Solaris 파일 시스템

파일 시스템의 성능(2)

지난 호에 이어 이번 호에서도 파일 시스템의 성능에 영향을 줄 수 있는 몇 가지 중요한 요인들과 파일 시스템의 패러미터들이 어떻게 성능에 영향을 미치는가에 대해 알아볼 텐데, 구체적으로 데이터 순차적 웍로드에 대한 몇 가지 사항들을 좀더 살펴보고, 파일 시스템 쓰기 작업과 데이터 집중식 랜덤 웍로드 등에 대해 소개한다.

정리 · 김봉환 | 한국 썬 시스템엔지니어링 본부 과장
원저 · Rechard Mc Dougall

데이터 순차적 웍로드

RAID 볼륨에 대한 클러스터 크기 조절

128KB 이상으로 클러스터 크기를 설정하기 전에 두 가지 사항을 고려해야 한다. 첫 번째는 기본값이 128KB로 설정된 Solaris 운영 체계의 SCSI 드라이버가 제공하는 SCSI 최대 전송률이다. 만일 파일 시스템이 512KB로 요청을 발생시킨다 해도 SCSI 드라이버는 이 요청들을 128KB 단위의 적은 조각(chunk)으로 쪼갤 것이다. Solstice Disk Suite나 Veritas Volume Manager와 같은 볼륨 관리자에서도 동일한 제약 사항이 발생한다. 대형 클러스터 크기를 사용하고자 하는 장비를 다룰 때마다 우리는 보다 큰 전송률을 사용하기 위해 /etc/system 구성 파일에 있는 SCSI와 볼륨 관리자 패러미터를 설정해줘야 한다. 다음에 표시한 /etc/system 파일의 변경 사항은 보다 큰 클러스터 크기를 설정하기 위해 필요한 사항에 대해 보여준다.

```
*
* Allow larger SCSI I/O transfers, parameter is bytes
*
set maxphys = 2097152

*
* Allow larger DiskSuite I/O transfers, parameter is bytes
*
set md_maxphys = 2097152


*
* Allow larger VxVM I/O transfers, parameter is 512 byte units
*
set vxio:vol_maxio = 4096
```

*값은 구성에 따라 최적값이 변경될 수 있다

두 번째 고려할 사항은 RAID 장비와 볼륨이 여러 개의 물리적 장비로 이뤄져 하나의 커다란 볼륨을 형성한다는 사실이며, 수많은 I/O 요청이 RAID 볼륨에 도달하면 I/O 요청에 어떠한 일이 벌어지는가에 대한 주의를 기울여야 한다. 예를 들어 간단한 RAID 레벨 0 스트라이프는 Sun StorEdge A5200 스토리지 서브 시스템에서 7장의 디스크에 구축될 수 있다. I/O는 인터레이스 크기나 스트라이프 크기에 따라 각 7장의 디스크에 엇갈리게 일어난다.

별개의 장비들간에 I/O가 엇갈리게 발생하는 효과를 이용해 동시에 여러 가지를 요청하는 파일 시스템에서 파생되는 I/O를 여러 개로 나눌 수 있다. 파일 시스템에서 오는 512KB 용량의 단일 읽기 요청을 생각해보자. 이 512KB 용량의 요청이 128KB의 인터레이스 크기로 구성된 RAID 볼륨에 도착하면, 이것은 7개가 아닌 4개의 128KB로 나눠지는 것이다.

RAID 볼륨 내에 7장의 디스크 장비를 가지고 있으므로, 동시에 7개의 I/O 작업을 수행할 수 있으며, 7장의 디스크에서 동시에 I/O 발생 요청을 하는 파일 시스템을 제공하는 것이 이상적이다. 이렇게 하기 위해서는 RAID 볼륨의 전체적인 스트라이프 대역폭의 크기와 같은 I/O를 발생시켜야 한다. 그래야만 정확히 7개의 구성 요소로 나눠지기 때문이다. 따라서 I/O를 7×128KB 혹은 각각 896KB로 발생시켜야 하며, 이렇게 하기 위해 클러스터 크기를 896KB로 세팅하는 것이다.

RAID 레벨 5도 비슷하지만 한 가지 알아둬야 할 점은 n-1개의 장비에만 효율적인 공간을 가질 수 있다는 것이다. 따라서 레벨 5의 8웨이 RAID 스트라이프는 7웨이 방식의 RAID 0 스트라이프와 동일한 스트라이프 대역폭과 클러스터 크기를 가지게 된다. RAID 5의 클러스터 크기는 다음과 같이 맞춘다.

- **RAID 레벨 0, 스트라이핑** : 클러스터 크기 = 스트라이프 멤버의 수×인터레이스 크기
- **RAID 레벨 1, 미러링** : 클러스터 크기 = 단일 디스크와 동일
- **RAID 레벨 1+0, 스트라이핑 + 미러링** : 클러스터 크기 = 미러당 스트라이프 멤버의 수×인터레이스 크기

파일 시스템을 생성할 때 newfs 옵션을 이용하거나 tunefs 명령어를 이용해 클러스터 크기를 설정할 수 있다. 896KB의 클러스터 크기를 제공하는 파일 시스템을 생성하려면 다음과 같이 옵션이 포함된 newfs 명령어를 이용하면 된다.

```
# newfs -C 112 /dev/md/dsk/d20                    UFS
```

파일 시스템을 생성한 후에는 tunefs 명령어를 사용해 클러스터 크기를 변경할 수도 있다.

```
# tunefs -a 112 /dev/md/dsk/d20                   UFS
```

UFS 미리 읽기의 한계

UFS 미리 읽기 알고리즘은 여러 명의 사용자들간에도 동일하게 적용되므로 두 개의 프로세스가 동일한 파일을 읽는 것은 미리 읽기 알고리즘을 깨뜨리게 된다.

VxFS 파일 시스템 미리 읽기

Veritas VxFS 파일 시스템 역시 미리 읽기를 구현하지만 VxFS는 미리 읽기 크기를 설정할 때에는 다른 메커니즘을 이용한다.

VxFS의 미리 읽기 크기는 마운트되면서 Veritas Volume Manager를 사용할 때 자동으로 설정된다. 마운트 명령어는 볼륨 관리자를 검색한 후 지정된 볼륨에 가장 적절한 미리 읽기 옵션을 설정한다. 옵션은 커맨드 라인 옵션을 이용해 마운트될 때 설정하거나 /etc/vx/tunefstab 파일 내에 설정할 수도 있다.

VxFS 파일 시스템은 미리 읽을 데이터량을 결정하는 read_nstream 패러미터와 관련된 read_pref_io 패러미터를 사용한다. 미리 읽기의 기본값은 64K이다. read_nstream 패러미터는 한번에 뛰어난 성능을 낼 수 있도록 read_pref_io 크기의 병렬 읽기 요청을 위한 희망값을 반영하게 된다. 파일 시스템은 미리 읽기 크기를 결정하기 위해 read_nstream에 read_pref_io를 곱한 값을 이용한다. read_nstream의 기본값은 1이다.

다음의 예는 vxtunefs 명령어를 이용해 미리 읽기 크기를 896KB (=917504바이트)로 설정하는 방법을 나타낸 것이다.

```
# mount -F vxfs /dev/dsk/c0t3d0s7 /mnt
# vxtunefs -o read_pref_io=917504 /mnt            VxFS
```

LSI QFS 파일 시스템 미리 읽기

QFS 파일 시스템은 유사한 방식으로 미리 읽기를 구현하며, 미리 읽기를 시행할 클러스터당 블록의 개수를 반영한 maxcontig 패러미터를 사용한다. QFS 파일 시스템의 maxcontig 패러미터는 maxcontig 옵션을 이용해 마운트될 때 설정해야 한다.

```
# mount -o maxcontig=112 samfs1                   QFS
```

스토리지 장비 미리 읽기

최신 I/O 시스템은 스토리지 장비 내에 지능형 시스템을 갖추고 있으며, 이러한 레벨에서는 사전 패치 기능도 제공한다. 예를 들어 Sun StorEdge A3500 스토리지 컨트롤러는 파일 시스템의 클러스터 크기에 비례하는 컨트롤러 레벨에서 수행될 미리 읽기 크기를 조절하는 옵션도 가지고 있다.

메모리 맵 파일의 미리 읽기

메모리 맵 파일은 또 다른 미리 읽기 알고리즘을 발생시키는데, 이는 메모리 맵 파일이 파일 시스템 내에서 읽기 로직을 우회하기 때문이다. 메모리 맵 파일을 통한 순차적 액세스는 메모리 세그먼트 드라이버 내의 MAV_SEQUENTIAL을 이용해 탐지하거나 작동시킬 수 있으며, 이것은 매핑된 파일 seg_vn을 구현한다. 미리 읽기는 파일 시스템 클러스터 크기를 이용하지 않으며, 64K로 고정되어 있다.

파일 시스템 쓰기 작업의 이면

만일 우리가 각 I/O를 동기화시켜서 기록한다면 각 쓰기 작업이 완료되는 프로세싱간의 긴 시간을 기다려야 할 것이며, 실제 작업보다 I/O의 작업 완료를 위한 대기 시간에 대부분의 실행 시간을 소비하게 될 것이다. Unix는 쓰기 작업시 매우 효율적인 방법을 사용하는데, 이는 운영 체계를 뛰어넘어 쓰기 작업을 하므로 애플리케이션이 작업을 계속할 수 있도록 해준다. 이러한 비동기 방식 쓰기 작업의 실행은 파일 시스템이 데이터 블록을 쓰는 기본 방법이며, 동기 방식 쓰기 작업은 특정 파일 옵션이 설정될 때만 사용된다.

비동기 방식 쓰기 작업은 애플리케이션이 각 I/O에 대한 대기 시간 없이 작업을 계속 진행할 수 있도록 해주며, 운영 체계는 인접한 쓰기 작업을 하나로 묶어주기에 충분하도록 쓰기 작업을 지연시킨다. 이러한 기능으로 인해 순차적인 쓰기 작업시 여러 번의 소규모 쓰기 작업 대신 단 몇 번의 대규모의 쓰기 작업으로 끝낼 수 있다.

UFS 파일 시스템 쓰기 작업의 이면

UFS 파일 시스템은 동일한 클러스터 크기 패러미터인 maxcontig를 이용해 물리적인 쓰기 작업이 이뤄지기 전에 얼마나 많은 쓰기 작업을 하나로 묶을지를 조절한다. 미리 읽기 작업의 경우와 마찬가지로 동일한 가이드라인이 뒤따르게 되며, RAID 장비가 사용되면 이것이 다시 반복된다. 주어진 장비의 스트라이프 크기에 클러스터 크기를 정렬시킬 때는 주의가 필요하다. 다음 예제는 순차적으로 8k 크기의 쓰기 작업을 발생시키기 위해 mkfile 명령어를 이용해 쓰기 작업을 할 때의 IO 통계치를 나타낸 것이다.

```
# mkfile 500m testfile&
# iostat -x 5
```

device	r/s	w/s	kr/s	kw/s	wait	actv	svc_t	%w	%b
sd3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd49	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd50	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd64	0.0	39.8	0.0	5097.4	0.0	39.5	924.0	0	100
ssd65	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd66	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd67	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0

iostat 명령어는 ssd64라는 디스크에 초당 39.8회의 쓰기 작업이 이뤄지고 있음을 보여주며, 이것은 평균적으로 초당 5,097.4KB의 양이다. 만일 우리가 전송률을 초당 I/O 수치로 나누면 평균 전송률은 128KB임을 유추할 수 있다. 이것으로 볼 때 기본값 128K의 클러스터 크기는 512바이트의 읽기 요청이 128KB 그룹으로 묶여지고 있다는 것을 확인할 수 있다.

UFS 파일 시스템의 클러스터 크기를 변화시켜서 그 결과를 매우 쉽게 관찰해볼 수 있다. 클러스터 크기를 1MB 혹은 1,024KB로 바꿔보자. 이렇게 하려면 maxcontig 설정치를 128로 바꿔줘야 하는데, 이것은 128개의 8KB 블록 혹은 1MB를 나타내는 것이다. 또한 /etc/system 내의 maxphys도 미리 설정해놓아야 한다.

```
# tunefs -a 16 /ufs
maximum contiguous block count changes from 16 to 128; default=8

# mkfile 500m testfile&
# iostat -x 5
```

UFS

device	r/s	w/s	kr/s	kw/s	wait	actv	svc_t	%w	%b
sd3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd49	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd50	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd64	0.2	6.0	1.0	6146.0	0.0	5.5	804.4	0	99
ssd65	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd66	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0
ssd67	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0

이제 ssd64 디스크에 대해 초당 6.0회의 쓰기 작업이 이뤄지고 있음을 iostat 명령어를 통해 볼 수 있으며, 이것은 초당 6,146KB임을 알 수 있다. 이때 전송률을 초당 I/O 횟수로 나누면 평균 전송률은 1,024KB임을 유추할 수 있다. 새로운 1,024KB의 클러스터 크기는 512바이트의 읽기 요청이 1,024KB 그룹으로 묶여지는 것이다.

UFS 클러스터링 알고리즘은 파일에 한 번에 1개의 프로세스나 쓰레드가 쓰여질 때 비로소 적절하게 동작한다는 것을 명심하자. 1개 이상의 프로세스나 쓰레드가 동일한 파일에 동시에 쓰기 작업을 한다면 UFS 내에서 일어나는 쓰기 알고리즘의 지연이 쓰기 작업을 불규칙한 크기로 깨뜨리기 시작한다.

UFS 쓰기 작업 억제

Solaris 2.x에서부터 포함되기 시작한 UFS 파일 시스템은 파일당 쓰기 작업이 이뤄지는 데이터의 양을 한정하는 억제 기능을 가지고 있다. 이것은 기본값으로 파일당 최적의 쓰기 용량을 384KB로 제한시켜 사용자들이 메모리 전체가 포화 상태가 되어 곤란을 겪지 않도록 해준다.

UFS 쓰기 억제에 대한 기본 패러미터는 사용자가 대부분의 디스크나 스토리지 시스템의 순차적 쓰기 성능을 완전히 사용하는 것을 방지한다.

만일 사용자가 이미 쓰기 작업을 할 때 디스크나 스트라이프, RAID 컨트롤러가 100% 사용되는 상황을 경험했다면 UFS 쓰기 억제 기능이 실행되었을 것이다.

쓰기 억제를 조절해주는 두 개의 패러미터가 있는데, High water mark와 Low water mark가 그것이다. UFS 파일 시스템은 쓰기 작업량이 ufs_HW 시스템 변수 내의 바이트의 양보다 현저하게 클 경우에는 쓰기 작업을 중지시키며, 쓰기 작업할 양이 ufs_LW보다 적으면 비로소 쓰기 작업을 재개한다.

사용자는 시스템이 동작중에도 UFS 쓰기 억제 크기를 증가시킬 수 있으며, adb 명령어를 이용해 결과치의 변화를 온라인 상에서 관찰할 수도 있다.

```
# adb -kw
physmem 4dd7

ufs_HW/W 0t16777216                                    UFS
ufs_HW:         0x600000        =       0x1000000
ufs_LW/W 0t8388608
ufs_LW:         0x40000         =       0x800000
```

또한 사용자는 /etc/system 내에 쓰기 억제를 고정적으로 설정할 수가 있다. 권고할 만한 쓰기 억제 설정값으로는 전체 메모리 용량의 64분의 1 크기로 High water mark를 설정하고, Low water mark는 128분의 1로 설정하는 것이다. 예를 들어 1GB 메모리 용량의 시스템이라면 ufs_HW는 16777216로, ufs_LW는 8388608로 설정한다.

```
*
* ufs_LW = 1/128th of memory
* ufs_HW = 1/64th of memory                            UFS*
set ufs_LW=8388608
set ufs_HW=16777216
```

· 쓰기 억제 기능이 없는 Veritas VxVM

Veritas VxVM 파일 시스템에는 비슷한 쓰기 억제 기능이 없다는 것을 유념하자. 대용량 파일 시스템을 생성할 때에는 메모리 페이징 초과 현상이 발생하는 것과 같은 사항에 유의해야 한다.

· LSI QFS 파일 시스템 쓰기 억제

LSI의 QFS 파일 시스템은 UFS와 비슷한 쓰기 억제 기능이 있는데, 이것은 파일 시스템이 마운트될 때 wr_throttle 옵션을 사용해 설정할 수 있다. Wr_throttle 옵션은 KB 단위이며, 파일 시스템이 쓰기 작업을 한 후 이를 중단하기 전에 설정해야 하는데, 256KB부터 32,768KB 사이로 설정해준다. QFS 쓰기 억제 설정치의 사용 예는 다음과 같다.

```
# mount -o wr_throttle=16384 /qfs1                     QFS
```

· RAID 레벨 5 스트라이프와 클러스터 정렬

앞서 스토리지 장비의 스트라이프 크기에 클러스터의 크기를 맞추는 것이 매우 중요하며, I/O가 각 스트라이프의 독립적인 여러 개의 요청에 대해 어떻게 나눠지는지를 살펴본 바 있다. RAID 레벨 5 정렬을 이용할 때에는 또 하나의 중요한 관련 요인이 있다.

RAID 레벨 5 볼륨은 패리티 정보를 계산해 데이터 일관성을 보호하고, 단일 드라이브에 데이터를 저장할 때 발생할 수 있는 데이터 유실을 일으키지 않으면서 또 다른 데이터를 저장해준다. 스트라이프에 기록할 때마다 주어진 스트라이프에 대한 모든 데이터를 읽어서 패리티 정보가 계산되고, 패리티를 산출해 이 정보를 기록한다. 예를 들어 5개의 128KB 인터레이스 와이드 RAID 레벨 5 스트라이프를 가지고 있으며 각 스트라이프에 128KB를 기록하고자 한다면, 각 드라이브에서 128KB의 데이터를 읽어와서 패리티를 재계산하고 새로운 패리티 블록을 기록한 후 128KB의 데이터를 기록하게 된다. 단일 기록만을 위해 여러 개의 읽기 및 쓰기 작업을 해야 한다.

모든 오버헤드는 필연적으로 쓰기 작업에 대한 손실을 야기시키는데, 사실 RAID 레벨 5 볼륨은 동일한 RAID 레벨 0 스트라이프에 기록하는 것보다 많이 느리다. 이러한 오버헤드는 스트라이프의 크기보다 쓰기 요청의 크기가 적을 경우 읽고 수정하고 쓰는 작업을 하는 데에 최악이다. 만일 정확한 스트라이프의 크기 만큼만 기록한다면 쓰기 작업만 해도 된다. 왜냐하면 전체 스트라이프의 패리티를 계산하는 데 필요한 모든 것을 이미 가지고 있기 때문이다. 스트라이프 크기 만한 I/O를 RAID 레벨 5 볼륨에 기록한다는 것은 필연적으로 부분적인 스트라이프 기록보다 빠를 수밖에 없다.

전체 스트라이프 쓰기를 하는 것이 더욱 효과적이므로 가능한 한 정확한 스트라이프 크기를 기록해야 하며, 스트라이프 크기에 정확하게 일치하는 파일 시스템의 클러스터 크기를 설정한다. 이 값은 앞서 언급했듯이 디스크의 개수에서 1을 뺀 후 인터레이스 크기를 곱해주면 된다. 그러나 한 가지 알아둬야 할 사항이 있다. 쓰기 크기 I/O를 기록한다 하더라도 스트라이프에 절반만 기록하려 한다면 두 개의 부분적인 스트라이프를 기록하려는 것을 막아야 한다. 이것은 전체 스트라이프 쓰기 작업보다 몇배가 느리기 때문이다.

이러한 문제를 해결하기 위해 몇몇 파일 시스템은 사전 설정 범주 내에 스트라이프와 클러스터를 정렬시켜주는 옵션을 가지고 있다. 이것은 쓰기 정렬로 알려져 있는데, Veritas와 QFS는 쓰기 정렬을 설정할 수 있는 옵션을 가지고 있으나 UFS는 이러한 옵션을 제공하지 않는다.

스트라이프 정렬은 소프트웨어 RAID 레벨 5를 구현하는 데 있어서 매우 중요한 사항이다. 볼륨 관리자는 초기화될 때 각 요청을 기록해야 하기 때문이다. 하드웨어 RAID 레벨 5 구현은 이점에 있어서 좀더 자유로운데, 이는 비휘발성 메모리 캐시(NVRAM : Nonvolatile Memory Cache)가 장착되어 있어 각 쓰기 작업을 정확하게 재정렬하도록 하드웨어 상에서 충분한 쓰기 지연을 시켜주기 때문이다. 만일 Sun StorEdge A5000/5200 스토리지 서브 시스템을 가지고 있거나 Veritas VxVM이나 Disk Suite RAID 5의 독립적인 SCSI 디스크 그룹을 사용한다면 쓰기 정렬은 사용자에게 추가적인 성능 개선을 제공해줄 것이다.

VxFS의 경우 사용자는 mkfs 명령어를 이용해 파일 시스템이 형성될 때 정렬 사항을 지정해줄 수 있다. 정렬 인수는 바이트 단위다.

```
# mkfs -F vxfs -o align=524288 /dev/vx/dsk/benchvol /mnt
```
VxFS

LSC QFS의 경우 사용자는 -a 옵션으로 파일 시스템을 구축할 때 정렬을 설정한다. 정렬 인수는 KB 단위다.

```
# sammkfs -a 512 samfs1
```
QFS

· 순차적 쓰기 성능의 로깅 효율성

파일 시스템 로깅은 파일 시스템 성능의 효율을 극대화한다. 파일 시스템에 기록되기 전에 데이터를 로그 장비에 기록해야 하기 때문이다. UFS와 VxFS 모두 메타 데이터 로깅 옵션을 가지고 있는데, 이것은 파일 크기를 변경할 때마다 이 정보를 메타 데이터 로그에 기록해야 한다는 의미한다.

UFS와 VxFS 로깅을 이용할 때 로그는 파일 시스템에 부착되며 파일 시스템 데이터 블록처럼 동일한 스토리지 매체에 존재한다. 이는 스토리지 장비가 데이터를 파일에 기록할 때 데이터와 로그 사이에서 검색해야 한다는 의미이다.

UFS의 경우 이것은 파일 생성시 몇 가지 추가적인 검색을 발생시키지만 전체적인 성능에 미치는 영향은 미미하다. UFS를 이용하면 대형 최적화 쓰기만을 보게 된다.

VxFS 파일 시스템은 각 파일 시스템의 변경 사항을 로그에 기록한다. 이는 매번 쓰기 작업을 동기화시키는 것과 마찬가지다. 이는 순차적 쓰기 성능에 큰 영향을 미친다. 마운트된 후 150MB의 파일이 여기에 생성된 파일 시스템의 예를 간단히 들어보자. 파일 생성 작업에는 거의 4분정도가 소요되지만 CPU 사용 시간은 26초에 불과하다. 나머지 시간들은 기나긴 I/O 대기 시간인 것이다. 친숙한 대형 클러스터된 쓰기 작업보다는 주로 8KB의 I/O에 매달리는 것을 볼 수 있을 텐데, 디스크가 할 수 있는 랜덤한 8KB의 쓰기 작업수에 제한을 받는 것이다.

```
# mount /vxfs
# cd /vxfs
# time mkfile 150m 150m

real        3m53.34s
user        0m0.31s
sys         0m26.47s

# iostat -xc 5
                    extended device statistics              cpu
device    r/s    w/s   kr/s   kw/s  wait  actv  svc_t  %w  %b  us  sy  wt  id
fd0       0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0   6   6  82   5
sd3       0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd0      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd1      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd2      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd3      0.0   94.6    0.0  762.4   0.0   1.1   12.0   0  88
ssd4      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
```
VxFS

위의 예에서 초당 94회의 I/O를 볼 수 있다. 이것은 초당 100~200회에 달하는 평균값에서도 매우 떨어지는 수치인데, 검색 패턴을 통해 설명될 수 있다. 여기서는 TAZ 디스크 탐색 툴을 이용해 15MB 용량의 파일 생성시 디스크 장비의 검색 패턴을 살펴보았다. 두 개의 붉은 선 사이에서 디스크가 검색하는 데에 대부분의 시간을 소비하고 있다는 것을 발견할 수 있다. 이것은 블록 0에 근접한 로그를 나타내며 일반 디스크의 4분의 1정도의 데이터를 보여준다. 이것은 기나긴 검색 시간과 왜 디스크가 초당 90회 정도의 I/O밖에는 수행할 수 없는지 그 이유를 잘 설명해준다.

로깅은 순차적 성능에 커다란 영향을 미치므로 임의의 대용량 파일을 생성하고자 한다면 파일 시스템을 마운트할 때에는 로그가 없는 베리타스 옵션을 사용할 수 있다. 이것은 DB 테이블 스페이스를 생성하거나 대용량 임시 파일을 생성해야 하는 고성능 컴퓨팅 업무를 할 때 매우 유용하다.

```
# mount -o nolog /vxfs
# cd /vxfs
# time mkfile 150m 150m

real       32.5
user        0.1
sys        23.6
# iostat -xc 5
device    r/s    w/s   kr/s   kw/s  wait  actv  svc_t  %w  %b  us  sy  wt  id
fd0       0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0  15  42   0  43
sd3       0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd0      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd1      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd2      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
ssd3      0.0  100.2    0.0 6400.2   0.0  14.5  144.8   0 100
ssd4      0.0    0.0    0.0    0.0   0.0   0.0    0.0   0   0
```
VxFS

소요 시간이 4분에서 32초로 떨어졌다는 사실을 주의깊게 살펴보자. 이미 우리가 예상했던 바와 같이 너무 많은 클러스터를 기록한 것이다. TAZ 유틸리티로 탐색해보면 블록 0에 근접한 로그 장비에 기록하는 작업은 더 이상 일어나지 않으며, 그 결과 우리는 쓰기 작업이 집중적으로 일어나고 있음을 알 수 있다.

일반적으로 로깅은 성능에 대해 약간의 비용을 지불해야 하지만 성능 오버헤드를 해결해줄 수 있는 로깅을 각각 구현할 수 있도록 여러 가지 옵션을 설정해줄 수도 있다. 베리타스는 비로깅으로부터 지연식 비동기 풀로깅에 이르는 여러 가지 로깅 레벨을 설정할 수 있다.

UFS와 VxFS, QFS 모두 각각의 디스크 상에 위치하는 메타 데이터를 허용하는 옵션을 가지고 있는데, 이것은 TAZ 탐색에서 보았던 검색 패턴을 제거한다. UFS는 데이터에서 로그를 분리시키는 데 필요한 디스크 슈트 로깅을 필요로 하며, VxFS는 NFS 가속기를 구매해야 하고, QFS는 표준 QFS 파일 시스템이 가지고 있는 옵션만으로도 이를 가능케 해준다.

데이터 집중식 랜덤 워크로드

truss 명령어를 이용하면 읽기 및 쓰기, lseek 시스템 호출을 살펴봄으로써 애플리케이션의 액세스 패턴 속성을 검사할 수 있다. 다음은 프로세스 id 19231번인 애플리케이션이 생성한 시스템 호출을 추적하기 위해 truss 명령어를 이용했다.

```
# truss -p 19231

lseek(3, 0x0D780000, SEEK_SET)          = 0x0D780000
read(3, 0xFFBDF5B0, 8192)               = 0
lseek(3, 0x0A6D0000, SEEK_SET)          = 0x0A6D0000
read(3, 0xFFBDF5B0, 8192)               = 0
lseek(3, 0x0FA58000, SEEK_SET)          = 0x0FA58000
read(3, 0xFFBDF5B0, 8192)               = 0
lseek(3, 0x0F79E000, SEEK_SET)          = 0x0F79E000
read(3, 0xFFBDF5B0, 8192)               = 0
lseek(3, 0x080E4000, SEEK_SET)          = 0x080E4000
read(3, 0xFFBDF5B0, 8192)               = 0
lseek(3, 0x024D4000, SEEK_SET)          = 0x024D4000
```

읽기 및 lseek 시스템 호출로부터 인수를 사용해 읽기 작업이 수행된 각 I/O의 크기와 검색 옵셋을 결정했다. lseek 시스템 호출은 16진법으로 된 파일 내의 옵셋을 보여줬다. 상기 예에서 처음 두 번의 검색은 0x0D780000과 0xA6D0000 옵셋에 있는데, 이것은 각각 바이트 번호 225968128과 38617088에 해당된다. 이 두 개의 주소는 랜덤한 것으로 보이며 나머지 옵셋을 검사해봐도 읽기 작업은 모두 랜덤한 것으로 보이게 된다. 읽기 시스템 호출에 대한 인수를 살펴보고, 세 번째 인수와 마찬가지로 각 읽기 작업의 크기를 보면 모든 읽기는 정확하게 8,192바이트 혹은 8KB임을 알 수 있다. 즉 상기 예에서 검색 패턴은 모두 랜덤하며 파일은 8K 블록 단위로 읽혀진다는 것을 알 수 있다.

랜덤 I/O에 대한 파일 시스템을 설정할 때 고려해야 할 몇 가지 사항은 다음과 같다.

- I/O 크기와 파일 시스템 블록 크기를 매칭시켜야 한다
- 적절한 크기의 대형 파일 시스템 캐시를 선택한다
- 사전 패치나 미리 읽기 기능을 정지시키거나 미리 읽기는 각 I/O의 크기로 제한한다
- DB와 같이 애플리케이션의 자체적인 캐싱을 사용할 때에는 파일 시스템 캐싱을 정지시킨다

대용량 쓰기 작업 등이 포함되는 워크로드에 대한 여러 I/O 크기에 파일 시스템 블록 크기를 매칭시키는 것은 매우 중요하다. 여러 블록 크기가 아닌 파일 시스템에 쓰기 작업을 하는 것은 블록의 부분적 쓰기를 초래하게 되며, 이것은 오래된 블록을 읽고 새로운 내용이 업데이트되며 전체 블록이 재기록되어야 한다. 홀수 크기의 쓰기 작업을 하는 애플리케이션은 읽기/수정/쓰기 주기를 삭제할 수 있도록 가능한 한 복수의 블록 크기에 근접한 기록이 될 수 있도록 수정돼야 한다.

랜덤 I/O 워크로드는 대부분 매우 작은 블록(2KB에서 8KB) 내의 데이터를 액세스하며 스토리지 장비의 각 I/O는 검색한 후 I/O가 이뤄지는데, 이는 동시에 1개의 파일 시스템 블록만을 읽어들이기 때문이다. 각 디스크 I/O는 수ms가 소요되며 I/O가 일어나는 동안 애플리케이션은 I/O가 완료될 때까지 기다려야만 한다. 이것이 애플리케이션 반응 시간의 대부분을 차지하게 된다. 그 결과 메모리 내에 적재되는 캐싱 파일 시스템 블록에 따라 애플리케이션의 성능은 큰 차이를 보이게 되며, 이러한 값 비싸고 느린 I/O는 회피하게 된다.

I/O 대기 시간에 소비되는 시간을 획기적으로 줄이기 위해 캐싱을 이용한다. 우리는 메모리를 이용해 디스크 블록을 미리 읽도록 캐싱하며, 디스크 블록이 다시 필요할 경우에는 메모리에서 간단하게 이를 읽어옴으로써 스토리지 장비로 다시 가는 일은 없게 된다.

마치며

다음 호에서는 Solaris 파일 캐싱 구현을 알아보고, 파일 시스템이 캐시를 어떻게 이용하며 이들이 파일 시스템 성능에 어떻게 영향을 미치는지를 알아본다.