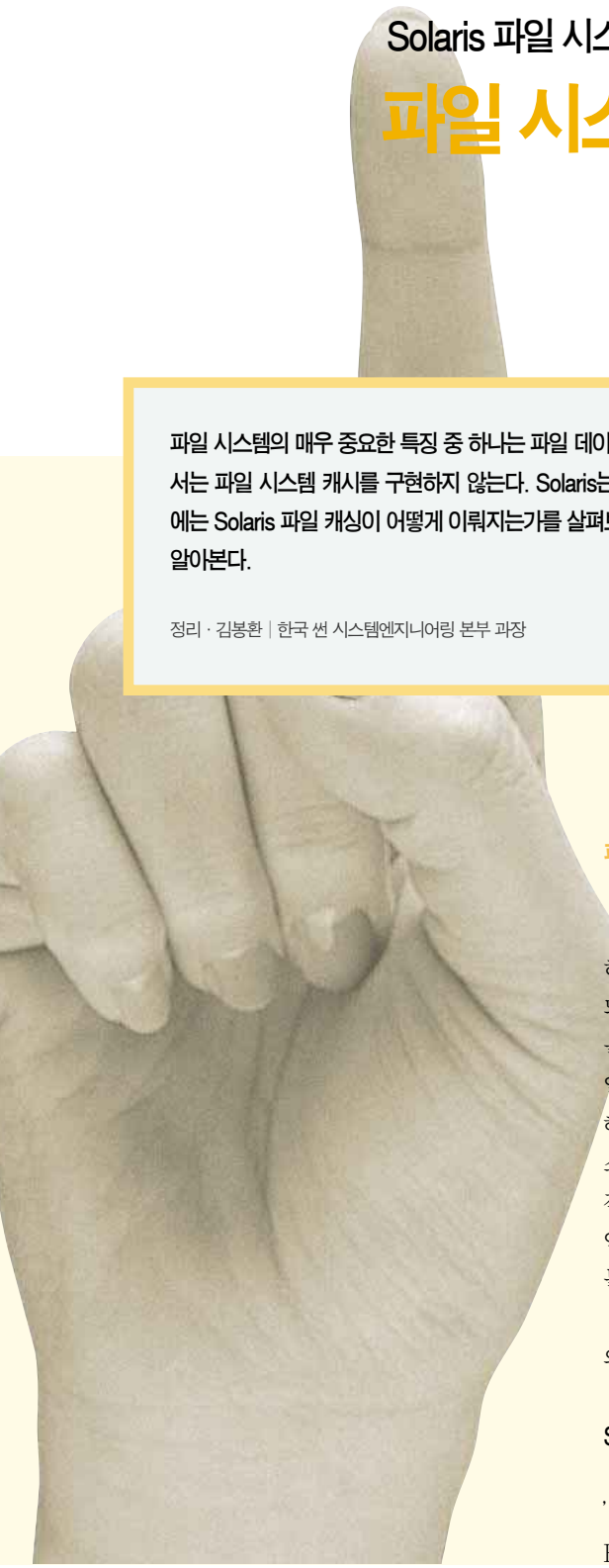
Solaris 파일 시스템

파일 시스템의 캐시

파일 시스템의 매우 중요한 특징 중 하나는 파일 데이터를 캐싱하는 능력이다. 하지만 아이러니컬하게도 파일 시스템에서는 파일 시스템 캐시를 구현하지 않는다. Solaris는 파일 시스템 캐시를 가상 메모리 시스템에서 구현한다. 이번 호에는 Solaris 파일 캐싱이 어떻게 이뤄지는가를 살펴보고, 파일 시스템 캐시와 가상 메모리 시스템 간의 상호 작용을 알아본다.

정리 · 김봉환 | 한국 썬 시스템엔지니어링 본부 과장



파일 시스템 캐싱

오래 전부터 Unix는 최근에 읽어들인 데이터의 복사본이나 블록 캐시에 기록된 블록을 보존하는 방식으로 I/O 시스템의 파일 시스템 캐싱을 구현했다. 이 블록 캐시는 디스크 상에 기록되어 있으며 물리적 디스크 섹터에서 데이터를 캐싱한다. 그림 1은 프로세스가 한 조각의 파일을 읽고 있는 예를 나타낸 것이다. 프로세스는 운영 체계에게 시스템 읽기 호출을 발생시켜 파일의 한 세그먼트를 읽는다. 파일 시스템은 파일에 대한 직 · 간접 블록 내의 블록 번호를 검색해 파일에 대한 관련 디스크 블록을 살펴본 다음 I/O 시스템에 해당 블록을 요청한다. I/O 시스템은 먼저 디스크에서 블록을 검색한 후 블록 버퍼 캐시로부터 디스크 블록을 읽어들여 부가적인 순차적 읽기를 만족시킨다. 디스크 블록이 메모리 내에 캐시된다 하더라도 이것은 물리적인 블록 캐시이므로 파일 시스템에 우선 요청한 다음, 캐시된 모든 읽기 데이터에 대한 물리적 블록 번호를 찾아야 한다.

구형 버퍼 캐시의 크기는 일반적으로 커널 구성 패러미터에 의해 고정되어 있다. 버퍼 캐시의 크기를 변경하려면 커널을 새로 구축한 후 재시동해야 한다.

Solaris 페이지 캐시

Solaris는 페이지 캐시라는 새로운 방식으로 파일 시스템 데이터를 캐싱한다. 페이지 캐시는 '85년 SunOS 4에서 새로 나온 가상 메모리 영역의 한 부분으로 개발되었으며, System V Release 4 Unix에서 사용되었다. 현재 Linux와 Windows NT에서도 페이지 캐시와 유사한

방식을 사용하고 있다. 페이지 캐시는 두 가지 중요한 차이점이 있는데, 하나는 자체적으로 동적 크기 조절이 가능해 애플리케이션이 사용하지 않는 모든 메모리를 이용할 수 있다는 점이며, 또 하나는 디스크 블록보다는 파일 블록을 캐시한다는 점이다.

무엇보다도 핵심적인 차이점은 페이지 캐시가 물리적 블록 캐시가 아닌 가상 파일 캐시라는 점이다. 이것은 운영 체계가 단순히 파일 참조만 찾아 파일 데이터를 검색하고 옵셋을 찾게 해주므로, 파일에 관련된 물리적 디스크 블록 번호를 찾아서 물리적 블록 캐시로부터 블록을 검색하지 않아도 된다. 이것은 훨씬 효과적인 방법이다.

그림 2는 새로운 페이지 캐시를 보여준다. Solaris가 처음에 파일을 읽기 시작하면, 파일 시스템이 페이지 크기 조각들의 메모리로 들어간 후 사용자에게 되돌려지는 과정을 통해 디스크에서 파일 데이터를 읽어오게 된다. 이후 동일한 파일 데이터의 세그먼트를 읽게 되면 이것은 파일 시스템을 통해 물리적인 검색을 실시할 필요 없이 곧바로 페이지 캐시에서 읽어올 수 있다. 구형 버퍼 캐시도 아직 Solaris에서 사용하고 있지만, 이는 직 · 간접 블록과 inode라는 메타 데이터 아이템인 물리적 블록 번호에 의해서만 알 수 있는 파일 시스템 데이터에 한해서다. 모든 파일 데이터는 페이지 캐시를 통해 캐싱하고 있다.

그림 2는 일견 단순해 보인다. 파일 시스템이 여전히 페이지 캐시 검색 기능을 포함하고 있지만 파일 시스템이 해야 할 대부분의 작업은 아주 단순화되어 있기 때문이다. 페이지 캐시는 가상 메모리 시스템에서 구현되며, 가상 메모리 시스템은 사실 페이지 캐시 이론에서 구축된 것이다. 물리적 메모리의 각 페이지도 바로 파일과 옵셋에 의해 동일한 방법으로 인식된다. 개별 메모리 공간과 연관된 메모리의 페이지가 스왑 장비와 연결된 반면, 파일과 연관된 페이지는 정규 파일과 연결되어 있다. 여기서는 Solaris 가상 메모리 시스템에 대해 깊게 다루지는 않는다. Solaris 메모리 시스템에 대한 보다 자세한 사항은 'http://www.sun.com/sun-on-net/performance/vmsizing.pdf'를 참조한다. 반드시 알아둬야 할 점은 파일 캐시는 프로세스 메모리와 비슷하며, 사용자는 파일 캐싱이 여분의 메모리 시스템처럼 동일한 동적 페이징을 공유하는 것을 보게 될 것이다.

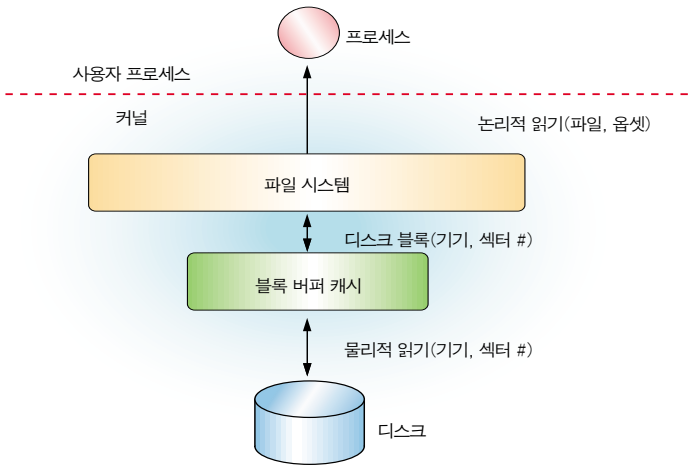


그림 1. 구형 버퍼 캐시

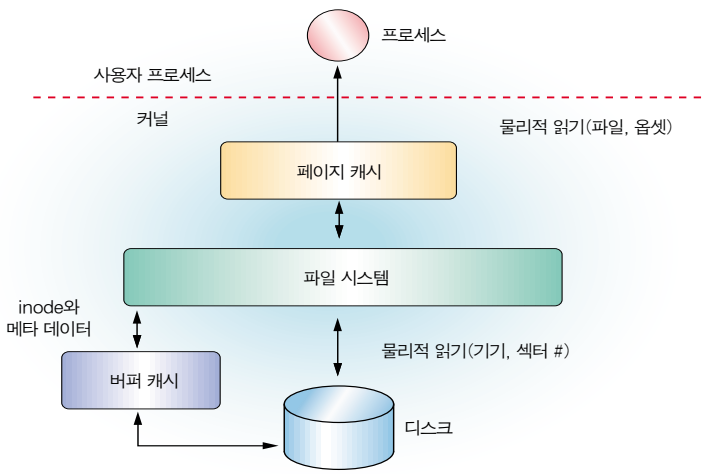


그림 2. 페이지 캐시를 경유한 파일 시스템 캐싱

Solaris의 구형 버퍼 캐시

구형 버퍼 캐시는 Solaris에서 inode와 파일 메타 데이터를 캐싱할 때 사용된다. Unix 구형 버전에서 버퍼 캐시는 nbuf 커널 패러미터에 의해 크기가 고정되며 512바이트 버퍼 단위로 지정된다. 사용자는 필요할 때마다 nbuf의 크기를 늘려서 버퍼 캐시를 조절할 수 있는데, bufhwm 커널 패러미터에 의해 규정된 최고값까지 가능하다. 기본값으로 버퍼 캐시는 물리적 메모리의 2%까지 늘릴 수 있다. Sysdef 명령어를 이용해 버퍼 캐시를 최고 한도까지 늘리는 방법을 알아보자.

```
# sysdef
*
* Tunable Parameters
*
 7757824        maximum memory allowed in buffer cache (bufhwm)
    5930        maximum number of processes (v.v_proc)
      99        maximum global priority in sys class (MAXCLSYSPRI)
    5925        maximum processes per user id (v.v_maxup)
      30        auto update time limit in seconds (NAUTOUP)
      25        page stealing low water mark (GPGSLO)
       5        fsflush run rate (FSFLUSHR)
      25        minimum resident memory for avoiding deadlock
(MINARMEM)
      25        minimum swapable memory for avoiding deadlock
(MINASMEM)
```

여기서 버퍼 캐시 안에 inode와 메타 데이타만을 보존하므로 그다지 큰 버퍼는 필요치 않게 된다. 사실 우리는 inode당 300바이트만 필요하며, 이는 동시에 액세스되는 것을 가정할 때 2GB의 파일당 1MB 정도 된다(이 법칙은 UFS 파일 시스템에 대한 것임을 기억하자). 예를 들어 총 100GB의 스토리지 공간에 100개의 파일을 가진 DB를 가지고 있으며, 동시에 이 파일중 50MB 정도를 액세스한다고 가정해보자. 우리가 필요한 것은 기껏해야 inode에 대해는 100×300 바이트 = 3KB 정도이며, 메타 데이타에 대해서는 50/2×1MB = 25MB 정도이다(직접 및 간접 블록).

물리적으로 5GB의 메모리를 가진 시스템의 경우, bufhwm의 기본값은 사용자에게 102MB의 bufhwm을 제공하는데, 이것은 버퍼 캐시를 위해서는 충분하고도 남는 양이다. 메모리를 많이 확보해야 할 경우, bujfhwm을 30MB로 제한함으로써 메모리를 절약할 수 있다. 이러한 사항은 /etc/system 파일의 bufhwm 패러미터를 조절해 구현할 수 있다. 이 bufhwm의 단위는 KB이다. 이 bufhwm을 예제보다 적게 구성하려면 다음 사항을 /etc/system에 넣으면 된다.

```
*
* Limit size of bufhwm
*
set bufhwm=30000
```

이제 sar -b 명령어를 이용해 버퍼 캐시 적중률 통계치를 살펴보자. 버퍼 캐시 통계치는 논리적인 읽기 횟수와 버퍼 캐시 내의 쓰기 횟수, 버퍼 캐시 영역 밖의 물리적 읽기 및 쓰기 횟수, 읽기/쓰기 적중률을 보여준다.

```
# sar -b 3 333

SunOS zangief 5.7 Generic sun4u   06/27/99

22:01:51   bread/s   lread/s   %rcache  bwrit/s   lwrit/s   %wcache  pread/s  pwrit/s
22:01:54         0      7118       100        0         0       100        0        0
22:01:57         0      7863       100        0         0       100        0        0
22:02:00         0      7931       100        0         0       100        0        0
22:02:03         0      7736       100        0         0       100        0        0
22:02:06         0      7643       100        0         0       100        0        0
22:02:09         0      7165       100        0         0       100        0        0
22:02:12         0      6306       100        8        25        68        0        0
22:02:15         0      8152       100        0         0       100        0        0
22:02:18         0      7893       100        0         0       100        0        0
```

이 시스템에서 버퍼 캐시는 100%의 읽기 캐싱 능력을 보여주며 쓰기 횟수는 적다. 이 측정값은 랜덤한 형태의 읽기 작업을 하는 100GB의 파일을 가진 시스템에서 산출된 것이다. 사용자는 소량의 대형 파일을 가진 시스템(예를 들어 DB 시스템)에서는 100%의 읽기 캐시 적중률을 얻고자 시도해야 하며 다수의 파일 시스템에서는 90% 이상의 캐시 적중률을 얻어야 한다.

페이지 캐시와 가상 메모리 시스템

가상 메모리 시스템은 페이지 캐시에서 구현되며 파일 시스템은 이 기능을 이용해 파일을 캐시한다. 즉 파일 시스템 캐싱의 특성을 이해하려면 가상 메모리 시스템이 페이지 캐시를 어떻게 구현하는지 알아야 한다는 것이다. 가상 메모리 시스템은 물리적 메모리를 페이지라고 알려진 조각들로 나누는데, 이것은 UltraSPARC 시스템에서는 8KB의 단위를 갖는다. 가상 메모리 시스템은 파일에서 데이터를 메모리로 읽어올 때 한 번에 한 페이지씩 읽어온다(최근에 발표한 UltraSPARC-Ⅲ는 Solaris 9를 사용해 복수 페이지 크기를 지원한다 : MPSS).

```
# ./rreadtest testfile&

# vmstat
 procs     memory            page                    disk          faults      cpu
 r b w   swap  free  re  mf  pi  po  fr  de  sr  so  -- -- -- --  in   sy   cs us sy id
 0 0 0   5043  2064   5   0  81   0   0   0   0  15   0   0   0   0 168  361   69  1 25 74
 0 0 0  50508  1336  14   0 222   0   0   0   0  35   0   0   0   0 210  902  130  2 51 47
 0 0 0  50508   648  10   0 177   0   0   0   0  27   0   0   0   0 168  850  121  1 60 39
 0 0 0  50508   584  29  57  88 109   0   0   6  14   0   0   0   0 108 5284  120  7 72 20
 0 0 0  50508   484   0  50 249  96   0   0  18  33   0   0   0   0 199  542  124  0 50 50
 0 0 0  50508   492   0  41 260  70   0   0  56  34   0   0   0   0 209  649  128  1 49 50
 0 0 0  50508   472   0  58 253 116   0   0  45  33   0   0   0   0 198  566  122  1 46 53
```

이 예제를 살펴보면 파일을 랜덤하게 읽어오는 프로그램을 가동시키면서 수많은 페이지 추가(디스크에서 메모리로 페이지가 이동)가 발생하는데, 이는 vmstat의 pi 컬럼의 숫자로 나타난다. Vmstat의 free 메모리 컬럼이 매우 낮은 값으로 떨어지는 것에 주목하자. 사실 free 메모리값은 거의 0에 가깝게 된다. 파일 시스템은 파일의 페이지 크기의 조각들 내에서 페이징될 때마다 매번 물리적 메모리의 페이지를 소모하기 때문이다. 사용자가 처음 시스템을 부팅하면 많은 양의 free 메모리가 확보되는 것을 본 적이 있을 것이다. 그리고 시스템이 메모리를 사용하면서 free 메모리는 점점 0에 가까워지게 되고, 결국 시스템이 다운되는 것을 경험한 적도 있을 것이다. 이는 지극히 정상적인 일이다. 파일 시스템은 각 파일을 읽고 쓰면서 이를 캐시하기 위해 가용한 모든 메모리를 사용하기 때문이다.

Solaris 7까지는 최근에 사용하지 않은 메모리 페이지를 검색하는 페이지 스캐너는 메모리를 free list에 되돌려놓는다. 페이지 스캐너는 lotsfree라고 알려진 시스템 패러미터값으로 메모리가 떨어지면 작동하기 시작한다. 이 예제에서는 페이지 스캐너가 파일 시스템이 사

용한 메모리를 대체하기 위해 초당 50페이지를 스캔한다.

페이지 캐시의 bufhwm과 동등한 패러미터는 없다. 페이지 캐시는 애플리케이션이 최근 사용하지 않은 프로세스 메모리까지 포함해 모든 가용한 메모리를 점점 소모한다. 시스템 페이지 빈도와 페이지 스캐너 수행 빈도는 파일 시스템이 페이지를 디스크에서 읽거나 디스크에 기록하는 빈도에 비례한다. 대형 시스템에서는 큰 페이징값을 나타나며, 이것이 정상이다. 파일 시스템을 통해 초당 10MB를 읽는 시스템을 생각하면 이것은 초당 1,280회의 page insert로 환산할 수 있으며, 페이지 스캐너는 초당 1,280회의 빈 페이지를 확보하기 위해 충분한 메모리를 스캔할 수 있어야 한다. 실제 페이지 스캐너는 초당 1,280페이지보다 훨씬 빨리 스캔할 수 있어야 한다. 페이지 스캐너가 가져오는 메모리가 모두 free 메모리가 되지는 않기 때문이다(페이지 스캐너는 최근에 사용하지 않은 메모리만을 free 메모리로 만든다). 만일 페이지 스캐너가 3페이지 중 1페이지만 free 메모리로 만들 수 있다면 페이지 스캐너는 초당 3,840페이지를 수행해야 한다. 하지만 이처럼 높은 스캔률에 대해 염려할 필요는 없다. 만일 사용자가 파일 시스템을 많이 사용한다면 이러한 빠른 스캔 속도는 정상이기 때문이다. 대개 높은 스캔률에 대한 사항들은 메모리가 부족하다는 의미이나, 본지를 통해 여러 가지 상황에서 높은 스캔률이 왜 정상적인가를 이해할 수 있을 것이다.

· 파일 시스템 페이징 트릭

몇몇 파일 시스템은 다음 두 가지 방법으로 메모리의 부담을 줄이고 있다. 순차적 액세스를 통한 사용 가능 메모리를 확보하거나 사용 가능 메모리가 lotsfree(1/64) 이하로 떨어질 때 페이지를 방출하는 방법이 그것이다. 지난 호에서 살펴보았듯이 파일이 순차적으로 액세스될 때 UFS에서는 free 메모리가 발생하므로, 대형 파일을 순차적으로 스캔할 경우에는 캐시를 사용할 필요가 없다. 즉 우리가 파일을 생성하거나 순차적으로 읽을 때에는 고속으로 스캐닝되지 않는다는 것이다.

몇몇 파일 시스템은 메모리가 lotsfree 이하로 떨어지면 페이지 캐시 사용을 제한하도록 몇 가지 사항을 점검한다. UFS 파일 시스템이 사용하는 부가적인 패러미터로 pages_ before_pager가 있다. 이 패러미터는 페이지 스캐너가 시동될 때의 메모리의 합계를 나타내는데, 기본값은 200페이지다. 사용 가능 메모리가 lotsfree보다 많은 1.6MB까지 떨어질 때(UltraSPARC에서) 파일 시스템은 페이지 캐시 사용을 줄이기 시작한다. 메모리양이 lotsfree와 pages_before_ pager를 더한 값이 되면 Solaris 파일 시스템은 다음과 같은 작업을 수행한다.

· 페이지가 기록된 후 모든 페이지를 살펴본다
· UFS와 NFS는 순차적 액세스를 통해 free 메모리를 확보한다
· NFS는 미리 읽기를 중단한다
· NFS는 비동기 방식이 아닌 동기 방식으로 기록한다
· VxFS는 free 메모리를 확보한다(몇몇 버전에서만)

모든 페이징이 나의 시스템에 해로울까?

앞서 언급했듯이 높은 페이징이나 빠른 스캔 속도는 정상적인 사항이다. 이는 애플리케이션이 혼자서 많은 메모리를 사용함으로써 페이지 스캐너가 더욱 활발하게 작동되도록 압박을 받는 것과 같다. 만일 사용자가 초당 수백페이지 이상의 속도로 스캔한다면 페이지 스캐너가 점검하는 데 필요한 시간은 불과 수초로 줄어들게 된다. 즉 지난 수초동안 사용되지 않았던 모든 페이지는 파일 시스템 사용시 페이지 스캐너가 모두 가져오게 된다. 이것은 애플리케이션의 성능에 매우 부정적인 영향을 미친다. 따라서 '우선 페이징'이 필요하다.

만일 파일 시스템의 I/O가 발생하는 동안 시스템이 느려졌다면 그 이유는 파일 시스템이 작업하면서 계속적으로 애플리케이션의 페이지 I/O를 유발하기 때문이다. 예를 들어 파일 시스템을 상당히 많이 사용하는 OLTP 애플리케이션을 생각해보자. DB는 파일 시스템 I/O를 유발하고, 이로 인해 페이지 스캐너는 시스템에서 페이지를 지속적으로 가져오게 된다. OLTP 애플리케이션 사용자는 마지막 트랜잭션에서 스크린 컨텐츠 읽기를 15초동안 멈추게 된다(애플리케이션이 쉬고 있는 상태). 이 시간동안 페이지 스캐너는 사용자 애플리케이션과 연관된 페이지들이 참조되지 않은 것을 알게 되며, 이를 훔쳐와 사용할 수 있다. 스캐너가 페이지를 훔쳐간 후, 사용자가 다음 키보드를 누르면 사용자의 애플리케이션은 페이지백될 때까지 강제로 대기하게 된다(보통 몇 초정도 걸린다). 이에 따라 사용자들은 애플리케이션이 스왑 장비에서 페이지인될 때까지 대기해야 한다. 이러한 현상은 물리적 메모리만으로 모든 애플리케이션이 구동될 정도로 충분한 메모리를 가진 시스템에서도 마찬가지로 나타난다.

우선 페이징 알고리즘은 파일 캐시 주변 영역을 효과적으로 설정해주므로 파일 시스템 I/O는 애플리케이션의 불필요한 페이징을 일으키지 않는다. 페이징 알

고리즘은 중요도의 순서에 따라 페이지 캐시 내의 서로 다른 형태의 페이지 우선 순위에 의해 이 작업을 수행한다.

· 최고 – 메모리를 점유하는 애플리케이션을 포함한 실행 공유 라이브러리와 연관된 페이지
· 최저 – 정규 파일 캐시 페이지

순차적으로 데이터를 읽을 때 메타 데이터의 읽기 오버헤드가 매우 적게 발생한다는 것을 의미한다. 하지만 랜덤한 방법으로 파일을 읽을 때에는 읽고자 하는 모든 데이터 블럭에 대한 블럭 주소를 살펴봐야 하는데, 이것은 블럭 기반 파일 시스템에서 시행했던 방법과 유사하다.

동적 페이지 캐시는 사용 가능한 메모리가 0이 되는 시점까지 점점 증가하며, 페이지 스캐너가 동작하더라도 시스템에 충분한 메모리가 남아있다면 스캐너는 정규 파일과 관련된 페이지만 가져온다. 파일 시스템의 효율적인 페이징이란 시스템 내의 모든 페이지에 대한 것이 아닌 자신에 대한 것이다.

애플리케이션과 커널이 메모리 부족 현상을 느끼는 실제 메모리 부족 현상이 발생하면 스캐너는 애플리케이션에서 페이지를 가져올 수 있게 된다. 참고로 Solaris 7 전 버전에서는 기본값으로 우선 페이징은 불가능하도록 설정되어 있다. 이런 경우 우선 페이징을 사용하려면 Solaris 7이나 커널 패치 105181-13을 완료한 Solaris 2.6, 커널 패치 103640-25 이상을 완료한 Solaris 2.51을 사용해야 한다. 우선 페이징을 동작시키려면 /etc/system에서 다음과 같이 설정해준다. 단 Solaris 8 이후는 자동 사용이 내부적으로 성립되어 있어 이러한 작업을 할 필요가 없다.

```
*
* Enable Priority Paging
*
set priority_paging=1
```

/etc/system에서 priority_paging=1로 구성하면 새로운 메모리를 튜닝 및 캐시할 수 있으므로 이전의 페이징 워터마크를 2배나 높일 수 있고, 시스템 시동시의 lotsfree 상태로 만들 수 있다. 캐시 가능 메모리는 minfree와 desfree, lotsfree의 세 가지 패러미터를 이용해 튜닝할 수 있다 (우선 페이징에 대한 보다 자세한 내용은 'http://www.sun.com/sun-on-net/performance/priority_paging.html'를 참조한다).

우선 페이징은 실행 가능 권한을 가지고 주소 공간으로 매핑될 때 실행 파일과 정규 파일의 차이점을 기록을 통해 나타내준다. 즉 메모리 맵 시스템 호출을 가진 주소 공간으로 매핑되는 실행 가능 비트의 정규 파일은 실행 가능한 것으로 처리되며, 메모리에 매핑된 데이터 파일은 실행 비트를 가지고 있지 않다는 것을 보장해줘야 한다.

Solaris 7에는 페이징 카운터의 확장된 구성을 이용해 사용자가 어떠한 페이징이 일어나는지를 관찰할 수 있는 기능이 있다. 사용자들은 애플리케이션의 메모리 부족으로 인해 발생하는 페이징과 파일 시스템을 통한 페이징의 차이점을 알 수 있다. Solaris 7에서 memstat 명령어를 이용하면 페이징 카운터를 볼 수 있으며, memstat 명령어의 출력은 vmstat와 비슷하지만 여분의 필드는 페이징의 다른 형태를 보여준다. 정규 페이징 카운터뿐만 아니라 memstat 명령어는 페이징의 형태를 실행 가능과 애플리케이션, 파일 등 세 가지 형태로 보여준다. memstat 필드는 다음과 같다. 참고로 memstat 패키지는 'http://bigadmin.sun.com'에서 별도로 다운받아 설치해야 한다.

표 1. 여러 가지 파일 시스템의 ACL 지원 여부

컬럼	내용
pi	초당 총 페이지인
po	초당 총 페이지아웃
fr	초당 자유로운 총 페이지
sr	초당 페이지에서 페이지 스캔률
epi	초당 실행 가능 페이지인
epf	초당 자유로운 실행 가능 페이지
api	스왑 장비 스왑으로부터 초당 애플리케이션 페이지인
apo	초당 스왑 장비로 페이지아웃되는 애플리케이션
apf	초당 자유로운 애플리게이션 페이지
fpi	초당 파일 페이지인
fpo	초당 파일 페이지아웃
fpf	초당 자유로운 파일 페이지

사용자가 memstat 명령어를 이용하면 테스트 파일을 랜덤하게 읽을 때 다음과 같은 결과를 볼 수 있다. 스캐너는 메모리를 통해 초당 수백페이지를 스캐닝하고 실행 가능한 페이지를 사용할 수 있으며, 애플리케이션 페이지는 스왑 장비로 페이지아웃되어 마치 이들을 사용한 것처럼 된다. 메모리가 충분한 시스템에서 이는 사용자가 바라는 운영 방식이 아니다. 우선 페이징을 하면 이러한 일이 발생하는 것을 방지할 수 있다.

```
# ./readtest testfile&

# memstat 3
memory ---------- paging ------ -executable- - anonymous - -- filesys - --- cpu ---
```

```
 free re  mf    pi  po    fr de    sr epi epo epf api apo apf   fpi fpo fpf us sy wt id
 2080 1   0   749 512   821  0   264   0   0 269   0 512 549   749   0   2  1  7 92  0
 1912 0   0   762 384   709  0   237   0   0 290   0 384 418   762   0   0  1  4 94  0
 1768 0   0   738 426   610  0  1235   0   0 133   0 426 434   738   0  42  4 14 82  0
 1920 0   2   781 469   821  0   479   0   0 218   0 469 525   781   0  77 24 54 22  0
 2048 0   0   754 514   786  0   195   0   0 152   0 512 597   754   2  37  1  8 91  0
 2024 0   0   741 600   850  0   228   0   0 101   0 597 693   741   2  56  1  8 91  0
 2064 0   1   757 426   589  0   143   0   0  72   8 426 498   749   0  18  1  7 92  0
```

우선 페이징을 할 수 있다면 memstat 명령어를 사용해서 차이점을 발견할 수 있다.

```
# ./readtest testfile&

# memstat 3
memory  ---------- paging ----------  - executable -  - anonymous -  -- filesys -- --- cpu ---
 free  re  mf    pi  po    fr  de    sr  epi  epo  epf  api  apo apf   fpi fpo   fpf   us  sy  wt id
3616  6   0   760   0   752   0   673    0    0    0    0    0   0   760   0   752    2   3  95  0
3328  2  19   816   0   925   0  1265    0    0    0    0    0   0   816   0   925    2  10  88  0
3656  4 195   765   0   792   0   263    0    0    0    2    0   0   762   0   792    7  11  83  0
3712  4   0   757   0   792   0   186    0    0    0    0    0   0   757   0   792    1   9  91  0
3704  3   0   770   0   789   0   203    0    0    0    0    0   0   770   0   789    0   5  95  0
3704  4   0   757   0   805   0   205    0    0    0    0    0   0   757   0   805    2   6  92  0
3704  4   0   778   0   805   0   266    0    0    0    0    0   0   778   0   805    1   6  93  0
```

우선 페이징이 가능하게 되면 사용자는 가상 메모리 시스템의 특성 간의 차이점을 볼 수 있다. 동일한 테스트 프로그램을 이용해 파일 시스템을 다시 랜덤하게 읽으면 시스템이 페이징되면서 스캐너는 페이지를 관리하기 위해 동작하지만 지금 스캐너는 파일 페이지만 자유롭게 한다. 실행 가능한 익명의 메모리 컬럼에서 0으로 구성된 열에 의해 스캐너가 먼저 파일 페이지를 선택한다는 것을 확인할 수 있다. 이러한 활동은 fpi와 fpf 컬럼에 나타나며, 이것은 파일 페이지가 읽혀지고 더 많은 읽기 작업을 할 수 있도록 페이지 스캐너가 동일한 개수를 방출한다는 것을 의미한다.

파일 시스템 성능에 영향을 주는 페이징 패러미터

우선 페이징이 가능해지면 파일 시스템 스캔률이 상승한다. 페이지 스캐너는 프로세스 개별 메모리와 실행 가능한 것들을 건너뛰어야 하므로, 가져올 수 있는 파일 페이지를 발견하기 전에 더 많은 페이지를 스캔해야 하기 때문이다. 높은 스캔률은 파일 시스템을 많이 사용하는 시스템에서 항상 발견되는 현상이며, 이것이 메모리 부족 현상을 결정하는 요인으로 작용해서는 안된다. Solaris 7의 경우에는 메모리 부족 현상을 나타내주는 memstat 명령어를 이용하면 된다.

만일 파일 시스템 작업을 빈번하게 하는 경우 스캐너 패러미터가 충분치 않으면 파일 시스템의 성능이 제한된다. 이를 상쇄하기 위해서는 스캐너가 파일 시스템을 지속시키기에 충분할 정도의 빠른 스캔 속도를 유지할 수 있도록 스캐너 패러미터를 조정해줘야 한다. 스캐너는 기본으로 fastscan 패러미터에 의해 제한되는데, 이는 스캐너가 스캔할 수 있는 초당 페이지 숫자를 반영한다. 기본값으로는 초당 메모리의 1/4을 스캔하며, 초당 64MB까지 스캔할 수 있도록 제한되어 있다.

페이지 스캐너가 충분히 빠르게 스캐닝할 수 있도록 하려면 fastscan값을 증가시켜줘야 한다. 최고 초당 1GB에, 메모리의 1/4로 셋팅할 것을 권한다. 이것은 fastscan 패러미터를 131072로 설정하면 된다. Handspreadpages 패러미터는 fastscan 패러미터가 증가될 때마다 동일한 값으로 설정해줘야 한다.

파일 시스템의 또 다른 제한 요소는 maxpgio이다. Maxpgio 패러미터는 페이지 스캐너가 할 수 있는 페이지의 최대값이며, 파일 시스템의 쓰기 성능을 제한하면서 할 수 있는 파일 시스템 페이지의 합계로 제한되어 있다. 만일 시스템의 메모리가 충분하다면 maxpgio값은 좀더 큰 65536으로 설정할 것을 권한다. Sun Enterprise 10000 시스템에서는 이 값이 maxpgio의 기본값으로 설정되어 있다. 예를 들어 4GB의 시스템이라면 메모리의 1/4은 1GB이므로 fastscan값은 131072가 된다. 이 시스템의 패러미터는 다음과 같을 것이다.

```
*
* Parameters to allow better file system throughput
*
set fastscan=131072
set handspreadpages=131072
set maxpgio=65536
```

단 이러한 설정을 사용하는 스토리지가 어떤 것이냐에 따라서 더 크게 변경될 수도 있다(지난 호 기사 참조).

마치며

지금까지 Solaris가 가상 파일 캐시와 페이지 캐시를 어떻게 사용하는가를 살펴보았다. 즉 페이지 캐시는 메모리 시스템을 이용해 파일 시스템 캐시를 모아주며, 파일 시스템 캐시와 가상 메모리 시스템 패러미터 간에 밀접한 관계가 있다. 여기에 언급된 내용은 모두 Solaris 7까지만 적용된다. Solaris 8은 과도한 스캐닝에 따른 문제와 대형 메모리의 지원 free list의 합리적인 이용을 위해서 Cyclic Cashe라는 것을 적용했다. 이와 관련된 내용은 추후 기회가 되면 다루겠다.

참고로 이 글은 Richard Mc Dougall 씨가 쓴 Solaris 파일 시스템 글을 정리한 것이다.