ / 2,
        arqsz,
        ((double) sdev.rq_ticks) / itv * HZ / 1000.0,
        await,
        /* The ticks output is biased to output 1000 ticks per second */
        svctm,
        /* Again: ticks in milliseconds */
        util / 10.0);
```

Though it is nice to understand the calculations behind every value provided in performance tools, the most important thing to recall is that a large number of outstanding I/O requests on any given bus is not desirable when faced with performance concerns.

In the following `iostat` example, we use an I/O request size of 2K, which results in low service time and queue length but high disk utilization.

```
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &

atlorca2:~ # iostat -t -d -x 1 100

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

sdj          0.00   0.00 5492.00  0.00 21968.00    0.00 10984.00     \
0.00     4.00     0.97    0.18   0.18  96.70
sdj          0.00   0.00 5467.00  0.00 21868.00    0.00 10934.00     \
0.00     4.00     0.95    0.17   0.17  94.80
sdj          0.00   0.00 5413.00  0.00 21652.00    0.00 10826.00     \
0.00     4.00     0.96    0.18   0.18  96.40
sdj          0.00   0.00 5453.00  0.00 21812.00    0.00 10906.00     \
0.00     4.00     0.98    0.18   0.18  97.80
sdj          0.00   0.00 5440.00  0.00 21760.00    0.00 10880.00     \
0.00     4.00     0.97    0.18   0.18  96.60
```

Notice how the `%util` remains high, while the request size falls to 4 sectors/(I/O), which equals our 2048-byte block size. In addition, the average queue size remains small, and wait time is negligible along with service time. Recall that wait time includes round-trip time, as discussed previously. Now that we have low values for `avgrq-sz`, `avgqu-sz`, `await`, and `svctm`, we must decide whether we have a performance problem. In this example, the answer is both yes and no. Yes, the device is at its peak performance for a single thread data query, and no, the results for the fields typically focused on to find performance concerns are not high.

Multiple Threads (Processes) of I/O to a Disk

Now that we have covered the basics, let us address a multiple read request to a device.

In the following example, we proceed with the same block size, 2K, as discussed previously; however, we spawn a total of six read threads to the given device to illustrate how service time, queue length, and wait time differ. Let's run six `dd` commands at the same time.

```
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
atlorca2:~ # dd if=/dev/raw/raw8 of=/dev/null bs=2k &
```

Note that the previous code can be performed in a simple `for` loop:

```
for I in 1 2 3 4 5 6
do
dd if=/dev/raw/raw8 of=/dev/null bs=2k &
done
```

Let's use `iostat` again to look at the `dd` performance.

```
atlorca2:~ # iostat -t -d -x 1 100
```

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

sdj          0.00   0.00 5070.00  0.00 20280.00    0.00 10140.00     \
0.00     4.00     4.96    0.98   0.20 100.00
sdj          0.00   0.00 5097.00  0.00 20388.00    0.00 10194.00     \
0.00     4.00     4.97    0.98   0.20 100.00
sdj          0.00   0.00 5103.00  0.00 20412.00    0.00 10206.00     \
0.00     4.00     4.97    0.97   0.20 100.00
```

The queue length (`avgqu-sz`) is 4.97, while the max block request size holds constant. The service time for the device to act on the request remains at 0.20ms. Furthermore, the average wait time has increased to 0.98ms due to the device's response to multiple simultaneous I/O requests requiring a longer round-trip time. It is useful to keep the following example handy when working with a large multithreaded performance problem because the device may be strained, and striping at a volume manager level across multiple devices would help relieve this type of strain.

Using a Striped lvol to Reduce Disk I/O Strain

To illustrate the reduction of strain, let us create a VG and 4000MB `lvol` striped across two disks with a 16k stripe size.

```
atlorca2:/home/greg/sysstat-5.0.6 # pvcreate /dev/sdi
  No physical volume label read from /dev/sdi
  Physical volume "/dev/sdi" successfully created

atlorca2:/home/greg/sysstat-5.0.6 # pvcreate /dev/sdj
  No physical volume label read from /dev/sdj
  Physical volume "/dev/sdj" successfully created

atlorca2:/home/greg/sysstat-5.0.6 # vgcreate vg00 /dev/sdi /dev/sdj
  Volume group "vg00" successfully created
```

```
atlorca2:/home/greg/sysstat-5.0.6 # lvcreate -L 4000m -i 2 -I 16 -n \
lvol1 vg00
  Logical volume "lvol1" created

atlorca2:/home/greg/sysstat-5.0.6 # lvdisplay -v /dev/vg00/lvol1
    Using logical volume(s) on command line
  ------ Logical volume ------
  LV Name                /dev/vg00/lvol1
  VG Name                vg00
  LV UUID                UQB5AO-dp8Z-N0ce-Dbd9-9ZEs-ccB5-zG7fsF
  LV Write Access        read/write
  LV Status              available
  # open                 0
  LV Size                3.91 GB
  Current LE             1000
  Segments               1
  Allocation             next free (default)
  Read ahead sectors     0
  Block device           253:0
```

We again use sequential 2k reads with dd to measure the performance of the disks.

```
atlorca2:/home/greg/sysstat-5.0.6 # raw /dev/raw/raw9 /dev/vg00/lvol1 \
/dev/raw/raw9: bound to major 253, minor 0
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
atlorca2:/home/greg/sysstat-5.0.6 # dd if=/dev/raw/raw9 of=/dev/null \
bs=2k &
```

Note that the previous command can be performed in a simple `for` loop, as previously illustrated. Again we use `iostat` to measure disk throughput.

```
atlorca2:/home/greg # iostat -x 1 1000

avg-cpu:  %user   %nice    %sys %iowait   %idle
           0.01    0.01    0.03    0.11   99.84
Device:    rrqm/s wrqm/s   r/s    w/s  rsec/s  wsec/s    rkB/s    \
wkB/s avgrq-sz avgqu-sz   await  svctm  %util

sdi          0.00   0.00 2387.00  0.00 9532.00    0.00  4766.00     \
0.00     3.99     3.04     1.28   0.42 100.00
sdj          0.00   0.00 2380.00  0.00 9536.00    0.00  4768.00     \
0.00     4.01     2.92     1.22   0.42 100.00
sdi          0.00   0.00 2318.00  0.00 9288.00    0.00  4644.00     \
0.00     4.01     3.14     1.35   0.43  99.70
sdj          0.00   0.00 2330.00  0.00 9304.00    0.00  4652.00     \
0.00     3.99     2.82     1.21   0.43  99.50
```

Notice that the average wait time per I/O and the service time have increased slightly in this example. However, the average queue has been cut almost in half, as well as the physical I/O demand on the device `sdj`. The result is similar to a seesaw effect: As one attribute drops, another rises. In the previous scenario, the LUN (`sdj`) is physically composed of multiple physical mechanisms in the array called (array group), which remains a hidden attribute to the OS. By using the LVM strategy, we reduce some of the contingency for one LUN or array group to handle the entire load needed by the device (`lvol`). With the previous demonstration, you can see the advantages of striping, as well as its weaknesses. It seems true here that, for every action, there is an equal and opposite reaction.

Striped lvol Versus Single Disk Performance

In the following example, we compare a striped raw `lvol` to a raw single disk. Our objective is to watch the wait time remain almost constant, while the queue size is cut almost in half when using a `lvol` stripe instead of a single disk.

First let's look at performance using the lvol. In this example, we start six dd commands that perform sequential reads with block size set to 512k. The dd commands run at the same time and read a raw device bound to the lvol (as illustrated previously) with a 16k stripe size. Remember, iostat shows two disk devices for our lvol test because the lvol is striped across two disks. At 512KB, iostat yields values as follows:

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

sdi          0.00   0.00 152.00  0.00 156672.00    0.00 78336.00     \
0.00  1030.74     3.00   19.60   6.58 100.00
sdj          0.00   0.00 153.00  0.00 155648.00    0.00 77824.00     \
0.00  1017.31     2.99   19.69   6.54 100.00

sdi          0.00   0.00 154.00  0.00 157696.00    0.00 78848.00     \
0.00  1024.00     2.98   19.43   6.49 100.00
sdj          0.00   0.00 154.00  0.00 157696.00    0.00 78848.00     \
0.00  1024.00     3.01   19.42   6.49 100.00
```

Notice that the I/O queue length when reading lvol1 is much shorter than the following identical dd sequential read test on a raw disk sdj as shown next. Though the identical blocking size of 512k is used, the service time decreases. Here are the results of the test with a raw disk.

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

sdj          0.00   0.00 311.00  0.00 318464.00    0.00 159232.00     \

0.00  1024.00     5.99   19.30   3.22 100.00
sdj          0.00   0.00 310.00  0.00 317440.00    0.00 158720.00     \
0.00  1024.00     5.99   19.31   3.23 100.00
sdj          0.00   0.00 311.00  0.00 318464.00    0.00 159232.00     \
0.00  1024.00     5.99   19.26   3.22 100.00
```

The raw device, sdj in the test using lvol1, reflects that the read requests per second (r/s) remain constant (152 on disk device sdi and 153 on disk device sdj), yielding a net result of 305 read requests per second. The `lvol` test also shows an improvement in the average wait time; however, we hurt the service time. The service time for the `lvol` test is about 6.5ms, whereas it is 3.2ms for the raw disk. Upon closer inspection, we notice that the service time is higher due to the I/O issued to the device. In the `lvol` example, we have in fact submitted 512KB every other time (because we are striping and blocking our I/O both on 512k) so that each total I/O submitted to the device is in a smaller queue, thereby reducing the wait time to be serviced. However, in the single device example, the queue wait time is high because we are waiting on the device to finish on the given I/O request, so with no overhead, the return is faster for the service. This example illustrates the seesaw effect discussed previously, in which a device (single LUN or `lvol`) is slammed, in which case the end user would need to address the application's need to perform such heavy I/O with a single device. In the previous example, tweaking the device or `lvol` buys no performance gain; it just moves the time wait status to another field.

note

With a wider stripe, some performance would be gained in the previous sequential I/O example, but it is unrealistic in the real world. In addition to adding more disks for a wider stripe, you could add more paths to the storage for multipath I/O. However, multipath I/O comes with its own list of constraints.

Multipath I/O

Many administrators have heard about load balance drivers, which allow disk access through multiple paths. However, very few multipath I/O drivers provide load balance behavior to I/Os across multiple HBA paths as found in enterprise UNIX environments. For example, device drivers such as MD, Autopath, Secure Path (`spmgr`), and Qlogic's secure path are dedicated primarily to providing an alternate path for a given disk. Though HP's Secpath does offer a true load balance policy for EVA HSG storage on Linux, all the other drivers mentioned only offer failover at this time.

The one true load balancing driver for Linux (HP's Secure Path) provides a round robin (RR) load balance scheduling policy for storage devices on EVA and HSG arrays. Unfortunately, just because a driver that provides load balancing, such as the HP Secure

Path driver, exists does not mean support is available for your system. Support for array types is limited. Review your vendor's storage requirements and device driver's hardware support list before making any decisions about which driver to purchase. Keeping in mind that restrictions *always* exist, let's review a typical RR policy and its advantages and disadvantages.

Though we want to discuss load balancing, the vast majority of Linux enterprise environments today use static (also known as "manual") load balancing or preferred path. With this in mind, we keep the discussion of RR to a minimum.

In the next example, we proceed with a new host and new array that will allow the RR scheduling policy.

The following example illustrates RR through Secure Path on Linux connected through Qlogic HBAs to an EVA storage array. Due to configuration layout, we use a different host for this example.

```
[root@linny5 swsp]# uname -a
Linux linny5.cxo.hp.com 2.4.21-27.ELsmp #1 SMP Wed Dec 1 21:59:02 EST \
2004 i686 i686 i386 GNU/Linux
```

Our host has two HBAs, `/proc/scsi/qla2300/0` and `/proc/scsi/qla2300/1`, with Secure Path version 3.0cFullUpdate-4.0.SP, shown next.

```
[root@linny5 /]# cat /proc/scsi/qla2300/0
QLogic PCI to Fibre Channel Host Adapter for QLA2340:
        Firmware version:  3.03.01, Driver version 7.01.01
Entry address = f88dc060
HBA: QLA2312 , Serial# G8762

[root@linny5 swsp]# cat /etc/redhat-release
Red Hat Enterprise Linux AS release 3 (Taroon Update 3)
```

Continuing with our raw device testing, we must bind the block device to the character device.

```
[root@linny5 swsp]# raw /dev/raw/raw8 /dev/spdev/spd
```

Use the Secure Path command `spmgr` to display the product's configuration.

```
[root@linny5 swsp]# spmgr display
  Server:  linny5.cxo.hp.com    Report Created: Tue, May 10 19:10:04 2005
  Command: spmgr display
  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
  Storage:  5000-1FE1-5003-1280
  Load Balance: Off  Auto-restore: Off
  Path Verify: On    Verify Interval: 30
  HBAs: 2300-0  2300-1
  Controller:  P66C5E1AAQ20AL, Operational
               P66C5E1AAQ20AD, Operational
  Devices:  spa  spb  spc  spd
```

To reduce space needed for this example, a large part of the spmgr display has been truncated, and we focus only on device spd, as per the following:

```
  TGT/LUN    Device          WWLUN_ID                               #_Paths
    0/  3    spd             6005-08B4-0010-056A-0000-9000-0025-0000    4

        Controller  Path_Instance    HBA         Preferred?  Path_Status
          P66C5E1AAQ20AL                                  YES
                  hsx_mod-0-0-0-4    2300-0        no           Available
                  hsx_mod-1-0-2-4    2300-1        no           Active

       Controller  Path_Instance      HBA         Preferred?  Path_Status
          P66C5E1AAQ20AD                                  no
                    hsx_mod-0-0-1-4    2300-0        no           Standby
                    hsx_mod-1-0-1-4    2300-1        no           Standby
```

Notice that two HBAs and two controllers are displayed. In this case, the EVA storage controller P66C5E1AAQ20AL has been set to preferred active on this particular LUN, in which both of the fabric N_ports enable connection to the fabric. In this configuration, each N_Port connects to different fabrics, A and B, which are seen by Qlogic 0 and 1. In addition, each HBA also sees the alternate controller in case a failure occurs on the selected preferred controller.

We should also to mention that not all arrays are Active/Active on all paths for any LUN at any given time. In this case, the EVA storage array is an Active/Active array because both N_Ports on any given controller have the capability to service an I/O. However, any one LUN can only access a single N_Port at any moment, while another LUN can access the alternate port or alternate controller. Now that we have a background in EVA storage, we need to discuss how the worldwide name (WWN) of a given target device can be found. In the following illustration, we simply read the content of the device instance for the filter driver swsp.

```
[root@linny5 swsp]#  cat /proc/scsi/swsp/2
swsp LUN information:

Array WWID:     50001FE150031280
```

Next, we initiate load balancing and start our raw device test, which is identical to the test performed earlier in this chapter.

```
[root@linny5 swsp]# spmgr set -b on 50001FE150031280

[root@linny5 swsp]# dd if=/dev/raw/raw8 of=/dev/null bs=512k
```

While this simple test runs, we collect iostat-x-d1 100 measurements, and we collect a few time captures.

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

sdd          0.00   0.00 5008.00  0.00 319488.00    0.00 159744.00    \
0.00    63.80     9.81    1.96   0.19  93.00
sdd          0.00   0.00 4992.00  0.00 320512.00    0.00 160256.00    \
0.00    64.21     9.96    1.96   0.18  91.00
sdd          0.00   0.00 4992.00  0.00 318464.00    0.00 159232.00    \
0.00    63.79     9.80    2.00   0.18  92.00
sdd          0.00   0.00 4992.00  0.00 319488.00    0.00 159744.00    \
0.00    64.00     9.89    1.98   0.19  95.00
```

Notice that the blocking factor for a given I/O has changed to 64 sectors per I/O, which equals 32k block size from the swsp module. To get a good comparison between a sequential read test with RR enabled and one with RR disabled, we must disable load balance and rerun the same test. We disable load balancing in the following example.

```
[root@linny5 swsp]# spmgr set -b off 50001FE150031280

[root@linny5 swsp]# dd if=/dev/raw/raw8 of=/dev/null bs=512k

[root@linny5 swsp]# iostat -x 1 100

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

sdd          0.00   0.00 4718.00  0.00 302080.00    0.00 151040.00    \
0.00    64.03     9.48    2.01   0.21  98.00
sdd          0.00   0.00 4710.00  0.00 302080.00    0.00 151040.00    \
0.00    64.14     9.61    2.02   0.20  94.00
sdd          0.00   0.00 4716.00  0.00 302080.00    0.00 151040.00    \
0.00    64.05     9.03    1.91   0.20  95.00
sdd          0.00   0.00 4710.00  0.00 301056.00    0.00 150528.00    \
0.00    63.92     8.23    1.76   0.20  96.00
```

`Iostat` reports that the block size remains constant and that the average wait time for a given I/O round trip is slightly higher. This makes sense now that all I/O is on a single path. Because more I/O is loaded on a single path, average wait time increases, as do service times. Now that we have drawn a quick comparison between `spmgr` being enabled and disabled on a sequential read, we need to recap the advantages seen thus far.

In the previous example, no performance gain was seen by enabling load balancing with regard to `spmgr`. As we can see, no obvious performance increase was seen when RR was enabled through the host measurements. However, though the host's overall benefit from enabling load balancing was insignificant, the SAN load was cut in half. Keep in mind that a simple modification can impact the entire environment, even outside the host. Something as minor as having a static load balance with a volume manager strip across multiple paths or having a filter driver automate the loading of paths can have a large impact on the overall scheme.

Finally, with respect to load balance drivers, it is important to watch for the max block size for any given transfer. As seen in the previous `iostat` examples, the Secure Path product reduces the block size to 32k, and if LVM were to be added on top of that, the block would go to 16K. This small block transfer is great for running small, block-heavy I/O traffic, but for large data pulls, it can become a bottleneck.

Filesystems

Application data access through a filesystem is much more common than access through raw storage. Filesystem meta structures are maintained by the filesystem's driver, removing the overhead from the application. For this reason, very few applications are written to perform raw I/O, except for a few database systems whose creators believe they can maintain the integrity or performance better than a standardized filesystem. This section addresses performance characteristics with regards to multiple filesystems and draws a comparison to the previous section on raw device access.

The configuration for this section is identical to the previous section. We use the IA64 host `atlorca2`. Filesystem types used for comparisons are `xfs` and `ext3`. To begin, we define a filesystem using the same disks from previous examples. By creating a filesystem, we simply add an additional layer to the data management overhead.

Journaling to a Separate Disk

The following `fdisk` output shows device `sdj` with one primary partition of type `83` known as Linux native. The objective is to compare performance of a large sequential read. We compare performance with raw to performance with XFS, illustrating performance overhead.

The first step in comparing performance between raw and filesystem is to set a baseline. Here we set a performance baseline for a single threaded read on a raw device named `/dev/sdj`.

```
atlorca2:~ # fdisk -l /dev/sdj

Disk /dev/sdj: 250.2 GB, 250219069440 bytes
255 heads, 63 sectors/track, 30420 cylinders
```

```
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdj1               1       30420   244348618+  83  Linux
```

The fdisk output shows that partition 1 is active, with 250GB of capacity.

The XFS filesystem type offers the performance feature of creating a journal log on a disk separate from the filesystem data. Next we demonstrate this feature and test it. In this example, device sdk is the journal log device, and sdj is used for the metadata device.

```
atlorca2:~ # mkfs.xfs -f -l logdev=/dev/sdk1,size=10000b /dev/sdj1

meta-data=/dev/sdj1              isize=256    agcount=16, \
agsize=3817947 blks
         =                       sectsz=512
data     =                       bsize=4096   blocks=61087152, \
imaxpct=25
         =                       sunit=0      swidth=0 blks, \
unwritten=1
naming   =version 2              bsize=4096
log      =/dev/sdk1              bsize=4096   blocks=10000, version=1
         =                       sectsz=512   sunit=0 blks
realtime =none                   extsz=65536  blocks=0, rtextents=0
```

The following demonstrates a large block write and the performance boost that an external logger offers.

```
atlorca2:/xfs.test # dd if=/dev/zero of=/xfs.test/zero.out bs=512k

atlorca2:~ # iostat -x 1 100| egrep "sdj|sdk"

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

sdj         0.00 32512.00  0.00 281.00    0.00 262144.00     0.00 \
131072.00   932.90   141.99  394.24   3.56 100.00
```

```
sdk          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
sdj          0.00 36576.00  0.00 277.00    0.00 294929.00    0.00 \
147464.50  1064.73   142.75  511.21   3.61 100.00
sdk          0.00   0.00  0.00  1.00    0.00    5.00     0.00    2.50    \
5.00     0.10  105.00 105.00  10.50
sdj          0.00 35560.00  0.00 279.00    0.00 286736.00    0.00 \
143368.00  1027.73   141.26  380.54   3.58 100.00
sdk          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
sdj          0.00 36576.00  0.00 277.00    0.00 294912.00    0.00 \
147456.00  1064.66   142.35  204.65   3.61 100.00
sdk          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
```

The journal log device provides little to no added performance. There is almost no I/O to disk `sdk`, which contains the log. Putting the log on a separate disk won't help performance because we are not diverting a meaningful amount of I/O. The journal log device provides no added benefit because the intent to modify/write is established at the beginning of the file access. From this point forward, the I/O is completed on the meta device, as depicted in the previous `iostat`.

Determining I/O Size for Filesystem Requests

As shown previously, the `dd` command `bs` option sets the block size to 512k for each I/O transaction. We can see this in `iostat`. To calculate this value, find the average request size (`avgrq-sz`) column from `iostat`. In this example, we find that `avgrq-sz` has a value of 1024 sectors. To calculate the block size, multiply `avgrq-sz` by the sector size (512 bytes). In this example:

1024 (sectors) × 512 (bytes/sector) = 524288 (bytes) / 1024 (KB/bytes) = 512KB

However, the same `dd` command using the XFS filesystem reveals that the largest `avgrq-sz` value set forth by a sequential read is equal to 256 (sectors) regardless of block size set by `dd`. Following the same calculations, we determine that an XFS sequential read has a block size set to 128KB. The block size of any I/O is an important item to understand because not all programs control the block size. A program can request a given block size; however, a lower-layer driver can require that the I/O request be broken into smaller requests as illustrated in the previous `iostat` output.

Thus we have demonstrated that using a remote journal provides no performance improvements for large data access on sequential reads and writes. However, when writing to our XFS filesystem with a large number of small files, the opposite becomes true: The remote journal does in fact help.

Loading a Filesystem with Small Block I/O Transfers

In the next test, we write 512KB files as fast as the system allows while watching the load on the journal device and the filesystem meta device. To run this test, we must write a short, simple program to control the number of files to create during our test phase. The program is as follows:

```
#!/bin/sh
count=1
total=0
while [ $total -ne $* ]
do
total='expr $total + $count'
  touch $total
        dd if=/xfs.test/zero.out of=/xfs.test/$total bs=512k > \
        /dev/null 2>&1  #Using dd to control the BS.
        done
```

This program uses the same `dd` command used throughout our testing. It is important to always keep the control in any test constant. By running the previous program, thousands of 512KB files are created, causing an increased load on the journal log device (`sdk`), as depicted in the following listing:

```
atlorca2:/xfs.test # ./count_greg.sh 10000
atlorca2:~ # iostat -x 1 100| egrep "sdj|sdk"

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

sdj          0.00 16219.00  0.00 181.00    0.00 131169.00     0.00 \
65584.50   724.69     9.31   54.90   2.77  50.20
sdk          0.00   0.00  0.00 53.00    0.00 3200.00     0.00  \
1600.00    60.38     1.43   28.96   9.40  49.80
sdj          0.00 20274.00  0.00 201.00    0.00 163936.00     \
0.00 81968.00   815.60    11.90   54.74   2.83  56.90
sdk          0.00   0.00  0.00 39.00    0.00 2752.00     0.00  \
1376.00    70.56     1.19   26.26  14.00  54.60
sdj          0.00 20273.00  0.00 198.00    0.00 163936.00     \
0.00 81968.00   827.96    10.96   54.34   2.77  54.80
sdk          0.00   0.00  0.00 50.00    0.00 3072.00     0.00  \
1536.00    61.44     1.43   30.48  10.90  54.50
sdj          0.00 16217.00  0.00 200.00    0.00 131138.00     \
0.00 65569.00   655.69    10.22   56.56   2.71  54.10
sdk          0.00   0.00  0.00 50.00    0.00 2982.00     0.00  \
1491.00    59.64     1.37   28.78  10.92  54.60
```

By creating thousands of files or modifying the same file thousands of times a minute, we can see the added load on the journal device `sdk` as well as the filesystem device `sdj`. By understanding the end goal, we can make better decisions about how to size and lay out a filesystem.

In addition, notice the block size of the I/O request submitted to the filesystem and how the filesystem responded. In the previous example on a sequential read, the block size is restricted to 128K. However, on a sequential write, the blocking structure on the XFS filesystem is that which is set forth by the command calling the SCSI write, setting the average request size to 512k as shown in the previous `iostat` illustration. However, will the same results be found using a completely different filesystem?

Let's repeat our test on ext3, also known as ext2 with journaling, to depict I/O latency and blocking factors.

```
atlorca2:/ext3.test # dd if=/ext3.test/usr.tar of=/dev/null bs=512k
atlorca2:/ # iostat -x 1 100¦ grep sdj

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

sdj         60.00   0.00 898.00  0.00 229856.00    0.00 114928.00    \
0.00   255.96     0.98    1.10   1.09  98.20
sdj         58.00   0.00 939.00  0.00 240360.00    0.00 120180.00    \
0.00   255.97     0.98    1.04   1.04  97.50
sdj         62.00   1.00 918.00  3.00 234984.00   32.00 117492.00    \
16.00   255.17     0.97    1.06   1.05  97.00
sdj         62.00   0.00 913.00  0.00 233704.00    0.00 116852.00    \
0.00   255.97     0.98    1.07   1.07  97.80
sdj         58.00   0.00 948.00  0.00 242664.00    0.00 121332.00    \
0.00   255.97     0.96    1.01   1.01  96.20
sdj         62.00   0.00 933.00  0.00 238824.00    0.00 119412.00    \
0.00   255.97     0.97    1.04   1.04  97.30
```

The sequential read test holds the same blocking factor as previously seen on XFS, with a little savings on overhead with respect to average wait time and service time. To continue our example, let's proceed with the file create script discussed previously.

```
#!/bin/sh
count=1
total=0
while [ $total -ne $* ]
do
total='expr $total + $count'
  touch $total
        dd if=/ext3.test/zero.out of=/ext3.test/$total bs=512k > \
        /dev/null 2>&1  #Using dd to control the BS.
done
```

```
atlorca2:/ext3.test # ./count_greg.sh 10000

atlorca2:/ # iostat -x 1 100| grep sdj

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

sdj          0.00 51687.00  0.00 268.00    0.00 416864.00     0.00 \
208432.00  1555.46   128.62  346.74   3.45  92.40
sdj          0.00 27164.00  1.00 264.00    8.00 219040.00     4.00 \
109520.00   826.60   139.42  521.48   3.77 100.00
sdj          0.00 656.00  1.00 113.00    8.00 5312.00   4.00  2656.00   \
46.67    21.95  516.42   3.85  43.90
sdj          0.00   0.00  1.00  0.00    8.00    0.00    4.00     0.00   \
8.00     0.00    1.00   1.00   0.10
sdj          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
sdj          0.00 52070.00  1.00 268.00    8.00 419936.00     4.00 \
209968.00  1561.13   128.44  346.09   3.43  92.30
sdj          0.00 27800.00  0.00 271.00    0.00 224160.00     0.00 \
112080.00   827.16   139.27  513.93   3.69 100.00
sdj          0.00 662.00  2.00 112.00   16.00 5368.00   8.00  2684.00   \
47.23    20.45  489.31   3.91  44.60
sdj          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
sdj          0.00   0.00  1.00  0.00    8.00    0.00     4.00     0.00
8.00     0.00    0.00   0.00   0.00
sdj          0.00 51176.00  0.00 273.00    0.00 412792.00     0.00 \
206396.00  1512.06   128.12  336.55   3.37  92.00
sdj          0.00 27927.00  0.00 274.00    0.00 225184.00     0.00 \
112592.00   821.84   138.00  510.66   3.65 100.00
sdj          0.00 658.00  0.00 105.00    0.00 5328.00   0.00  2664.00   \
50.74    17.44  492.92   3.57  37.50
sdj          0.00   0.00  1.00 128.00    8.00 1024.00   4.00   512.00   \
8.00     2.88   22.32   0.35   4.50
```

Notice how the filesystem buffers the outbound I/O, submitting them to the SCSI layer in a burst pattern. Though nothing is wrong with this I/O pattern, you must understand that the larger the burst, the larger the strain on the storage array. For example, exchange servers save up and de-stage out a burst of I/O operations, which can flood an array's write pending cache, so you should monitor for excessive write burst. However, the write block size maintains a 512k average block, which is similar to the XFS on writes with large block requests.

Utilizing Key Benefits of a Filesystem

As we've seen, I/O block sizes, stripe size, and filesystem layouts have unique benefits that aid I/O performance. In addition to these items, most filesystems have unique characteristics that are designed to guess the next request, trying to save resources by anticipating requests. This is accomplished by read-ahead algorithms. Ext2, Ext3, JFS, and XFS all have the capability to perform read-ahead, as shown with XFS in the following XFS source code for mounting `/usr/src/linux/fs/xfs/xfs_mount.c`:

```
/*
 * Set the number of readahead buffers to use based on
 * physical memory size.
 */
if (xfs_physmem <= 4096)               /* <= 16MB */
        mp->m_nreadaheads = XFS_RW_NREADAHEAD_16MB;
else if (xfs_physmem <= 8192)   /* <= 32MB */
        mp->m_nreadaheads = XFS_RW_NREADAHEAD_32MB;
else
        mp->m_nreadaheads = XFS_RW_NREADAHEAD_K32;
if (sbp->sb_blocklog > readio_log) {
        mp->m_readio_log = sbp->sb_blocklog;
} else {
        mp->m_readio_log = readio_log;
}
mp->m_readio_blocks = 1 << (mp->m_readio_log - sbp->sb_blocklog);
if (sbp->sb_blocklog > writeio_log) {
        mp->m_writeio_log = sbp->sb_blocklog;
```

```
} else {
        mp->m_writeio_log = writeio_log;
}
mp->m_writeio_blocks = 1 << (mp->m_writeio_log - sbp->sb_blocklog);
```

Although read-ahead is a powerful attribute, a concern exists. It is not fair to say that read-ahead causes these drawbacks, as a true increase in read performance can be seen on any filesystem that uses read-ahead functionality. When using read-ahead, filesystem block size is an important factor. For example, if filesystem block size is 8k and sequential read pattern exist where an application is reading 1K sequential blocks (index), read-ahead kicks in and pulls an extra predefined number of blocks, where each block is equal to the filesystem block size. To sum up the concern with read-ahead, one must be careful not to read in more data than is needed. Another performance boost can be found by utilizing buffer cache.

Filesystems such as XFS, Reiser, Ext2, and Ext3 use the buffer cache and reduce the amount of memory for an application to process data in the buffer cache, forcing more physical I/O (PIO). Later in this chapter we discuss the difference between raw, Oracle Cluster File System (OCFS), and XFS in an example with buffer cache and read-ahead. Before we jump too far ahead, though, we need to cover one last topic with respect to disk performance.

Linux and Windows Performance and Tuning Sector Alignments

We have covered some in-depth I/O troubleshooting tactics for character and block devices. Now we need to address the rumor mill about disk geometry alignment. Geometry alignment, also known as sector alignment, is the new craze in Windows performance tweaking. Cylinders lie in a small band, like a ring on a platter. The cylinders are then divided into tracks (wedges), which contain sectors, which are described in great detail in Chapter 6, "Disk Partitions and File Systems." However, to discuss performance concerns with sector alignment, we would like to first depict sector locations (see Figure 4-1).

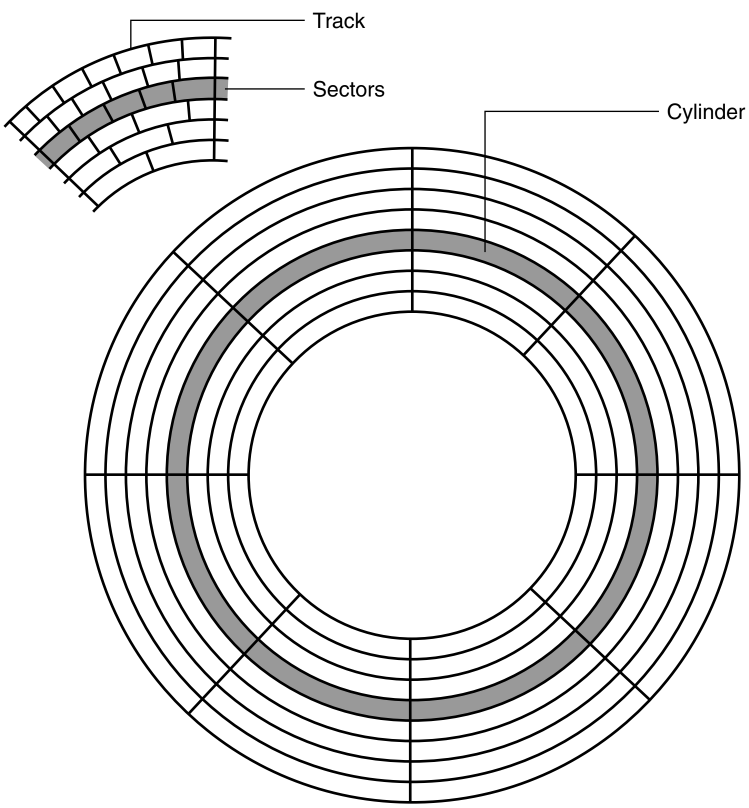

Figure 4-1 Cylinders, tracks, and sectors

Sector alignment provides little to no performance boost in Linux. To date, no issues exist with regards to how partitions and filesystems interact with sectors alignment for a given platter, regardless of whether the platter is logical or physical within Linux. However, for those who are interested, a performance boost has been documented with respect to DOS 6.X, Windows 2000, and greater. See http://www.microsoft.com/resources/documentation/Windows/2000/server/reskit/en-us/Default.asp?url=/resources/documentation/Windows/2000/server/reskit/en-us/prork/pree_exa_oori.asp and http://www.microsoft.com/resources/documentation/windows/2000/professional/reskit/en-us/part6/proch30.mspx for more information.

Performance Tuning and Benchmarking Using bonnie++

Now that we have covered some basics guidelines about I/O performance metrics, we need to revisit our primary goal. As already mentioned, our primary goal is to deliver methods for finding performance problems. In all circumstances, a good performance snapshot should be taken at every data center before and after changes to firmware roles, fabric changes, host changes, and so on.

The following is generalized performance data from a single LUN RAID 5 7d+1p, and it is provided to demonstrate the performance benchmark tool called `bonnie`. The following test does not depict the limit of the array used for this test. However, the following example enables a brief demonstration of a single LUN performance characteristic between three filesystems. The following `bonnie++` benchmark reflects the results of the equipment used throughout this chapter with XFS and Ext3 filesystems.

```
atlorca2:/ext3.test # bonnie++ -u root:root -d \
/ext3.test/bonnie.scratch/ -s 8064m -n 16:262144:8:128


Version  1.03 ------Sequential Output------ --Sequential Input- --Random-
              -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine  Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
atlorca2  8064M 15029  99 144685  44 52197  8 14819  99 124046   7 893.3  1
              ------Sequential Create------ --------Random Create--------
                   -Create-- --Read--- -Delete-- -Create-- --Read--- -
Delete--
```

```
files:max:min       /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP \
/sec %CP
  16:262144:8/128   721  27 14745  99  4572  37   765  28  3885  28 \
6288  58
```

Testing with XFS and journal on the same device yields the following:

```
atlorca2:/ # mkfs.xfs -f -l size=10000b /dev/sdj1
meta-data=/dev/sdj1              isize=256    agcount=16, \
agsize=3817947 blks
         =                       sectsz=512
data     =                       bsize=4096   blocks=61087152, \
imaxpct=25
         =                       sunit=0      swidth=0 blks, \
unwritten=1
naming   =version 2              bsize=4096
log      =internal log           bsize=4096   blocks=10000, version=1
         =                       sectsz=512   sunit=0 blks
realtime =none                   extsz=65536  blocks=0, rtextents=0

atlorca2:/ # mount -t xfs -o logbufs=8,logbsize=32768 /dev/sdj1 \
/xfs.test
atlorca2:/ # mkdir /xfs.test/bonnie.scratch/
atlorca2:/ # mount -t xfs -o logbufs=8,logbsize=32768 /dev/sdj1 \
/xfs.test

atlorca2:/xfs.test # bonnie++ -u root:root -d /xfs.test/bonnie.scratch/ \
-s 8064m -n 16:262144:8:128

Version  1.03   ------Sequential Output------ --Sequential Input- --
Random-
              -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine  Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
```

```
atlorca2      8064M 15474  99 161153  21 56513   8 14836  99 125513   9 \
938.8   1
                    ------Sequential Create------ --------Random Create--
------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -
Delete--
files:max:min        /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP \
/sec %CP
    16:262144:8/128  1151  24 12654 100  9705  89  1093  22 12327  99 \
6018  71
```

Testing with XFS without remote journal provides these results:

```
atlorca2:~ # mount -t xfs -o logbufs=8,logbsize=32768,logdev=/dev \
/sdk1/dev/sdj1 /xfs.test

atlorca2:/xfs.test # mkdir bonnie.scratch/
atlorca2:/xfs.test # bonnie++ -u root:root -d /xfs.test/bonnie.scratch/ \
-s 8064m -n 16:262144:8:128


Version  1.03       ------Sequential Output------ --Sequential Input- --
Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --
Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP \
 /sec %CP
atlorca2      8064M 15385  99 146197  20 58263   8 14833  99 126001   9 \
924.6   1
              ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -
Delete--
files:max:min       /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP \
/sec %CP
    16:262144:8/128  1175  24 12785 100 10236  95  1097  22 12280  99 \
6060  72
```

Just by changing the filesystem and journal log location, we pick up some nice performance on sequential I/O access with respect to XFS. The point of the previous demonstration is to identify factors other than hardware that increase performance. As we have seen, simply changing the filesystem layout or type can increase performance greatly. One other performance tool we enjoy using for SCSI measurements is IOzone, found at www.iozone.org.

Assessing Application CPU Utilization Issues

As with any performance problem, usually more than one factor exists. Our troubleshooting performance journey continues with coverage of application CPU usage and how to monitor it. In this section, CPU usage and application-specific topics are covered, focusing on process threads.

Determining What Processes Are Causing High CPU Utilization

To begin, we want to demonstrate how a few lines of code can load a CPU to a 100% busy state. The C code "using SLES 9 with long integer" illustrates a simple count program, which stresses the CPU in user space. It is important to understand that our application is not in system space, also called kernel mode, because we are not focusing on any I/O as previously discussed in this chapter.

Example 1 goes like this:

```
#include <stdio.h>
#include <sched.h>
#include <pthread.h> /* POSIX threads */
#include <stdlib.h>

#define num_threads  1

void *print_func(void *);

int main ()
```

```
{
        int x;
        printf("main() process has PID= %d PPID= %d\n", getpid(),
        getppid());

        pthread_t tid[num_threads];
        /* Now to create pthreads */
        for (x=0; x <= num_threads;x++)
        pthread_create(tid + x, NULL, print_func, NULL );

        /*wait for termination of threads before main continues*/
        for (x=0; x < num_threads;x++)
        {
        pthread_join(tid[x], NULL);
        printf("Main() PID %d joined with thread %d\n", getpid(),
        tid[x]);
        }
}

void *print_func (void *arg)
{
        long int y;
        printf("PID %d  PPID = %d  TID = %d\n",getpid(), getppid(),
        pthread_self());
        /* creating a very large loop for the CPU to chew on  :) */
/* Note, the following line may be changed to: for (y=1;y>0;y++) */
        for (y=1; y<10000000000000;y++)
        printf ("%d\n", y);
        return 0;
}
```

Note that instead of just counting to a large integer, you may want to create an infinite loop. The C code in Example 2 (also shown in Chapter 8, "Linux Processes: Structures, Hangs, and Core Dumps") generates an infinite loop with a signal handler used to kill the threads.

```
#include <pthread.h> /* POSIX threads */
#include <signal.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <errno.h>

#define num_threads  8

void *print_func(void *);
void threadid(int);
void stop_thread(int sig);
_syscall0(pid_t,gettid)

int main ()
{
        int x;
        pid_t tid;
        pthread_t threadid[num_threads];

       (void) signal(SIGALRM,stop_thread); /*signal handler */

        printf("Main process has PID= %d PPID= %d and TID= %d\n",
        getpid(), getppid(), gettid());

        /* Now to create pthreads */
        for (x=1; x <= num_threads;++x)
        pthread_create(&threadid[x], NULL, print_func, NULL );

        sleep(60); /* Let the threads warm the cpus up!!! :) */
        for (x=1; x < num_threads;++x)
                pthread_kill(threadid[x], SIGALRM);

        /*wait for termination of threads before main continues*/
        for (x=1; x < num_threads;++x)
        {
```

```
        printf("%d\n",x);
        pthread_join(threadid[x], NULL);
        printf("Main() PID %d joined with thread %d\n", getpid(),
        threadid[x]);
        }
}

void *print_func (void *arg)
{
        printf("PID %d  PPID = %d  Thread value of pthread_self = %d and
        TID= %d\n",getpid(), getppid(), pthread_self(),gettid());
        while(1);  /* nothing but spinning */
}

void stop_thread(int sig) {
pthread_exit(NULL);
}
```

To continue with Example 1, compile the code and run the application as follows:

```
atlorca2:/home/greg # cc -o CPU_load_count CPU_Load_count.c -lpthread
atlorca2:/home/greg # ./CPU_load_count | grep -i pid
main() process has PID= 5970 PPID= 3791
PID 5970  PPID = 3791  TID = 36354240
PID 5970  PPID = 3791  TID = 69908672
```

When running `top`, shown next, we find that our CPUs have a load, yet the PID reports zero percent usage for the `CPU_load_count` program. So where is the load coming from? To answer this question, we must look at the threads spawned by the parent process.

```
atlorca2:~ # top
top - 20:27:15 up 36 min,  3 users,  load average: 1.24, 1.22, 1.14
Tasks:  79 total,   1 running,  78 sleeping,   0 stopped,   0 zombie
 Cpu0 :  0.1% us,  0.6% sy,  0.0% ni, 99.1% id,  0.2% wa,  0.0% hi, \
0.0% si
```

```
 Cpu1 :  1.7% us,  1.6% sy,  0.0% ni, 96.7% id,  0.0% wa,  0.0% hi, \
0.0% si
 Cpu2 : 44.1% us, 19.0% sy,  0.0% ni, 36.9% id,  0.0% wa,  0.0% hi, \
0.0% si
 Cpu3 : 43.3% us, 21.0% sy,  0.0% ni, 35.7% id,  0.0% wa,  0.0% hi, \
0.0% si
Mem:   4142992k total,   792624k used,  3350368k free,   176352k buffers
Swap:  1049568k total,        0k used,  1049568k free,   308864k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+   \
PPID RUSER     UID WCHAN     COMMAND
 5974 root      16   0  3632 2112 3184 R  0.7  0.1   0:00.13  3874 \
root        0 -         top
 5971 root      15   0  3168 1648 2848 S  3.3  0.0   0:02.16  3791 \
root        0 pipe_wait grep
 5970 root      17   0 68448 1168 2560 S  0.0  0.0   0:00.00  3791 \
root        0 schedule_ CPU_load_count
 5966 root      16   0  7376 4624 6128 S  0.0  0.1   0:00.01  5601 \
root        0 schedule_ vi
 5601 root      15   0  5744 3920 4912 S  0.0  0.1   0:00.11  5598 \
root        0 wait4     bash
 5598 root      16   0 15264 6288  13m S  0.0  0.2   0:00.04  3746 \
root        0 schedule_ sshd
```

You can see `top` shows that the `CPU_Load_count` program is running, yet it reflects zero load on the CPU. This is because `top` is not thread-aware in SLES 9 by default.

A simple way to determine a thread's impact on a CPU is by using `ps` with certain flags, as demonstrated in the following. This is because in Linux, threads are separate processes (tasks).

```
atlorca2:/proc # ps -elfm >/tmp/ps.elfm.out
```

`vi` the file, find the PID, and focus on the threads in running (`R`) state.

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY    \
TIME CMD
4 - root      5970  3791  0   -    - -  4276 -      20:26 pts/0  \
00:00:00 ./CPU_load_count
4 S root         -     -  0  77   0 -     - schedu 20:26 - \
00:00:00 -
1 R root         -     - 64  77   0 -     - schedu 20:26 - \
00:02:41 -
1 S root         -     - 64  79   0 -     - schedu 20:26 - \
00:02:40 -
```

Though `top` provides a great cursory view of your system, other performance tools are sometimes required to find the smoking gun, such as the `ps` command in the preceding example. Other times, performance trends are required to isolate a problem area, and products such as HP Insight Manager with performance plug-ins may be more suited.

Using Oracle statspak

In the following example, we use Oracle's statistics package, called `statspak`, to focus on an Oracle performance concern.

```
DB Name      DB Id   Instance  Inst Num  Release      Cluster Host
------------ ----------- ----------- -------- --------- ------- ----------
DB_NAME      1797322438 DB_NAME      3 9.2.0.4.0     YES     Server_name

            Snap Id     Snap Time      Sessions Curs/Sess Comment
        ------- ------------------ -------- --------- ------------------
Begin Snap:                        23942 Date 11:39:12      144       3.0

  End Snap:                        23958 Date 11:49:17      148       3.1

   Elapsed:                               10.08 (mins)

Cache Sizes (end)
~~~~~~~~~~~~~~~~~
               Buffer Cache:     6,144M      Std Block Size:      8K
           Shared Pool Size:     1,024M          Log Buffer: 10,240K
```

```
Load Profile
~~~~~~~~~~~~               Per Second   Per Transaction
                       ---------------  ---------------
           Redo size:     6,535,063.00         6,362.58
       Logical reads:        30,501.36            29.70 <-Cache Reads LIO
       Block changes:        15,479.36            15.07
      Physical reads:         2,878.69             2.80 <-Disk reads PIO
      Physical writes:        3,674.53             3.58 <-Disk Writes PIO
           User calls:        5,760.67             5.61
               Parses:            1.51             0.00
         Hard parses:             0.01             0.00
               Sorts:           492.41             0.48
              Logons:             0.09             0.00
            Executes:         2,680.27             2.61
        Transactions:         1,027.11

  % Blocks changed per Read:   50.75   Recursive Call %:        15.27
 Rollback per transaction %:    0.01      Rows per Sort:         2.29

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.98       Redo NoWait %:        100.00
            Buffer  Hit   %:   90.57    In-memory Sort %:        100.00
            Library Hit   %:  100.00        Soft Parse %:         99.45
         Execute to Parse %:   99.94         Latch Hit %:         98.37
Parse CPU to Parse Elapsed %:  96.00     % Non-Parse CPU:         99.99

 Shared Pool Statistics        Begin    End
                               ------  ------
             Memory Usage %:   28.16   28.17
    % SQL with executions>1:   82.91   81.88
  % Memory for SQL w/exec>1:   90.94   90.49
```

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                 % Total
Event                                          Waits    Time (s)  Ela Time
-------------------------------------- ----------- ---------- --------
db file sequential read                    1,735,573      17,690     51.88
log file sync                                664,956       8,315     24.38
CPU time                                                  3,172      9.32
global cache open x                        1,556,450       1,136      3.36
log file sequential read                       3,652         811      2.35

-> s  - second
-> cs - centisecond -     100th of a second
-> ms - millisecond -    1000th of a second
-> us - microsecond - 1000000th of a second
                                                              Avg
                                                  Total Wait  Wait    Waits
Event                         Waits   Timeouts    Time (s)   (ms)     /txn
---------------------- ----------- ---------- ---------- ------ --------
db file sequential read   1,738,577          0      17,690     10      2.8
```

This statspak output is truncated to conserve space. Some of the key points of interest are highlighted in bold: Logical reads (LIO), Physical reads/writes (PIO), Latches, Buffered I/O, and db file sequential read. Please understand that many hundred parameters exist, and Oracle has published many performance documents and created many classes that we recommend reading and taking. However, this chapter's main goal is performance troubleshooting from a wide view, or in other words, a quick reference guide to performance troubleshooting. The bold areas in the previous listing are some critical places to focus, especially with LIO and PIO.

To elaborate on the bold elements in the previous listing, we begin with LIO and PIO. LIO is the access of a memory register, residing in the database buffer cache, and PIO is an I/O operation request from Oracle to the system for a data block fetch from spindle. In short, LIO is faster than PIO, but PIO is not always the bottleneck. A critical piece of the puzzle with regards to performance problems on any database connected to any storage device is understanding that a small percentage of PIOs (from Oracle's viewpoint) are actually a read from the host cache, storage cache, or both. Thus, a PIO from the database's perspective may in fact still be a read from cache, so having a high number of PIOs is not always a bad omen. In the previous example, the PIO reads were around 1.7 million with an average latency of 10ms, which, by the way, is not bad. However, although the I/O is fine, a memory control performance problem may still linger in the background.

Thus, while I/O may in fact be fine, memory lock control also must be addressed for smooth, fast database operation. Note that in this example, the Latch hit percentage is around 98%, which raises a red flag. A latch is basically a protection agent for access control with regards to shared memory within the system's global area (SGA). In short, the goal is to keep memory available so that the latch-free count remains high, keeping the percentage of Latch hit percentage around 99.5%. Latch info can be viewed by reviewing both the willing-to-wait and not-willing-to-wait latches found in the `immediate_gets` and `immediate_misses` columns by using `V$LATCH` or by looking at Oracle's `statspack`. In addition to waiting on free memory segments with regards to latches, we need to touch on buffer waits.

When a database starts to show an increase in buffer waits, the objective is to focus on the two main issues. The first issue is that memory is running low, which is impacting the second issue, that of physical I/O read/writes. A buffer wait is logged when the database must flush a write to spindle to clear up some available cache for data processing. The quick solution to this problem is to run raw (or other proprietary filesystem such as OCFS) to bypass host buffer cache so that buffer cache is used for data processing only. However, the huge drawback to using a non-buffer cache filesystem is the loss of performance with respect to read-ahead as discussed earlier in this chapter.

Now that we have covered some I/O basics, both physical and logical, and memory control with respect to latches, we present an example of an application failing to initialize due to lack of shared memory space. We demonstrate a lack of shared memory without going into detail about system V message queues, semaphores, or shared memory. As with all applications, more memory equals faster performance, and without enough memory, system failure is imminent.

Troubleshooting "No Space Left on Device" Errors When Allocating Shared Memory

Our example shows a common error 28 example using a 64-bit kernel, with a 64-bit application failing to initialize due to a memory address problem. Our focus is on interprocess communication (IPCS), as with any large application that spawns multiple threads/processes. Using our 64-bit machine, we bring a 64-bit Oracle 10g instance online, which fails with the following error to demonstrate a failed IPCS.

```
ORA-27102: out of memory
Linux-x86_64 Error: 28: No space left on device
```

System parameters are as follows:

```
# ipcs
------ Shared Memory Segments --------
key shmid owner perms bytes nattch status
0x00000000 1245184 gdm 600 393216 2 dest
0x852af124 127926273 oracle 640 8558477312 15
------ Semaphore Arrays --------
key semid owner perms nsems
0x3fbfeb1c 1933312 oracle 640 154

==== kernel parameters ======
# sysctl -a
kernel.sem = 250 32000 100 128
kernel.msgmnb = 16384
kernel.msgmni = 16
kernel.msgmax = 8192
kernel.shmmni = 4096
kernel.shmall = 2097152
kernel.shmmax = 34359738368
==== process ulimits (bash shell)
$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
file size (blocks, -f) unlimited
max locked memory (kbytes, -l) 4
max memory size (kbytes, -m) unlimited
open files (-n) 65536
pipe size (512 bytes, -p) 8
```

```
stack size (kbytes, -s) 10240
cpu time (seconds, -t) unlimited
max user processes (-u) 16384
virtual memory (kbytes, -v) unlimited
```

This failure is a result of the kernel not being able to fulfill the shared memory request. Not enough space is a condition explained in `/usr/src/linux/ipc/shm.c`, which reads:

```
        if (shm_tot + numpages >= shm_ctlall)
                return -ENOSPC;
```

The program we tried to start previously required more shared memory than we had allocated, which in turn caused the Oracle application to fail on initialization. The solution is to increase shared memory by the kernel parameter. In this example, we simply increase it to `shmall=8388608`.

Additional Performance Tools

As we conclude this chapter, we cover some uncommon tools that can be used to monitor performance characteristics and build charts in most cases. `isag`, RRDtool, Ganglia (which uses RRDtool to monitor grid computing and clustering), and Nagios are great performance tools. More monitoring tools exist, but for the most part, they are common tools used every day such as `sar`, `iostat`, `top`, and `netstat`. Due to space limitations, we only cover `isag` in this chapter. However, the other tools are easy to find and configure if one so desires. `isag`, found at http://www.volny.cz/linux_monitor/isag/index.html, provides a nice graphical front end to `sar`. After `systat` tools have been loaded, `isag` should be included, as depicted here:

```
atlorca2:/tmp # rpm -qf /usr/bin/isag
sysstat-5.0.1-35.1
```

isag

Most engineers who work with sar, iostat, and other performance tools will enjoy using isag, the GUI front end to sar. To give a quick demonstration of how the tool works, we must enable sar to collect some data. To achieve a real-world demonstration, we repeat our previous bonnie++ test while running sar -A to collect as much detail as possible and display it through isag.

To demonstrate, we mount an XFS filesystem with a local journal and use an NFS mount point to a HPUX server to demonstrate both disk and network load through bonnie++ while monitoring through sar and isag.

```
atlorca2:/var/log/sa # mount -t xfs -o logbufs=8,logbsize=32768 \
/dev/sdj1 /xfs.test

atlorca2:/var/log/sa # df
Filesystem           1K-blocks      Used Available Use% Mounted on

hpuxos.atl.hp.com:/scratch/customers
                     284470272  50955984 218940848  19% \
/scratch/customers
/dev/sdj1            244308608      4608 244304000   1% /xfs.test

atlorca2:/var/log/sa #  sar -A -o 1 100 #This will build a fine in \
/var/log/sa that isag will use.

atlorca2:/var/log/sa # isag
```

The resulting screenshots provide information on CPU utilization (see Figure 4-2) and swap utilization (see Figure 4-3).

Remember, if you are swapping, you need more memory.

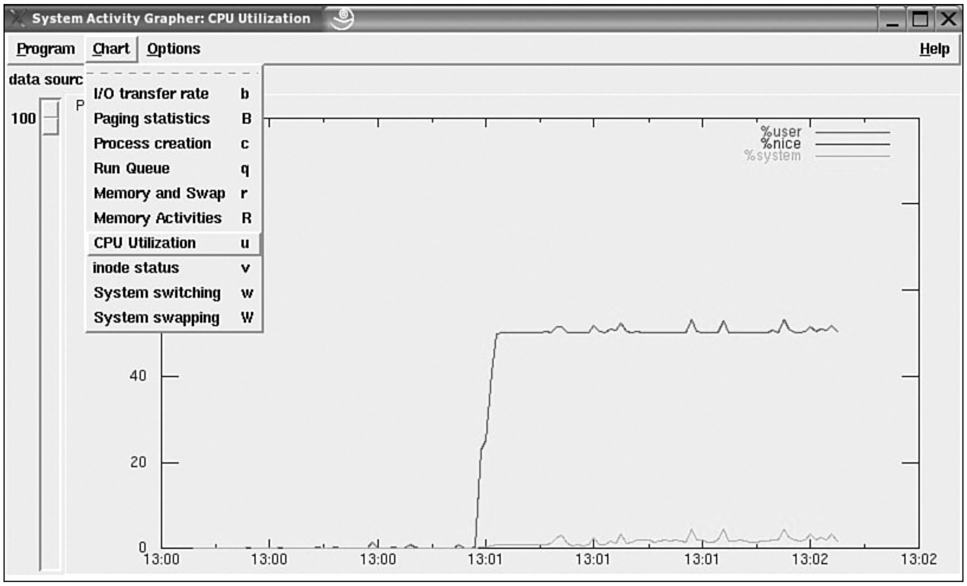

Figure 4-2 CPU utilization

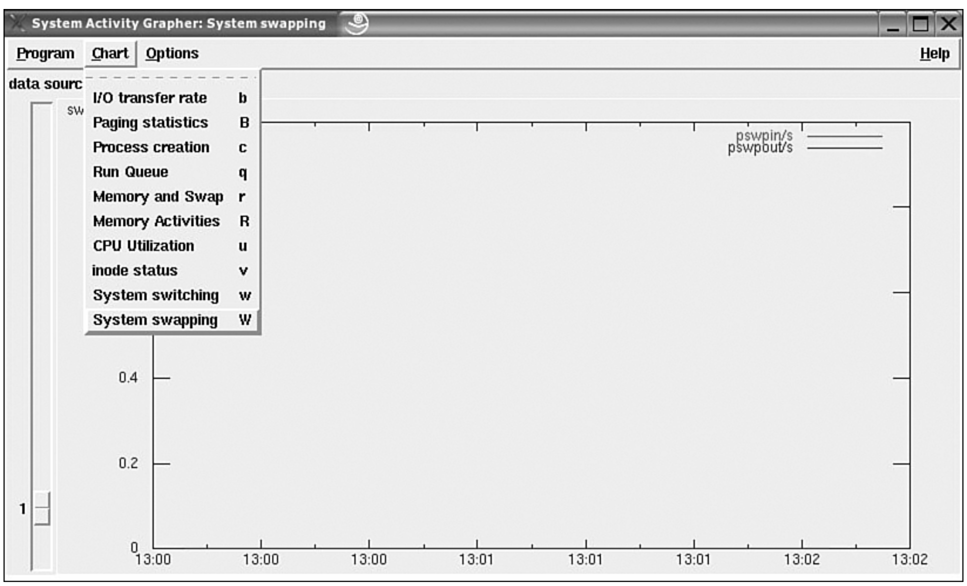

Figure 4-3 Swap utilization

Summary

Performance troubleshooting relies upon comparison. To know that performance is bad, you must have a preexisting benchmark from when performance was good. Though we can continue giving hundreds of examples of performance-related issues, a better approach is to understand the tools and know what to focus on.

To solve storage-related performance problems, you must be familiar with common tools such as `sar` and `iostat`. For threaded applications, make sure your version of `top` shows threads. Otherwise, focus on the `ps` command with the `m` flag. Other performance monitoring tools exist, such as Oprofile, Prospect, q-tools (q-syscollect, q-view, and q-dot), Perfmon, and Caliper.