지역성과 Fast File System

처음 UNIX가 등장했을때, UNIX의 창시자 Ken Thompson 자신이 첫 번째 파일 시스템을 개발하였다. 그 파일 시스템은 정말 단순해서 “오래된 UNIX 파일 시스템”이라 불린다. 디스크 상의 구조는 다음과 같다.

파일 시스템 전체에 대한 정보를 가진 슈퍼블럭(S)이 앞부분에 있다. 볼륨 크기, 아이노드 개수, 프리 블럭 리스트의 헤드를 가리키는 포인터 등이 존재한다. 모든 아이노드들은 디스크의 아이노드 영역에 존재한다. 마지막으로 디스크의 대부분의 영역은 데이터 영역으로 할당되어 있다.

이 구형 파일 시스템의 장점은 단순하다는 것이며, 파일 시스템의 가장 기본적인 개념인 파일과 디렉터리만을 제공한다. 비록 간단한 파일 시스템이기는 하지만, 그 이전에 있었던 레코드 기반의 저장 시스템과 비교하면 크게 발전한 것이다. 특히, 유닉스 파일 시스템의 계층 구조의 디렉터리는 그 이전 시스템들이 사용했던 단일 계층 디렉터리 구조와 비교하면 실로 엄청난 혁신이라 할수 있다.

44.1 문제 : 낮은 성능

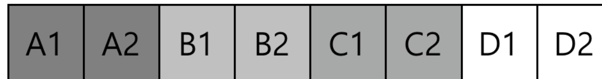
문제는 성능이 형편없다는 것이다. 버클리의 Kirk McKusick과 그의 동료들의 측정 결과에 따르면 [McK+84], 성능은 처음부터 안 좋게 시작해서 시간이 지남에 따라 더 안 좋아져서 나중에는 파일 시스템이 디스크의 전반적인 대역폭의 2%뿐이 사용 못하는 결과를 보인다고 한다!

구형 파일 시스템의 핵심 문제는 디스크를 마치 임의 접근 기억 장치(RAM)처럼 사용한다는 것이다. 데이터를 저장하는 매체가 디스크라는 사실을 무시하고 여기저기에 데이터를 저장하고 있기 때문에 디스크 헤드를 이동시키는 데 많은 시간이 소요된다. 예를 들어, 파일의 데이터 블럭이 대체적으로 아이노드에서 멀리 떨어져 있기 때문에 아이노드를 읽은 후 파일의 데이터 블럭을 접근하려면 디스크 헤드를 이동하는 데 많은

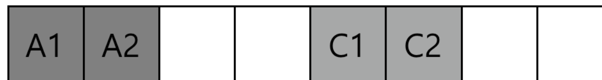
시간을 소요하게 된다. 아이노드를 읽은 후 해당 데이터 블럭을 읽는 작업은 사실 매우 흔하게 일어나는 작업이다.

더 안 좋은 것은 파일 시스템이 빈 공간을 효율적으로 관리하지 않기 때문에 결국에는 공간은 **단편화**(**fragment**) 된다. 빈 공간들이 디스크 전역에 흩어져 있으며, 새로운 블럭 할당 시 무조건 리스트에서 다음 빈 블럭을 할당한다. 그 결과 파일을 순차적으로 읽더라도 실제로는 디스크 전역을 오가며 블럭을 접근하기 때문에 성능이 심각하게 나빠진다.

예를 들어 다음과 같은 데이터 블럭 영역을 생각해 보자. 네 개의 파일(A, B, C 그리고 D)이 있으며 각 2개의 블럭의 크기를 갖는다.



B와 D가 삭제되면 다음과 같은 상태가 된다.



비어있는 공간은 보는 바와 같이 네 개의 블럭이 연속된 청크가 아닌 두 블럭 크기로 단편화되었다. 이제 네 개의 블럭으로 구성된 파일 E를 할당한다고 해 보자.

E를 이루는 블럭들은 디스크 상에서 흩어져 있게 된다. E를 읽거나 쓸 경우, 디스크로부터 최대(순차) 성능을 얻을 수 없다. 먼저 E1과 E2를 읽고, 디스크 헤드를 이동시켜 E3과 E4를 읽어야 한다. 오래된 UNIX 파일 시스템에서 이와 같은 단편화는 흔하게 발생하였다. 성능에 악영향을 준다(조각 모음 도구의 역할이 이 문제를 해결하는 것이다. 파일 블럭들을 이동하여 순차적으로 위치시킨다. 빈 공간들이 연속적으로 배치되도록 데이터 블럭들을 이동하고 변경 사항을 아이노드 등에 반영한다).

또 다른 문제가 있다. 블럭 크기가 너무 작다(512 바이트). 입출력 단위가 너무 작기 때문에 디스크로부터 데이터를 전송하는 것은 원천적으로 매우 비효율적이다. 작은 크기의 블럭은 **내부 단편화**(**internal fragmentation**, 블럭 내의 낭비)가 작은 장점이 있다. 그러나, 디스크 헤드의 이동시간에 비해 데이터 블럭의 크기가 작기 때문에 입출력이 상대적으로 비효율적이 된다. 문제를 다음과 같이 정리할 수 있다.

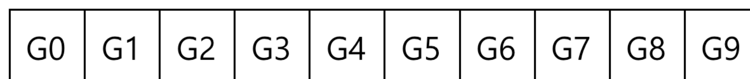
핵심 질문: 성능 개선을 위해 어떻게 디스크 상의 데이터를 구성해야 할까
성능을 위해 파일 시스템의 자료 구조를 어떻게 구성해야 할까? 이러한 자료 구조에는 어떤 종류의 할당 정책이 필요할까? 어떻게 파일 시스템에 디스크 특성을 반영할까?

44.2 FFS: 디스크에 대한 이해가 해답이다

Berkeley의 한 그룹이 좀 빠르고 좋은 파일 시스템을 만들기로 결정하였다. 그리고 독창적으로 **Fast File System**, **FFS**라고 명명하였다. 파일 시스템의 자료 구조와 할당 정책을 "디스크에 적합"[1]하게 설계하여 성능을 개선하는 것이 핵심이었다. 실제로 그들은 목적을 달성하였다. FFS는 파일 시스템 연구의 새로운 장을 열었다. 파일 시스템 인터페이스는 그대로 유지하면서(`open()`, `read()`, `write()` 그리고 `close()`와 다른 파일 시스템 콜을 포함), 내부 구현 방식을 변경하였다. 이 연구를 통해 파일 시스템 개발에 새로운 지평을 열었고, 그 길은 지금까지도 사용되고 있다. 거의 모든 현대의 파일 시스템은 기존의 인터페이스를 그대로 사용하면서(응용 프로그램과의 호환성을 유지하기 위해) 성능과 신뢰성 또는 기타 다른 목적으로 내부를 최적화하고 있다.

44.3 파일 시스템 구조: 실린더 그룹

첫 번째 단계는 디스크에 적합하게 파일 시스템 구성을 수정하는 것이었다. FFS는 디스크를 여러 개의 **실린더 그룹**(**cylinder group**, 현대의 Linux의 ext2와 ext3에서는 **블럭 그룹**(**block group**)으로 부르기도 함)으로 분할하였다. 실린더 그룹이 열 개 있는 디스크를 다음과 같이 생각해 볼 수 있다.



실린더 그룹은 FFS의 핵심 중의 핵심이다. 두 개의 파일을 같은 그룹에 배치시켜서, 한 파일을 읽고, 다음의 파일을 접근할 때 디스크 전역에 걸친 긴 탐색이 발생하지 않도록 하였다. FFS는 파일과 그 파일이 속한 디렉터리 블럭을 같은 그룹에 할당하였다. 물론 그룹이 꽉 차면 밀려나는 경우도 발생한다. 각 그룹은 다음과 같이 구성된다.

이제 실린더 그룹의 구성 요소를 설명하겠다. **슈퍼블럭**의 복사본(S)이 각 그룹마다

1) 역자 주: 여기서의 디스크는 하드디스크다. 불과 수년 전만 하더라도, 디스크라 하면 자연스럽게 하드디스크를 의미했었다.

여담 : FFS에서 파일 생성

파일이 생성될 때 어떠한 자료 구조들이 변경되어야 하는지 생각해 보자. 이 예제에서는 사용자가 `/foo/bar.txt`라는 파일을 생성하고 블럭 하나 크기 (4 KB) 라고 하자. 새로운 파일이기 때문에 새로운 아이노드가 필요하다. 아이노드 비트맵과 새로 할당받은 아이노드가 디스크에 기록된다. 파일에는 데이터도 있기 때문에 데이터 비트맵과 데이터 블럭도 자연스레 디스크에 기록되어야 한다. 그렇게 하여 최소한 네 번의 쓰기가 현재의 실린더 그룹에서 발생한다(이 쓰기들은 실제 기록이 되기 전에 버퍼에 잠시 동안 있을 수 있다). 하지만 이게 전부는 아니다! 파일 생성을 구체적으로 살펴보면 파일을 파일 시스템 계층에 저장하여야 하므로 디렉터리도 같이 갱신되어야 한다. 여기서는 `bar.txt`에 대한 항목을 추가하기 위해 부모 디렉터리인 `foo`가 갱신되어야 한다. 이 갱신은 `foo`가 있는 데이터 블럭에 같이 기록될 수도 있고 또는 새로운 블럭을 할당받아야 할 수도 있다(해당하는 데이터 비트맵과 함께). 변경된 디렉터리의 크기와 갱신 시간 필드(마지막 변경 시간과 같은)를 반영하기 위해 `foo`의 아이노드 역시 갱신되어야 한다. 전체적으로 단순히 파일 하나를 생성하기 위해 많은 작업이 필요하다! 다음에 당신이 파일을 만들 때는 좀 더 감사해야 할 것이다. 최소한 모든 것이 다 잘 동작하는 것에 대해 놀라야 할 것이다.

존재한다 (예, 하나가 훼손되더라도 다른 것을 사용하여 파일 시스템을 마운트하고 접근할 수 있다).

각 그룹 내에서 아이노드와 데이터 블럭들의 할당 여부를 알 수 있어야 한다. 그룹별로 **아이노드 비트맵**(ib) 과 **데이터 비트맵**(db) 이 존재한다. 각 그룹의 아이노드와 데이터 블럭의 할당 정보를 나타낸다. 비트맵은 파일 시스템의 빈 영역을 나타내는 훌륭한 방법이다. 큰 크기의 연속된 빈 공간을 쉽게 찾을 수 있다. 프리 리스트 사용시 발생하는 파일 시스템의 단편화 문제도 어느 정도 피할 수 있다.

마지막으로 아이노드와 데이터 블럭 영역은 아주 단순한 파일 시스템에서 봤던 것과 동일하다. 각 실린더 그룹의 대부분도 마찬가지로 데이터 블럭들로 이루어져 있다.

44.4 파일과 디렉터리 할당 정책

그룹의 구조가 결정되었다. 이제 성능 개선을 위해 파일과 디렉터리 그리고 관련된 메타데이터를 디스크에 어떻게 배치할지를 결정해야 한다. 기본 원칙은 간단하다. *관련있는 것끼리 모아라* (당연한 얘기지만, 관련없는 것들은 멀리 배치하여라).

이 원칙을 따르기 위해서는, 우선 무엇이 "관련있는"지를 파악해야 하고 그것들을 같은 블럭 그룹 내에 저장한다. 반대로 관련없는 항목들은 서로 다른 블럭 그룹에 저장한다. 이를 달성하기 위해서 FFS는 몇 가지 힌트를 사용한다.

첫 번째는 디렉터리의 위치 결정에 관한 것이다. FFS는 단순한 방법을 사용한다. 할당된 디렉터리의 수가 적고(그룹들 간에 디렉터리에 대한 균형을 맞추기 위해) 프리 아이노드의 수가 많은 실린더 그룹을 선택하여 디렉터리 데이터와 아이노드를 해당

그룹에 저장한다. 물론 다른 방법을 따를 수도 있다(예, 프리 데이터 블럭의 수를 고려할 수 있음).

파일의 경우 FFS는 두 가지 방법을 사용한다. 하나는 (일반적인 경우에) 아이노드와 파일의 데이터 블럭을 같은 그룹에 할당하여 아이노드와 데이터 간의 긴 탐색을 방지한다 (오래된 파일 시스템과 같은 방식으로). 두 번째로 동일한 디렉터리 내의 모든 파일들은 해당 디렉터리가 존재하는 실린더 그룹에 함께 저장한다. 네 개의 파일을 `/dir1/1.txt`, `/dir1/2.txt`, `/dir1/3.txt`와 `/dir99/4.txt`라는 이름으로 생성할 때 FFS는 첫 세 개의 파일은 서로의 근처(같은 그룹)에 저장하고 네 번째의 것은 멀리 떨어 뜨려 놓는다(다른 그룹에).

유의할 것은 이러한 방식은 파일 시스템 사용에 관한 완벽한 분석이나 그와 유사한 무엇을 통해서 나온 것은 아니라는 것이다. 대신에 상식(common sense)에 기반하였다 (그게 CS의 약자 아닌가?). 디렉터리 내의 파일들은 *대체적*으로 같이 접근이 된다(여러 파일을 컴파일하여 하나의 실행 파일로 링킹하는 것을 생각해 보자). 관련 파일들 간의 탐색이 짧아지기 때문에 FFS는 성능을 개선할 수 있다.

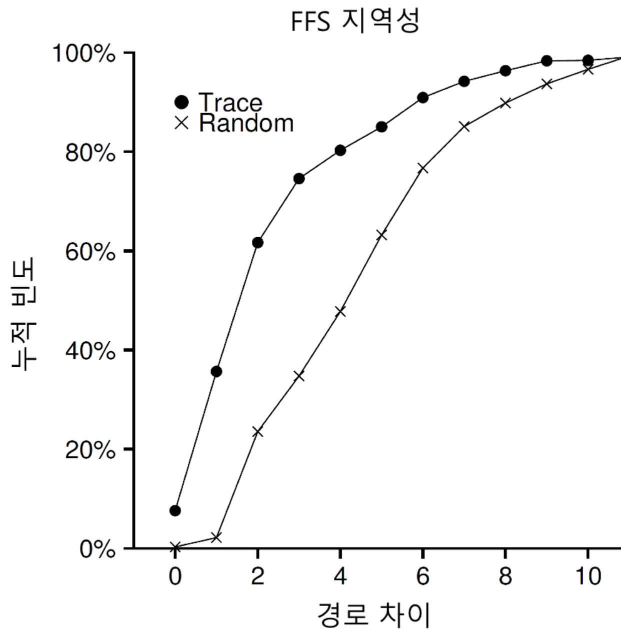
44.5 파일 접근의 지역성 측정

이러한 경험적 상식에 근거한 접근이 정당한지를 알기위해 namespace 지역성이 실제 존재하는지를 알아보자. 무슨 연유인지는 모르겠으나 선행 문헌에는 이 주제를 제대로 다룬 연구가 없는 것으로 보인다.

구체적으로 살펴보기 위해서 SEER 트레이스 [Kue94]를 사용하여 디렉터리 트리 내에서 파일 간의 접근이 얼마나 "멀리 떨어져"있는지를 분석하였다. 이를 경로 간의 거리라 하겠다. 예를 들어 파일 `f`가 열렸고 잠시 후 다시 그 파일이 열렸다면(다른 파일이 열리기 전에) 디렉터리 내에서 이 두 번 open 간의 거리는 0이다(같은 파일이므로). 만약 디렉터리 `/dir` 내의 파일 `f`가 열렸고(즉, `dir/f`) 같은 디렉터리에 파일 `g`가 열렸다면 (즉, `dir/g`), 파일 open 간의 거리는 같은 디렉터리 내에 있지만 다른 파일이므로 1이다. 여기서 사용되는 거리 기준을 다시 정리하면, 두 파일의 *조상 경로*명이 동일하게 될 때까지 디렉터리 트리를 얼마나 타고 올라가야 하는가이다. 트리 상에서 가까이 있을수록 낮은 값을 갖게 된다.

그림 44.1은 SEER 클러스터에 속해 있는 여러 워크스테이션들의 전체의 트레이스를 합하여 지역성을 나타낸 것이다. 경로 간의 거리를 x축에, 이에 대한 누적 빈도를 y축에 표현했다. SEER 트레이스에서 (그래프에 "Trace"라고 표기) 약 7% 가 직전에 열었던 파일을 다시 열었다. 거의 40%의 경우가, 같은 디렉터리 내의 파일 또는 같은 파일을 열었다(0과 1의 경로 차이를 나타냄). FFS 가 가정한 지역성은 이치에 맞는 것으로 보인다(최소한 이 트레이스의 경우).

흥미롭게도 25% 정도는 경로가 2만큼 떨어져 있었다. 이 현상은 사용자가 서로 관련된 디렉터리들을 묶어두고, 이들 디렉터리에 존재하는 파일들을 번갈아 접근할 때 발생하는 것으로 판명되었다. 예를 들어 `src` 디렉터리가 있고 목적 파일(object file, `.o` 파일)은 `obj` 디렉터리에 넣는다고 해 보자. 이 두 디렉터리들은 `proj` 디렉터리의 하위



〈그림 44.1〉 **SEER 트레이스의 FFS 지역성**

디렉터리로 proj/src/foo.c 이후에 proj/obj/foo.o를 접근하는 패턴을 보인다. 이들 디렉터리는 proj안에 존재한다. 두 접근 간의 경로 차이는 2가 된다. FFS는 이러한 지역성까지는 고려하지 않기 때문에 이러한 접근 간에는 탐색이 많이 발생하게 된다.

비교를 위해서 "랜덤" 트레이스에 대해서 지역성이 어떻게 되는지를 살펴보았다. 기존의 SEER 트레이스에서 임의의 순서로 파일들을 선택하여 랜덤 트레이스를 생성하였다. 랜덤 트레이스에서는 기대한 것과 같이 이름 공간의 지역성이 낮은 것을 볼 수 있다. 하지만, 결국 모든 파일은 동일한 조상(예, 루트)을 갖기 때문에 약간의 지역성은 보인다. 랜덤 트레이스는 비교 목적으로 유용하다.

44.6 대용량 파일 예외 상황

파일 배치 시 예외 상황이 있다. 파일 크기가 매우 큰 경우이다. 별도의 규칙을 적용하지 않으면 하나의 파일이 블럭 그룹(그리고 다른 블럭들)을 모두 채울수 있다. 바람직하지 않다. 왜냐하면 그 뒤의 "관련"있는 파일들은 다른 블럭 그룹에 저장되도록 만들어 파일 접근 지역성을 떨어뜨리기 때문이다.

FFS는 큰 파일들의 배치를 다음과 같이 처리한다. 첫 번째 블럭 그룹에 일정 수의 블럭을 할당한 후에(예, 블럭 12개 또는 아이노드의 직접 포인터 개수만큼) FFS는 파일의 "큰" 청크(예, 첫 번째 간접 블럭이 가리키고 있는 블럭들)를 다른 블럭 그룹(사용률이 낮은 것으로 선택)에 저장한다. 그리고 파일의 다음 청크는 마찬가지로 또

다른 블럭 그룹에 저장한다.

이 정책을 더 잘 이해하기 위해서 그림을 살펴보자. 큰 파일에 대한 예외가 없다면 하나의 큰 파일의 모든 블럭들을 디스크의 한 곳에 다 모아 저장할 것이다. 예로 블럭을 10개 사용하는 파일의 경우에 어떻게 저장되는지를 시각적으로 나타내었다.

다음의 그림이 FFS가 큰 파일에 대한 예외 처리 없이 저장한 경우이다.

G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
		0 1 2 3 4							
		5 6 7 8 9							

큰 파일에 대한 예외 처리를 하면 다음과 같이 파일은 디스크 전반에 걸쳐 청크 단위로 저장되는 것을 볼 수가 있다.

G0	G1	G2	G3	G4	G5	G6	G7	G8	G9
89		01		23		45		67	

예리한 독자라면 파일 블럭들을 디스크에 흩어놓으면 상대적으로 흔한 순차 파일 접근과 같은 경우에 성능이 좋지 않게 된다는 것을 알것이다(예, 사용자나 응용 프로그램이 청크를 0부터 9까지 순서대로 읽는 경우). 당신이 맞다! 성능이 나빠진다. 청크의 크기를 적절히 선택하는 것으로 이 문제를 약간 경감시킬 수 있다.

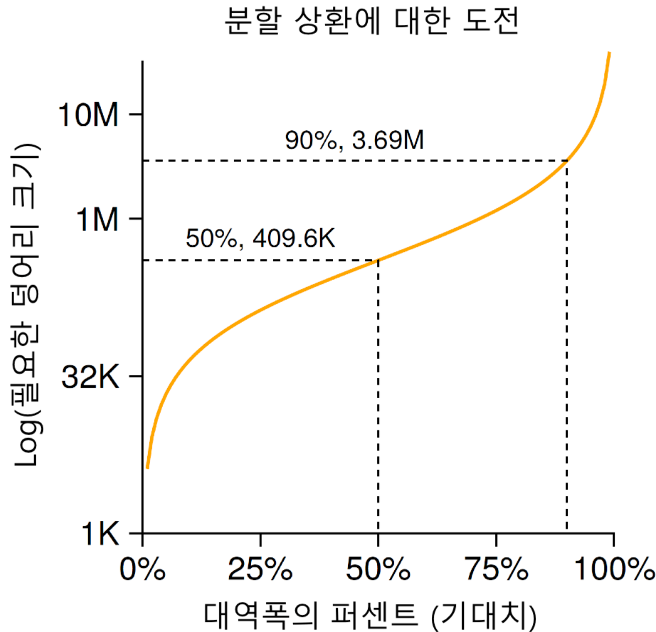
만약에 청크의 크기가 충분히 크다면, 디스크로부터 데이터를 전송하는 데 대부분의 시간을 사용하고, 상대적으로 적은 시간을 블럭 청크들 간의 탐색에 사용하게 될 것이다. 오버헤드 비용당 더 많은 작업을 처리하여 오버헤드를 줄이는 것을 보고 **점진적 경감(amortization)** 기법이라고 부르며 컴퓨터 시스템에서는 흔한 기술이다.

예제를 하나 살펴보자. 디스크의 평균 위치 잡기 시간(즉, 탐색과 회전 시간)이 10 msec이고, 디스크가 데이터를 40 MB/s의 속도로 전송한다고 해 보자. 전체 시간 중 탐색에 반을 사용하고 나머지를 데이터 전송에 쓰는(그래서 최대 디스크 성능의 50%를 얻는 것)이 목적이라고 한다면 위치 잡기를 위해서 10 msec를 쓰고 데이터를 전송하는 데 10 msec를 사용해야 한다. 이 경우 우리의 질문은 다음과 같다. 전송하는 데 10 msec를 사용하려면 청크는 얼마나 커야 할까? 간단하다, 디스크에 대한 장에서 차원 해석을 설명할 때 사용했던 우리 수식으로 풀면 된다.

$$\frac{40\ \text{MB}}{\text{sec}} \cdot \frac{1024\ \text{KB}}{1\ \text{MB}} \cdot \frac{1\ \text{sec}}{1000\ \text{msec}} \cdot 10\ \text{msec} = 409.6\ \text{KB} \tag{44.1}$$

이 식이 말하는 것은 다음과 같다. 만약 40 MB/s의 속도로 데이터를 전송이 가능할 때 시간의 반은 전송에, 반은 탐색에 쓰려고 한다면 매 탐색마다 409.6 KB의 데이터를 전송해야 한다는 말이다. 비슷한 방법으로 최대 대역폭의 90%를 달성하기 위한 청크의 크기도 계산할 수가 있으며(계산하면 3.69 MB), 또는 대역폭의 99%가 되도록 만들

때도 계산할 수 있다(40.6 MB!). 최대 대역폭에 가까운 성능을 원할수록 청크의 크기가 커지는 것을 볼 수가 있다(이 값들을 표현한 그림 44.2를 살펴보자).



〈그림 44.2〉 **분할 상환: 청크의 크기는 얼마나 커야 하는가?**

그렇지만, FFS는 큰 파일을 그룹들 간에 분산하는 데 있어 이와 같은 계산을 사용하지 않았다. 대신에, 아이노드 자체의 구조를 기반으로 하는 간단한 접근법을 사용하였다. 첫 번째 열두 개의 직접 블럭은 아이노드와 같은 그룹에 배치하였고, 각 간접 블럭과 그것이 가리키는 모든 블럭들은 다른 그룹에 배치 하였다. 블럭 크기가 4 KB이고 32비트 디스크 주소를 사용한다고 할 때 이 전략은 직접 포인터가 가리키는 파일의 첫 48 KB만 예외로 할당되고 파일의 매 1024개의 블럭(4 MB)마다 다른 그룹에 배치가 된다.

디스크 드라이브의 전송 속도는 빠르게 증가하고 있다. 제조사가 같은 면적에 더 많은 비트를 저장하는 기술을 개발하고 있기 때문이다. 물론 좋아지기는 하지만 탐색과 연관 있는 드라이브의 기계적인 측면(디스크 암 속도와 회전 속도)으로 인해 그 증가 정도는 아주 빠르지는 않다 [Pat98]. 이것은 시간이 갈수록 기계적 비용이 상대적으로 좀 더 비싸진다는 것을 뜻하며, 그 비용을 점진적으로 경감하기 위해서는 한번에 더 많은 데이터를 전송해야 한다는 것이다.

44.7 FFS에 대한 기타 사항

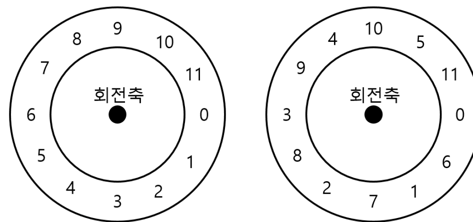
FFS는 혁신적인 기법을 개발하여 적용했다. 특히 작은 파일들을 효율적으로 저장하기 위해 많은 노력을 했다. 그 당시에는 2 KB 크기의 파일들이 많았다. 4 KB 블럭을

사용하였기 때문에 데이터를 전송하는 것에는 이득이 있었지만 공간 활용면에서는 비효율적이었다. 일반 파일 시스템에서는 **내부 단편화(internal fragmentation)**로 인해 대략 반 정도의 공간이 낭비되었다.

FFS 설계자들은 이 문제를 간단하게 해결하였다. 파일 시스템이 파일을 할당할 수 있도록 512 byte의 **서브블럭(sub-block)** 개념을 도입하였다. 작은 파일(1 KB 크기)을 생성할 때, 두 개의 서브블럭을 할당함으로써 4 KB 블럭 전체가 낭비되는 경우가 발생하지 않도록 하였다. 파일이 커지면, 파일 시스템은 4 KB가 될 때까지 512 byte 크기의 블럭을 추가로 할당한다. 서브블럭들의 합이 4 KB가 되는 시점에 FFS는 4 KB 블럭을 찾아서 서브블럭들을 *복사*해 넣은 후에 해당 서브블럭들을 다시 사용할 수 있도록 해제한다.

파일 시스템의 추가 동작이 많이 발생하기 때문에 이 과정이 비효율적이라고 생각할 수도 있다(정확히는 많은 I/O가 복사 때문에 발생한다). 이번에도 당신이 맞았다! FFS는 이러한 경우를 피하기 위해 `libc` 라이브러리를 수정하였다. 라이브러리는 쓰기 요청을 버퍼에 두었다가 4 KB 청크가 되면 파일 시스템에 내리는 식으로 대부분의 서브블럭을 사용하는 특수한 경우를 완전히 제거할 수 있도록 하였다.

FFS에서 개발 채택된 또 하나의 기발한 아이디어가 있다. 성능에 최적화된 디스크 배치이다. 그 당시에는(SCSI 또는 좀 더 근대화된 장치 인터페이스 이전의) 디스크가 그렇게 정교하지가 않았으며 호스트 CPU가 장치의 동작을 직접 제어했다. FFS의 문제는 그림 44.3의 왼편과 같이 디스크에 연속적인 섹터들에 파일을 저장해야 할 때 발생했다.



〈그림 44.3〉 **FFS: 표준 대 매개화된 배치**

순차 읽기 시 문제가 발생한다. FFS가 먼저 블럭 0번에 대한 읽기를 요청하고 그 요청이 끝나면 다음 블럭, 즉 블럭 1에 대한 읽기를 요청한다. 이미 늦었다. 블럭 1은 헤드를 지나갔다. 1번 블럭을 읽기 위해서는 한 바퀴를 다 돌아야 한다.

FFS는 이 문제를 그림 44.3의 오른쪽 그림과 같이 디스크의 배치를 바꾸는 식으로 해결하였다. 하나 건너에 블럭을 배치하여(이 예제의 경우) FFS는 다음 요청에 대한 블럭이 헤드 아래로 지나가기 전에 해당 블럭을 요청할 충분한 시간을 벌 수가 있었다. 사실 FFS는 추가 회전을 피하기 위해서 특정 디스크의 *몇 개*의 블럭을 건너뛰어서 배치할지 알 만큼 영리하였다. FFS는 디스크의 특정 성능 매개변수들을 검출해 낸 후에, 그 정보를 활용하여 정확한 시차에 따라 배치 기법을 결정할 수 있도록 하였다. FFS는 이 기법을 **매개화(parameterization)**라 불렀다.

팁 : 시스템을 유용하게 만들어라

FFS 가 큰 의미를 갖는 이유에는 두 가지가 있다. 첫째는 디스크를 고려한 배치방식의 도입이고, 둘째는 시스템을 쉽게 사용할 수 있도록 하는 기능들의 추가이다. 긴 파일 이름, 심볼릭 링크, 그리고 원자적으로 이름을 변경하는 명령 같은 것들이 시스템을 더욱 유용하게 만들었다. 각 주제들로 연구 논문을 쓰기는 어렵겠지만(“심볼릭 링크 : 하드 링크가 오랫동안 잃어버렸던 사촌”이라는 14장짜리 논문을 읽는 것을 상상해 보라), 그런 작은 기능들이 FFS를 좀 더 유용하게 만들었고 널리 사용되도록 하였다. 시스템을 쓰기 쉽게 만드는 것은 심오한 기술혁신만큼 또는 그보다 더 중요하다.

이 기법이 그렇게 대단치 않다고 생각할 수 있다. 사실, 이러한 배치 기법을 사용하면 최대 대역폭의 50%밖에 얻을 수가 없다. 왜냐하면 각 블럭을 한 번씩 읽기 위해서 같은 트랙을 두 바퀴 돌기 때문이다. 다행스러운 것은 현대의 디스크들은 훨씬 똑똑하다는 것이다. 내부적으로 한 트랙을 모두 내부 디스크 캐시에 버퍼링한다(보통 그러한 이유로 **트랙 버퍼**라고 불린다). 해당 트랙에 연달아 읽기 요청이 들어오면 디스크는 캐시에서 원하는 값을 리턴한다. 파일 시스템은 그러므로 더 이상 하위수준의 상세한 부분에 대해서는 신경쓰지 않아도 된다. 추상화와 상위 레벨 인터페이스는 제대로 설계된다면 좋은 것이 될 수 있다.

사용자 편의성을 개선하기 위한 기능들도 추가되었다. FFS는 **긴 파일 이름**을 지원하는 최초의 파일 시스템 중 하나이다. 전통적인 고정 길이 방식(예, 여덟 글자) 대신에 좀 더 표현력 있는 이름을 사용할 수 있게 되었다. 또, **심볼릭 링크** 개념을 도입한 최초의 파일 시스템이다. 이전 장에서 설명하였듯이 하드 링크는 디렉터리를 가리킬 수 없으며(파일 시스템에 순환 구조를 만들 수 있다는 우려 때문에), 같은 볼륨 내의 파일들만 가리킬 수 있다는(즉, 아이노드 번호가 의미를 갖는 공간) 제약이 있었다. 심볼릭 링크는 사용자가 시스템 내의 파일 또는 디렉터리에 상관없이 “가명(alias)”을 만들 수 있도록 허용하기 때문에 훨씬 더 융통성이 있다. FFS는 파일의 이름을 원자적으로 변경하는 `rename()` 명령도 소개하였다. 기본적인 기술 이외에도 사용성을 개선한 기법들로 인해서 FFS는 두터운 사용자 층을 얻을 수 있었다.

44.8 요약

FFS의 등장은 파일 시스템의 역사 상 분수령과 같은 순간이었다. 파일 관리의 문제가 운영체제 내에서 가장 흥미로운 주제 중 하나라는 것을 밝혔기 때문이며 장치들 중에서 가장 중요한 하드 디스크를 다루는 방법을 선보였기 때문이다. 그 이후로, 수백 개의 새로운 파일 시스템들이 개발되었지만 많은 파일 시스템들이 FFS에 근간을 두었다(예, Linux ext2와 ext3도 후손들이다). 모든 현대 시스템들은 디스크를 디스크처럼 대하라는 FFS의 핵심 가르침을 따르고 있다.

참고 문헌

[Kue94] **"The Design of the SEER Predictive Caching System"**
G.H. Kuenning
MOBICOMM '94, Santa Cruz, California, December 1994
Kuenning에 따르면 SEER 프로젝트에 대한 개론 중 최고이다. 이 프로젝트에서(많은 결과가 있었지만) 트레이스의 컬렉션이 나오게 되었다.

[McK+84] **"A Fast File System for Unix"**
Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry
ACM Transactions on Computing Systems.
FFS를 구현한 그의 업적을 근간으로 파일 시스템에 대한 공헌을 인정받은 McKusick은 최근 IEEE Reynold B. Johnson 상을 수상하였다. 그의 수상 연설에서 1,200 줄 밖에 안 되는 코드로 이루어진 원래의 FFS 소프트웨어에 대해서 설명을 하였다. 현대의 것들은 좀 더 복잡하다. 예로 BSD의 FFS는 이제 5만 줄 가량의 코드로 이루어져 있다.

[Pat98] **"Hardware Technology Trends and Database Opportunities"**
David A. Patterson
Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)
훌륭하고 간단한 디스크 기술의 동향에 대한 개론으로, 시대에 따라 어떻게 변하는지를 보여준다.