43

파일 시스템 구현

이번 장에서는 **vsfs**(**Very Simple File System**)라고 하는 간단한 파일 시스템 구현에 대해 소개하겠다. 이 파일 시스템은 UNIX 파일 시스템을 단순화한 것으로 디스크 자료 구조(on-disk structure)와 접근 방법 그리고 다양한 파일 시스템들의 정책들을 소개하기 위해 제작되었다.

파일 시스템은 순수한 소프트웨어다. 이점이 이책의 앞부분에서 다룬 CPU 가상화, 메모리 가상화 부분과 다른점이다. CPU 가상화나 메모리 가상화에서는 하드웨어가 필요하다. 특권모드로의 변환을 위한 명령어, 페이징을 위한 하드웨어 등이 그것이아. 파일 시스템에서는 성능 개선을 위해 하드웨어를 추가하지 않을 것이다(물론, 잘 동작하는 파일 시스템을 만들려면 장치 특성을 반영해야 한다.). 파일 시스템을 구현하는 다양한 방법이 존재하기 때문에 문자 그대로 AFS(Andrew 파일 시스템) [How+88]부터 시작하여 ZFS(Sun의 Zettabyte(10^{21}) 파일 시스템) [BM07]까지 매우 다양한 파일 시스템들이 개발되었다. 이 모든 파일 시스템들이 서로 다른 자료 구조를 갖고 있으며 각각은 장단점이 있다. 사례 연구를 통해 파일 시스템을 학습할 것이다. 먼저 이번 장에서 간단한 파일 시스템(vsfs)을 사용하여 개념을 소개하고, 파일 시스템에 대한 실제 사례들을 다루면서 현실에서는 어떻게 동작하는지 이해해 보도록 하겠다.

핵심 질문: 어떻게 간단한 파일 시스템을 만들 것인가

간단한 파일 시스템을 어떻게 만들 수 있을까? 디스크 위에는 어떤 자료 구조가 필요할까? 그러한 자료 구조는 어떤 정보를 추적해야 하는가? 그 자료 구조들은 어떻게 접근되어야 하는가?

43.1 생각하는 방법

파일 시스템에 대해 학습할 때, 두 가지 측면에서 접근할 것을 권장한다. 그 두 측면을 다 이해하게 되면 파일 시스템이 기본적으로 어떻게 동작하는지 이해하게 될 것이다.

첫 번째는 파일 시스템의 **자료 구조**이다. 즉, 파일 시스템이 자신의 데이터와 메타데이터를 관리하기위애 디스크 상에 어떤 종류의 자료 구조가 있어야 하겠는가? 우리가

여담: 파일 시스템의 개념적 모델

이전에도 언급했듯이 시스템을 배우려고 할 때에 실제로 하는 일은 모델을 잘 만드는 것이다. 파일 시스템을 이해하기 위한 개념적 모델에는 다음과 같은 질문들이 자연스레 포함되게 될 것이다. 디스크 상의 어떤 자료 구조가 파일 시스템의 데이터와 메타데이터를 저장하는가? 프로세스가 파일을 열면 무슨 일이 일어나는가? 읽기 또는 쓰기를 할 때에는 디스크 상의 어떤 자료 구조가 접근될까? 파일 시스템의 코드를 읽어서 구체적인 부분을 이해하려고 노력하는 대신(물론, 이것도 유용하기는 하지만), 개념적 모델을 확장시키려고 노력하면, 일어나는 현상에 대하여 이해를 더 쉽게 할 수 있다.

보게 될 첫 파일 시스템(vsfs를 포함하여)은 블럭과 다른 객체들을 배열과 같은 간단한 자료 구조로 표현하지만, SGI의 XFS와 같은 파일 시스템들은 좀 더 복잡한 트리 기반의 자료 구조를 사용한다 [Swe+96].

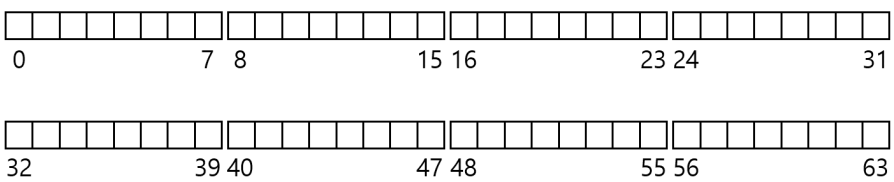
파일 시스템의 두 번째 측면은 **접근 방법**(**access method**)이다. 프로세스가 호출하는 `open()`, `read()`, `write()` 등의 명령들은 파일 시스템의 자료 구조와 어떤 관련이 있는가? 특정 시스템 콜을 실행할 때에 어떤 자료 구조들이 읽히는가? 어떤 것들이 쓰일까? 이 모든 과정이 얼마나 효율적으로 동작하는가?

파일 시스템의 자료 구조와 접근 방법을 이해하였다면, 실제 동작 방식에 대한 개념 모델을 제대로 정립하는 것이다. 그것이 시스템적 사고 방식의 핵심이다. 우리가 파일 시스템을 처음으로 구현해 갈 때, 개념적 모델을 잘 만들려고 노력하기 바란다.

43.2 전체 구성

이제 vsfs 파일 시스템의 자료 구조에 대해 디스크 상의 전체적인 구성을 개발해 보자. 가장 먼저 해야 할 것은 디스크를 **블럭**(**block**)들로 나누는 것이다. 간단한 파일 시스템은 단일 블럭 크기만 사용하기 때문에 여기서도 그렇게 하겠다. 일반적으로 사용되는 크기인 4 KB를 사용하겠다.

우리가 파일 시스템을 만들려는 디스크 파티션 구조는 단순하다. 4 KB 블럭들로 나열되어 있다. N개의 4 KB 블럭의 크기를 갖는 파티션에서 블럭은 0부터 $N-1$까지의 주소를 갖고 있다. 블럭이 64개 있는 작은 디스크를 가정하자.



파일 시스템을 생성하기 위해서 이들 블럭에 어떤 것을 저장할지 생각해 보자. 가장 먼저 떠오르는 것은 사용자 데이터이다. 사실, 파일 시스템의 대부분의 공간은 사용자 데이터로 이루어져 있다(또한 그렇게 되어야 한다). 사용자 데이터가 있는 디스크 공간을

데이터 영역(**data region**)이라고 하자. 간단하게 64개의 디스크 블럭 중 마지막 56개의 블럭처럼 디스크의 일정 부분을 데이터 영역으로 확보하자.

앞에서 (약간) 배웠던 것을 기억해 보면 파일 시스템은 각 파일에 대한 정보를 관리한다. 그 정보가 **메타데이터**(**metadata**)의 핵심이다. 파일을 구성하는 데이터 블럭(데이터 영역 내의)들과 그 파일의 크기, 소유자, 접근 권한, 접근과 변경 시간 등과 같은 정보들이 이에 해당한다. 파일 시스템은 이 정보를 보통 **아이노드**(**inode**)라고 부르는 자료 구조에 저장한다(아래에서 더 자세하게 다룰 것이다).

아이노드들의 저장을 위해 디스크 공간이 필요하다. 이 영역을 **아이노드 테이블**(**inode table**)이라 한다. 아이노드 테이블에는 **아이노드**들이 배열형태로 저장된다. 디스크 상의 자료 구조의 배치는 다음의 그림에서 보는 것과 같다. 전체 64개의 블럭 중 5개의 블럭이 아이노드를 저장하기 위해 할당된 것으로 가정하자(I로 그림에 표기되어 있다).

아이노드는 일반적으로 128에서 256 bytes 정도로 그렇게 크지 않다. 아이노드당 256 bytes를 가정하면 4 KB 블럭에는 16개의 아이노드를 저장할 수 있으며 우리 파일 시스템에서는 총 80개의 아이노드를 저장할 수 있다. 64개의 블럭으로 구성된 작은 파일 시스템 파티션에 최대 80개의 파일을 만들수 있다는 것을 의미한다. 같은 파일 시스템이라도 더 큰 디스크 파티션에 생성한다면 아이노드 테이블을 더 크게 만들 수 있기 때문에 생성가능한 최대 파일 개수를 증가시킬 수 있다.

우리의 예제 파일 시스템에는 데이터 블럭(D)과 아이노드 블럭(I)이 존재한다. 아직 완전하지 않다. 꼭 필요한 정보는 아이노드나 데이터 블럭의 사용여부에 대한 것이다. 각 블럭이 현재 사용 중인지 아닌지를 표현할 **할당 구조**(**allocation structure**)가 필요하다. 블럭이 사용 중인지 아닌지를 나타내는 정보는 어느 파일 시스템이든 꼭 있어야 한다.

블럭이 사용 중인지 아닌지를 표현하는 데에는 다양한 방법이 존재한다. 예를 들면 **프리 리스트(free list)**를 사용하여, 사용 중이 아닌 블럭들을 링크드 리스트 형태로 관리할 수 있다. 아이노드는 첫 번째 프리 블럭의 위치만 기억하면 된다. 다음 프리 블럭의 위치는 각 프리 블럭에서 정해진 위치에 기록한다. 우리는 단순한 **비트맵(bitmap)**을 사용한다. 데이터 영역에 있는 블럭들의 사용여부를 표현하기 위해서 **데이터 비트맵(data bitmap)**을, 아이노드 테이블에 있는 아이노드들이 사용 중인지를 나타내기 위해서 **아이노드 비트맵(inode bitmap)**을 사용한다. 비트맵은 비트들의 배열이다. 각 비트는 해당 블럭이나 객체(우리의 경우는 각 아이노드에 해당한다.)가 사용 중인지(1) 아닌지(1)를 나타낸다.

우리의 초소형 파일 시스템의 경우 하나의 4 KB 블럭(4096 바이트, 32 Kbit) 전체를 비트맵으로 사용하는 것은 공간의 낭비다. 4 KByte 크기의 비트맵이면 32 K개의 객체에 대한 할당 정보를 관리할 수 있다. 우리 예제 파일 시스템에는 아이노드가 80개, 데이터 블럭이 56개 있다[1]. 하지만 이해를 편하게 하기위해 넘어가자. 각 비트맵은 4 KB 블럭을 전부 사용하도록 한다.

집중력 있는 사람은(아직까지 안 졸았다면) 남은 한 블럭의 존재를 파악했을 것이다. 뭐지? 이 블럭은 **슈퍼블럭(superblock)**을 위한 공간이다. 그렇다. 전지전능한 수퍼맨의 그 "수퍼"이다. 그림에서는 S로 표기하였다. 슈퍼블럭은 이 파일 시스템 전체에 대한 정보를 담고 있다. 예를 들면 파일 시스템에 몇 개의 아이노드와 데이터 블럭이 있는지(이 경우에는 각각 80개와 56개가 있음), 아이노드 테이블은 어디에서 시작하는지(블럭 3번) 같은 정보를 담고 있다. 파일 시스템을 식별할 수 있는 매직 넘버도 갖고 있을 것이다(이 경우, vsfs). 파일 시스템이 깨진다는 것은 수퍼블럭이 저장된 디스크 블럭이 훼손되는 것이다. 일반적으로 대부분의 파일 시스템은 수퍼블럭을 몇 개 복사해둔다.

1) 역자 주: 136 비트면 충분하다. 커널 개발자에게는 1 비트 조차도 금같이 귀하다.

파일 시스템을 마운트할 때, 운영체제는 우선 슈퍼블럭을 읽어들여서 파일 시스템의 여러가지 요소들을 초기화하고, 그 후에 각 파티션을 파일 시스템 트리에 붙히는 작업을 진행한다. 이렇게 함으로 해서 디스크 볼륨[2]에 있는 파일들을 접근할 때, 해당 파일을 읽거나 쓰는 데 필요한 자료 구조의 위치를 파악한다.

43.3 파일 구성: 아이노드

파일 시스템의 디스크 자료 구조 중 가장 중요한 것은 **아이노드(inode)**이다. 거의 모든 파일 시스템들이 비슷한 구조를 갖고 있다. 아이노드는 **인덱스 노드(index node)**의 줄임말로서 역사적으로는 UNIX의 창시자인 Ken Thompson [RT74]이 처음 사용하였다. 이 노드들은 원래는 배열로 되어 있었는데, 각 배열은 특정 아이노드를 접근하기 위해 *탐색*된다.

여담: 자료 구조 — 아이노드

파일 메타데이터를 저장하기 위한 자료 구조로서 많은 파일 시스템들이 **아이노드(inode)**라는 일반적인 이름을 사용하고 있다. 이 자료 구조는 파일 크기, 접근권한 그리고 파일 블럭들의 위치 정보를 가지고 있다. 이름은 최소한 UNIX 초기 시절부터 사용되었다(그 이전의 시스템 부터는 아닐지라도 Multics 시절까지 거슬러 올라갈 수도 있을 것이다). **인덱스 노드(index node)**의 약어로서, 디스크 상의 아이노드의 배열의 인덱스로 아이노드 번호를 사용하던 것에서 기인한다. 이제 보게 될 것처럼, 아이노드의 설계는 파일 시스템 설계의 핵심 사항 중 하나이다. 대부분의 현대 시스템들은 관리하는 모든 파일에 대해서 이와 같은 자료 구조를 사용하고 있다. 그렇지만, 이름은 다를 수 있다(dnode, fnode 등으로 불린다).

각 아이노드는 숫자(**아이넘버(inumber)**라고 불림)로 표현된다. 앞 장에서는 이것을 파일의 **저수준 이름(low-level name)**이라고 불렀었다. vsfs(그리고 다른 간단한 파일 시스템)에서는 아이넘버를 사용하여 해당 아이노드가 디스크 상에 어디에 있는지를 직접적으로 계산할 수 있다. 위에서 사용한 vsfs의 아이노드 테이블을 예로 사용해 보자. 크기가 20 KB이고(4 KB 블럭 5개) 80개의 아이노드로 이루어져 있다(각 아이노드는 256bytes라고 가정). 아이노드 영역은 12 KB 위치부터 시작하고(슈퍼블럭은 0 KB 위치에서 시작, 아이노드 비트맵의 주소는 4 KB에서부터, 데이터 비트맵은 8 KB에서부터 시작한다. 아이노드 테이블은 그 바로 직후에 있음). vsfs 파일 시스템 파티션의 시작 부분은 아래의 그림과 같이 구성되어있다.

32번 아이노드를 읽기 위해서는 파일 시스템은 아이노드 영역에서의 오프셋을 계산한다($32 \cdot sizeof(inode)$ 또는 8192). 그 후 아이노드 테이블의 시작 위치를 더하면 (`inodeStartAddr=12 KB`), 원하는 아이노드 블럭의 정확한 바이트 주소를 구할 수 있다(`20 KB`). 디스크는 바이트 단위로는 접근이 불가능하며 대신에 대체적으로 512

2) 역자 주: 볼륨이라는 단어가 등장했다. 단일 파일 시스템 파티션로 이해하자.

바이트 크기를 갖는 섹터(sector)로 이루어졌다. 32번 아이노드가 존재하는 블럭을 가져오기 위해서는 파일 시스템은 섹터 주소 $\frac{20\times1024}{512}$ 또는 40에 대한 읽기 요청을 하여 해당 아이노드 블럭을 가져온다. 아이노드 블럭의 섹터 주소 iaddr은 다음과 같이 계산될 수 있다.

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

아이노드에는 파일에 대한 정보가 다 들어 있다. 파일의 *종류*(예, 일반 파일, 디렉터리 등), *크기*, 할당된 *블럭* 수, *보호 정보*(파일의 소유, 접근 권한 등), *시간* 정보와 더불어 데이터 블럭이 디스크 어디에 존재하는지(예, 포인터의 일종)와 같은 정보들이 담겨 있다. 이와 같은 파일에 대한 정보들을 **메타데이터(metadata)**라고 한다. 사용자 데이터가 아닌 기타 정보를 통틀어서 흔히 메타데이터라 부른다. ext2 [Poi07]의 아이노드 예를 그림 43.1에 나타내었다[3].

아이노드를 설계 시 가장 중요한 결정 중 하나는 데이터 블럭의 위치를 표현하는 방법이다. 간단한 방법은 아이노드 내에 여러 개의 **직접 포인터(direct pointer**, 디스크 주소)를 두는 것이다. 각 포인터는 파일의 디스크 블럭 하나를 가리킨다. 이 방법에는 제약이 있다. 파일 크기의 제한이 있다. 파일 크기가 (포인터의 개수)*(블럭 크기) 로 제한된다.

멀티 레벨 인덱스

큰 파일을 지원하기 위해서 파일 시스템 개발자들은 아이노드 내에 다른 자료 구조를 추가해야 했다. 일반적으로 사용되는 방법 중 하나는 **간접 포인터(indirect pointer)**라는 특수한 포인터를 사용하는 것이다. 간접 포인터는 데이터 블럭을 가리키지 않는다. 간접 포인터가 가리키는 블럭에는 데이터 블럭을 가리키는 포인터들이 저장된다. 직접 포인터와 간접 포인터를 결합해서 사용할 수 있다. 아이노드에는 정해진 수의 직접 포인터(예, 12개), 그리고 하나의 간접 포인터가 있다. 큰 파일에 대해서는 간접 블럭이 할당이 되고(디스크의 데이터 블럭 영역에서), 아이노드의 간접 포인터는 이 간접 블럭을 가리킨다. 블럭이 4 KB이고 디스크 주소가 4 바이트라고 하면 1024개의 포인터들을 추가할 수 있게 된다. 최대 파일 크기는 $(12 + 1024) \cdot 4$ K 또는 4144 KB가 된다.

3) 파일 유형에 관한 정보는 디렉터리 항목에 저장되므로 아이노드 자체에서는 찾을 수 없다.

크기	이름	이 아이노드 필드가 나타내는 정보
2	mode	파일이 읽기/쓰기/실행될 수 있는가?
2	uid	파일은 누가 소유하는가?
4	size	파일의 크기는 몇 바이트인가?
4	time	파일이 마지막 접근된 시간은?
4	ctime	파일이 생성된 시간은?
4	mtime	파일이 마지막으로 변경된 시간은?
4	dtime	아이노드가 삭제된 시간은?
2	gid	파일이 속한 그룹은?
2	links_count	파일에 대한 하드 링크의 수는?
4	blocks	이 파일에 몇 개의 블럭이 할당되었는가?
4	flags	ext2가 이 아이노드를 사용하는 방법은?
4	osd1	운영체제 종속적인 필드
60	block	디스크 포인터들의 집합(총 15개)
4	generation	파일의 버전(NFS에 의해 사용)
4	file_acl	모드 비트 이후의 새로운 권한 모델을 나타냄
4	dir_acl	접근 제어 목록(access control lists)이라 불림

〈그림 43.1〉 **Ext2의 간략한 아이노드**

4144 KByte는 사실 그리 큰 파일이 아니다. 더 큰 파일을 저장하고 싶을 수도 있다. 그 경우 아이노드에 **이중 간접 포인터**(**double indirect pointer**)를 추가한다. 이중 간접 포인터가 가르키는 블럭에는 간접 포인터들이 저장되어 있다. 각 간접 블럭은 데이터 블럭을 가리키는 포인터들을 가리키고 있다. 이중 간접 블럭을 사용하면 파일은 4 KB*1024*1024, 약 백만개의 4KByte 블럭을 가질수 있다. 즉, 4 GB가 넘는 크기의 파일을 지원할 수 있게 된다. 더 큰 파일을 표현해야 한다면, 다음에는 무엇이 나올지 감을 잡았을 것이다. **삼중 간접 포인터**(**triple indirect pointer**)를 사용하면 된다.

이제까지 설명한 것을 종합하면, 디스크 블럭들은 일종의 트리 형태로 구성되어 하나의 파일을 이룬다. 약간 한쪽으로 치우쳐진 형태의 트리다. 이러한 구성방식을 **멀티 레벨 인덱스** 기법이라 한다. 열두 개의 직접 포인터와 하나의 간접 블럭과 이중 간접 블럭을 사용하는 예를 살펴보자. 블럭 크기가 4 KB라고 하고 각 포인터의 크기가 4바이트라고 하자. 이 구조에서 파일은 4 GB가 조금 넘는 크기를 가질 수 있다(즉, $(12 + 1024 + 1024^2) \times 4\,\mathrm{KB}$). 삼중 간접 블럭을 사용하는 경우 최대 파일 크기가 어떻게

팁 : 익스텐트 기반의 접근법을 고려해 보자

포인터를 사용하는 대신 **익스텐트(extent)** 방식을 사용할 수도 있다. 익스텐트는 단순히 디스크 포인터와 길이(블럭 단위)로 이루어진다. 파일의 모든 블럭에 대해서 포인터를 써야 하는 대신에 디스크 상의 한 파일의 위치를 가리키기 위해서 하나의 포인터와 길이만 표현하면 된다. 디스크 상에 충분히 크고 연속적인 비어 있는 공간을 찾는 것이 어려울 수 있기 때문에 하나의 익스텐트만을 갖는 것은 제한적일 수 있다. 그렇기 때문에 익스텐트 기반의 파일 시스템은 하나 이상의 익스텐트를 갖는 것을 허용하여 파일을 할당할 때 좀 더 자유도가 높아질 수 있도록 한다.

두 기법을 비교하면 포인터 기반의 접근법은 가장 자유도가 높지만 각 파일당 많은 양의 메타데이터를 사용한다(특히 큰 파일들의 경우). 익스텐트 기반의 접근법은 자유도가 낮은 대신에 좀 더 집약되어 있다. 특히, 디스크에 여유 공간이 충분히 있고 파일들을 연속적으로 배치할 수 있을 때 잘 동작한다(거의 대부분의 파일 할당 정책의 목적이기는 하다).

되는지 계산해 볼 수 있겠는가?(힌트 : 엄청나게 크다) [4]

일반적으로 사용되는 파일 시스템인 Linux의 ext2 [Poi07]와 ext3, NetApp의 WAFL을 포함하여, 많은 파일 시스템들이 멀티 레벨 인덱스를 사용한다. SGI의 XFS와 Linux의 ext4와 같은 파일 시스템들은 간단한 포인터를 사용하는 대신 익스텐트를 사용한다. **익스텐트(extent)** 기반의 동작 방법은 앞의 여담에 소개되어 있다(가상 메모리의 논의에서 나온 세그멘트와 유사하다).

재미있는 사실은 트리의 형태가 매우 편향적이다. 파일의 시작 부분을 이루는 블럭들은 한 번의 포인터로 접근이 가능하다. 큰 파일의 경우, 파일의 끝부분에 있는 블럭들은 포인터를 세 번 따라가야 실제 블럭을 읽을 수 있다. 왜 이렇게 했을까? 실수 아닌가? 실수가 아니다. 오랜 고민과 연구의 결과이다. 많은 연구자들이 파일 시스템 사용형태를 분석해서 발견한 아주 중요한 "사실"이 있다. *대부분의 파일의 크기는 작다*는 것이다. 비대칭적 트리형태를 채용한 이유는 이러한 특성을 반영한 것이다. 대부분의 파일들이 작다면, 작은 파일을 빨리 읽고 쓸 수 있도록 파일구조를 설계해야 한다. vsfs의 아이노드는 첫 12개의 블럭들은 빨리 읽을 수 있도록 직접 포인터(대체적으로 12개)를 갖고 있다. 큰 파일들을 위해서 하나 (또는 그 이상)의 간접 블럭이 필요하다. 최근의 연구 결과는 Agrawal 등 [Agr+07]을 참고하자. 그림 43.2에 그 결과를 정리하였다.

파일은 위에서 다룬 방법말고도 다양한 방식으로 구성할 수 있다. 아이노드는 자료구조다. 데이터와 관련 내용들을 효율적으로 읽고 쓸 수 있다면, 어떤 방식도 사용가능하다. 파일 시스템 소프트웨어는 손쉽게 변경이 가능하다. 워크로드 특성의 변화에 따라, 새로운 파일 구성 방식을 연구개발할 자세가 되어 있어야 한다.

4) 역자 주 : 우리가 사용하는 리눅스의 EXT4 파일 시스템은 삼중 간접 포인터까지 가지고 있다.

대부분의 파일들의 크기가 작음	일반적인 파일 크기는 대략 2 KB
평균 파일 크기가 커지고 있음	평균 파일 크기는 거의 200 KB
큰 파일들이 대부분의 바이트를 사용함	몇 개의 큰 파일이 공간의 대부분을 채우고 있음
파일 시스템에는 많은 파일들이 있음	평균적으로 거의 100K가 있음
파일 시스템은 대략 반쯤 사용됨	디스크 용량이 커져도 대략 파일 시스템의 50%만 사용됨
디렉터리의 크기가 대체적으로 작음	대부분 20개 또는 그 이하의 적은 항목을 가짐

〈그림 43.2〉 **파일 시스템 측정 결과 정리**

43.4 디렉터리 구조

vsfs의 디렉터리는 (다른 많은 파일 시스템에서 그러하듯이) 간단하다. 디렉터리는 (항목의 이름, 아이노드 번호) 쌍의 배열로 구성되어 있다. 디렉터리의 데이터 블럭에는 문자열과 숫자가 쌍으로 존재하며 문자열 길이에 대한 정보도 있다(가변 길이의 이름을 가정함).

예를 들어 `dir`이라는 디렉터리(아이노드 번호 5)에는 `foo`, `bar`와 `foobar`라는 3개의 파일이 있고 각각의 아이노드 번호는 12, 13 그리고 24라고 하자. `dir`의 데이터 블럭은 아래와 같은 내용을 가지고 있을 것이다.

```
inum  |  reclen  |  strlen  |  name
  5         4          2        .
  2         4          3        ..
 12         4          4        foo
 13         4          4        bar
 24         8          7        foobar
```

각 항목은 아이노드 번호와 레코드 길이(이름에 사용된 총 바이트와 남은 공간의 합), 문자열 길이(실제 이름의 길이), 그리고 마지막으로 항목의 이름을 갖고 있다. 각 디렉터리는 . "dot"과 .. "dot-dot"이라는 두 개의 추가 항목이 있으며 dot 디렉터리는 현재 디렉터리(이 예제에서는 `dir`을)를 가리키며 dot-dot은 부모 디렉터리(이 경우에는 루트)를 가리킨다.

파일이 삭제되면(예, `unlink()` 호출) 디렉터리 중간에 빈 공간이 발생한다. 영역이 비었다는 것을 표시할 방법이 필요하다(예, 0번 아이노드는 삭제된 파일을 나타내기로 약속한 후 아이노드 번호를 0으로 쓰는 식). 항목의 길이를 명시하는 이유 중에 하나가 중간에 빈 공간이 생기기 때문이다. 새로운 디렉터리 항목을 생성할 때, 기존 항목이 삭제되어 생긴 빈 공간에 새로이 생성된 항목을 위치시킬 수도 있기 때문이다.

디렉터리들은 정확히 어디에 저장될까? 대부분 파일 시스템에서 디렉터리는 특수한 종류의 파일로 간주한다. 디렉터리는 자신의 아이노드를 가지며, 이 아이노드는 아이노드 테이블에 존재한다(아이노드의 type 필드에 "일반 파일"대신에 "디렉터리"라고

여담: 링크 기반의 파일 구조

아이노드를 설계하는 데 사용되는 또 다른 간단한 방법은 **연결 리스트**를 사용하는 것이다. 아이노드 안에 다수의 포인터를 두지 않고 파일의 첫 번째 블럭을 가리키는 포인터만 하나 둔다. 큰 파일의 경우, 파일의 마지막 블럭을 가리키는 포인터를 추가한다.

예상했을 수도 있겠지만, 링그 기반의 파일 블럭 구성은 특정 워크로드에 대해서 성능이 좋지않을 수 있다. 예를 들면 가장 마지막 블럭을 읽어야 하는 경우나 임의의 위치를 읽어야 하는 경우이다. 링크드 리스트 기반 구성의 성능을 개선하기 위해 데이터 블럭 내에 다음 번 블럭의 위치를 저장하지 않고, 링크드 리스트의 포인터 정보들만 테이블로 관리하기도 한다. 그러면, 이 테이블을 메모리에 상주시킬 수 있다. 다음 블럭의 주소를 각 블럭에 저장할 경우, 10번째 블럭을 읽기 위해서는, 첫 블럭부터 열번째 블럭까지를 차례로 읽어야 한다. 각 블럭의 위치를 테이블에 모아둘 경우, 이 테이블이 메모리에 존재하면, 각 블럭을 따로 읽는 연산은 발생하지 않는다. 테이블 항목은 데이터 블럭 `D`의 주소로 인덱스된다. 각 항목의 내용은 `D`의 다음 블럭을 가리키는 포인터, 즉 파일의 `D` 다음에 오는 블럭의 주소를 저장한다. NULL 값을 갖을 수도 있으며(파일의 끝을 나타냄), 또는 어떤 표식을 사용하여 특정 블럭이 비어 있음을 나타낼 수도 있다. 다음 포인터를 저장하는 테이블을 사용하면 연결식 할당 방법에서도 임의의 파일 접근을 효율적으로 할 수 있게 된다. 먼저 (메모리 내의) 테이블을 스캔하여 원하는 블럭을 찾은 후 (디스크 내의) 해당 위치를 직접 접근하면 된다.

이러한 테이블이 익숙하게 들리는가? 방금 설명한 내용이 그 유명한 **file allocation table**) 또는 **FAT** 파일 시스템의 기본 구조다. 그렇다, 지금은 고전이 된 NTFS 이전의 옛 Windows 파일 시스템 [Cus94]으로서 간단한 연결 방식의 할당 방법을 사용하였다. 표준 Unix 파일 시스템들과의 다른 점도 있다. 아이노드가 없다. 파일의 메타데이터는 디렉터리 항목에 저장된다. 때문에 하드 링크를 만드는 것은 불가능하다. 별로 세련된 것 같지는 않다. 그러나, 가장 광범위하게 사용되었던 파일 시스템이다. 상세 사항을 보기 원하면 Brouwer [Bro02]를 참고하자.

명시되어 있다). 디렉터리는 자신의 데이터 블럭을 갖고 있으며, 이들 블럭의 위치는 일반 파일과 마찬가지로 아이노드에 명시되어 있다. 이 데이터 블럭들은 데이터 블럭 영역에 존재한다.

이 예제에서 디렉터리는 가변 크기 레코드로 구성된 배열이다. 이 외에도 많은 방식이 존재할 수 있다. 앞에서도 언급했듯이 제대로 돌아만 간다면 어떤 자료 구조라도 상관없다. 예를 들면 XFS [Swe+96]는 디렉터리를 B-tree 형식으로 구성한다. 파일 생성 시 현재 디렉터리에 동일한 이름의 파일이 있는지를 먼저 검사해야 한다. 디렉터리 항목들이 선형배열로 구성되어 있는 경우, 항목들을 모두 검색해야 한다. B-tree와 같은 검색 전용 자료구조를 사용할 경우, 검색시간이 단축된다. 파일 생성을 좀 더 빨리 할 수 있다(단, 이전에 동일한 이름으로 파일이 생성된 적이 없어야 하는 조건이 있다).

여담: 빈 공간의 관리

빈 공간을 관리하는 여러 가지 방법이 있는데 그 중에 하나가 비트맵이다. 초기의 파일 시스템들은 **프리 리스트**(**free list**)를 사용하여 슈퍼블럭 안의 한 포인터가 첫 번째 프리 블럭을 가리키도록 하였다. 그리고 그 블럭은 다른 프리 블럭을 가리키는 포인터를 갖는 식으로 시스템 내의 프리 블럭들의 리스트를 만들었다. 블럭이 하나 필요하면 리스트의 헤드를 사용하고 슈퍼블럭은 그 다음의 블럭을 프리 리스트의 헤드가 되도록 한다.

현대의 파일 시스템은 좀 더 정교한 자료 구조들을 사용한다. 예를 들어 SGI의 XFS의 경우 **B-tree**의 일종을 사용하여 디스크의 어느 공간이 비어 있는지를 작은 크기로 표현한다. 어느 자료 구조를 사용하든 시간과 공간 사이의 절충이 가능하다.

43.5 빈 공간의 관리

파일 시스템은 아이노드와 데이터 블럭 사용 여부를 관리해야 한다. 그래야 새로운 파일이나 디렉터리를 할당할 공간을 찾을 수 있다. **빈 공간 관리**(**free space management**)는 모든 파일 시스템에서 중요하다. vsfs에서는 두 개의 비트맵을 사용한다.

파일 생성 시 아이노드를 할당해야 한다. 파일 시스템은 아이노드 비트맵을 탐색하여 비어 있는 아이노드를 찾아 파일에 할당한다. 파일 시스템은 해당 아이노드를 사용 중으로 표기하고(1로 표기) 디스크 비트맵도 적절히 갱신한다. 이와 유사한 동작이 데이터 블럭을 할당할 때에도 일어난다.

데이터 블럭 할당 시 고려해야 할 사항이 또 있다. 예를 들면, ext2와 ext3와 같은 Linux 파일 시스템의 경우 데이터 블럭 할당 시 가능하면 여러 개의 블럭들이 연속적으로 비어 있는 공간을 (예를 들면, 여덟 개) 찾아서 할당한다. 추후에 할당요청이 발생하면 기존에 할당된 공간에 이어서 블럭을 할당하기 위해서이다. 연속적으로 여러 개의 블럭들이 비어있는 공간을 할당함으로써 해당 파일에 대한 입출력 성능을 개선한다. 이러한 **선할당**(**pre-allocation**) 정책은 데이터 블럭 할당 시 자주 사용된다.

43.6 실행 흐름: 읽기와 쓰기

이제까지 파일과 디렉터리를 디스크에 저장하는 방법에 대해 학습했다. 이제 파일을 읽고 쓰는 세부 과정을 이해할 준비가 되었을 것이다. 되었어야만 한다. 이러한 **실행 과정**(**access path**)을 이해하는 것은 파일 시스템 동작을 완전히 이해하는 매우 중요한 두 번째 요인이다. 자, 집중하자!

다음의 예제에서는 파일 시스템은 마운트되었고 슈퍼블럭은 메모리 상에 위치한다고 가정하자. 다른 모든 것(예, 아이노드, 디렉터리)들은 디스크에 존재한다. 메모리에는 아직 탑재되지 않았다.

디스크에서 파일 읽기

간단한 예제로 단순히 파일을 열고(예, `/foo/bar`), 읽고, 읽은 후에 닫는 상황을 가정해 보자. 파일은 4KB의 크기(1개의 블럭)를 갖고 있다고 가정한다.

`open(`/foo/bar',`O_RDONLY)` 시스템 콜을 하면 파일 시스템은 먼저 파일 bar에 대한 아이노드를 찾아서 파일에 대한 기본적인 정보를 획득해야 한다(권한 정보, 파일의 크기 등). 파일 시스템은 아이노드를 찾아야 한다. 파일에 대한 전체 경로명를 갖고 있다. 파일 시스템은 경로명을 따라가서(**traverse**) 원하는 아이노드를 찾아야 한다.

경로명을 따라가는 것은 항상 파일 시스템의 루트에서 시작하며, **루트 디렉터리**(**root directory**)는 `/`로 표기된다. 파일 시스템이 디스크에서 가장 먼저 읽을 것은 루트 디렉터리의 아이노드이다. 루트 디렉터리의 아이노드는 어디에 있는가? 아이노드를 찾기 위해서는 i-number를 알아야 한다. 일반적으로 어떤 파일이나 디렉터리의 i-number는 부모 디렉터리에서 찾을 수 있다. 루트는 부모가 없다(정의상 없음). 루트 i-number는 "잘 알려진" 것이어야 한다. 파일 시스템이 마운트될 때 이 값이 결정된다. 대부분의 Unix 파일 시스템에서는 루트 디렉터리의 아이노드 번호는 2번이다. 파일 시스템은 아이노드 번호 2번을 포함하는 블럭을 읽는다(첫 번째 아이노드 블럭).

파일 시스템은 읽어들인 아이노드에서 데이터 블럭의 포인터를 추출한다. 포인터가 가리키는 블럭에는 루트 디렉터리의 내용이 들어 있다. 파일 시스템은 이 포인터들을 사용하여 디렉터리 정보를 읽고, `foo`라는 항목을 찾는다. 디렉터리에 많은 파일이 들어있을 수 있다. 예를 들어 3,000개. 이 경우, 모든 항목을 하나의 블럭에 저장할 수 없다. 하나의 디렉터리를 표현하기 위해 다수의 블럭이 사용된다. 하나 또는 그 이상의 디렉터리 데이터 블럭을 읽어서 `foo`에 대한 항목을 찾을 수 있다. `foo` 파일의 디렉터리 항목을 찾아서, `foo`의 아이노드 번호(44라고 하자)를 파악한다. 우리는 이 아이노드(44번 아이노드)가 필요하다.

다음 순서는 경로명을 따라가서 원하는 아이노드를 찾는다. 이 예제에서 파일 시스템은 `foo`의 아이노드가 있는 블럭과 그에 대한 디렉터리 데이터를 읽은 후에 마침내 `bar`에 대한 아이노드 번호를 찾아낸다. 마지막 단계의 `open()`은 `bar`에 대한 아이노드를 메모리로 읽어 들인다. 파일 시스템은 최종적으로 해당 파일에 대한 접근 권한을 확인하고, 이 프로세스의 open file-table에서 파일 디스크립터를 할당받아 사용자에게 리턴한다.

`open()` 이후에는, `read()` 시스템 콜을 통해 파일을 읽는다. 첫 번째 읽기는 (`lseek()`가 호출되지 않았다면 오프셋은 0) 아이노드를 통해 해당 블럭의 디스크상의 위치를 파악한 후 해당 블럭을 읽는다. 첫 번째 읽기 작업이므로 첫 번째 블럭을 읽게될 것이다(파일 오프셋이 0일 경우). 파일을 읽은 후 파일을 마지막으로 읽은 시간을 아이노드에 기록한다. 파일 오프셋은 파일을 읽거나 쓸 때, 해당 작업을 수행할 위치를 저장하는 변수이다. `read()`는 open-file-table에서 해당 파일 디스크립터에 대한 오프셋을 갱신한다. 다음에 읽기 작업을 수행할 때, 이전에 읽었던 다음 위치부터(파일의 두 번째 블럭) 읽도록 할 것이다.

어느 시점이 되면 그 파일을 닫아야한다. 할 일이 그렇게 많지 않다. 할당된 파일 디스크립터를 해제하면 된다. 현재로서는 파일 시스템이 해야 할 일은 그것이 전부다. 디스크 I/O는 발생하지 않는다.

이 과정에 대한 모든 설명이 그림 43.3에 나타나 있다(시간은 아래 방향으로 증가한다). 이 그림에서는 파일을 여는 과정 중에 파일의 아이노드를 찾기 위해서 여러 번의 읽기가 일어나는 것을 보여주고 있다. 그 뒤에 각 블럭을 읽기 위해서 파일 시스템은 먼저 아이노드를 읽고 블럭을 읽는다. 그리고 아이노드의 마지막 접근 시간을 갱신한다. 무슨 일이 발생하는지 다시 한 번 차분히 이해해 보자.

	데이터 비트맵	아이노드 비트맵	루트 아이노드	foo 아이노드	bar 아이노드	루트 data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**			**read**				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

〈그림 43.3〉 **파일 읽기의 시간 흐름(아래 방향으로 시간 증가)**

I/O 발생 횟수는 경로의 길이에 비례하는 것을 유의하여 보자. 경로가 하나 추가될 때마다 아이노드와 해당하는 데이터를 읽어야 한다. 디렉터리의 수가 많아지면 상황은 더 악화될 것이다. 여기서는 한 블럭만 읽어서 디렉터리의 내용을 얻었지만, 디렉터리가 크다면 원하는 항목을 찾기 위해서 많은 데이터 블럭을 읽어야 할 것이다. 그렇다. 파일 읽기만으로도 인생이 고달파 질 수 있다. 곧 알게 되겠지만 파일 쓰기의 경우는(특히, 새로운 파일을 만드는 것은) 훨씬 더 고달프다.

디스크에 쓰기

디스크 쓰기도 비슷한 과정을 밟는다. 먼저 파일을 연다(열기에서 했던 것과 같이). 그 후에 응용 프로그램은 `write()`를 호출하여 새로운 내용으로 파일을 갱신한다. 최종적으로 파일을 닫는다.

읽기와는 다르게 파일 쓰기는 블럭 **할당**을 필요로 할 수 있다(기존 블럭의 내용이 갱신되는 것이 아니라면). 새로운 파일에 쓸 때에는 각 `write()`는 데이터를 디스크에 기록해야 할 뿐만 아니라 파일에 어느 블럭을 할당할지를 결정해야 하며 그에 따라

여담: 읽기는 할당 자료 구조를 접근하지 않는다

비트맵과 같은 할당 자료 구조에 대해서 혼돈스러워하는 학생들을 많이 보았다. 새로운 블럭을 할당하지 않고 단순히 파일을 읽기만 할 때에도 비트맵을 여전히 읽을 것이라고 많이들 생각한다. 그건 사실이 아니다! 비트맵과 같은 할당 자료 구조는 할당할 때만 접근할 필요가 있다. 아이노드와 디렉터리 그리고 간접 블럭들은 읽기 요청을 완료하는 데 필요한 모든 정보를 갖고 있다. 아이노드가 이미 어떤 블럭을 가리키고 있다면 그 블럭이 할당되었는지를 확인해야 할 필요가 없다.

디스크에 다른 자료 구조들을 갱신해야 한다(예, 데이터 비트맵과 아이노드). 그러므로 파일에 대한 쓰기 요청은 논리적으로 다섯 번의 I/O를 생성한다. 하나는 데이터 비트맵을 읽기 위해서(이후에 블럭이 사용됨에 따라 새롭게 할당된 블럭을 사용 중으로 표시하기 위해 갱신된다), 또 다른 하나는 비트맵을 쓰기 위해서(디스크에 새로운 상태를 반영하기 위해), 그 다음의 두 개는 아이노드를 읽고 쓰기 위해서(새로운 블럭의 위치를 반영하기 위해 갱신된다), 그리고 마지막으로 실제 블럭을 기록하기 위해서 I/O가 발생한다.

파일 생성과 같은 단순 작업에서 꽤 많은 양의 쓰기가 발생한다. 파일 생성 시, 파일 시스템은 아이노드를 할당하고, 새로운 파일을 위한 디렉터리 항목을 할당해야 한다. 이 과정을 위한 전체 I/O 발생 횟수는 상당하다. 하나는 아이노드 비트맵을 읽기 위해서(프리 아이노드를 찾기 위해), 하나는 아이노드 비트맵에 쓰기 위해(할당되었다는 것을 표시하기 위해), 또 다른 하나는 새로운 아이노드 자체를 쓰기 위해(초기화하려고), 그리고 다른 하나는 디렉터리의 데이터 블럭에 쓰기 위해(아이노드 이름과 상위 수준의 이름을 연결시키기 위해서), 그리고 한 번의 읽기와 쓰기는 디렉터리 아이노드를 읽고 갱신하기 위해서 I/O가 발생된다. 만약 새로운 파일을 저장하기 위해서 디렉터리 크기가 증가하게 되면, I/O(데이터 비트맵과 새로운 디렉터리 블럭을 위해)가 추가된다. 파일 하나 생성하는 데 이 모든 과정이 다 필요하다. 실로 엄청나지 않은가[5].

구체적인 예를 살펴보자. `/foo/bar`를 생성하고 그 안에 세 개의 블럭을 쓰는 예이다. 그림 43.4가 (파일을 생성하는) `open()`을 호출하여 세 개의 4 KB 쓰기를 하는 동안에 일어나는 일을 보여준다. 시스템 콜 별로 읽기와 쓰기를 그룹지어 놓았다. 파일 생성이 얼마나 대형 작업인지를 알 수 있다. 경로명을 따라가서 마침내 파일을 생성하기까지 10번의 I/O가 발생하였다. 각 `write()`는 5번의 IO를 발생시켰다. 각 `write()`는 새로운 파일 블럭을 필요로 한다. 이를 흔히 allocating write라 한다. 아이노드를 읽고 쓰기, 데이터 비트맵을 읽고 쓰기, 그리고 끝으로 데이터 쓰기, 총 5번이다. 어떻게 하면 이 전체 과정을 효율적으로 만들 수 있겠는가? 이제부터는 사업화의 영역과 점점 가까워진다.

5) 역자 주: 폰에 저장된 사진 500장을 무선 인터넷으로 클라우드에 올리는 작업을 생각해 보자. 500개의 파일을 순식간에 생성해야 한다.

	데이터 비트맵	아이노드 비트맵	루트 아이노드	foo 아이노드	bar 아이노드	루트 data	foo data	bar data[0]	bar data[1]	bar data[2]
			read							
						read				
				read						
							read			
create (/foo/bar)		read								
		write								
							write			
					read					
					write					
				write						
					read					
write()	read									
	write									
								write		
					write					
					read					
write()	read									
	write									
									write	
					write					
					read					
write()	read									
	write									
										write
					write					

〈그림 43.4〉 **파일 생성 과정 (아래 방향으로 시간 증가)**

핵심 질문: 파일 시스템의 I/O 비용을 어떻게 줄일까

파일 열기, 읽기 또는 쓰기와 같은 단순한 동작이 디스크 위 여기저기에 엄청나게 많은 I/O를 발생시킨다. 파일 시스템은 이렇게 많은 I/O에 대한 엄청난 비용을 줄이기 위해서 무엇을 할 수 있을까?

43.7 캐싱과 버퍼링

앞의 예제에서 볼 수 있듯이 파일을 읽고 쓰는 것은 많은 I/O를 발생시킨다. 컴퓨터의 전체 성능에 중요한 영향을 미친다. 성능개선을 위해 대부분의 파일 시스템들은 자주 사용되는 블럭들을 메모리(DRAM)에 캐싱한다.

앞의 예제에서 파일을 여는 과정을 생각해 보자. 캐싱을 하지 않는다면 파일을 여는 동작은 디렉터리 레벨마다 최소한 두 번의 읽기가 필요하다(디렉터리 아이노드 읽기와 디렉터리 데이터 읽기). 경로가 많은 수의 디렉터리로 구성된 경우(예, `/1/2/3/.../100/file.txt`) 파일 시스템은 파일을 여는 데 문자 그대로 수백 번의

팁: 정적 대 동적 파티션 이해하기

서로 다른 고객/사용자 간의 자원을 나누는 데 있어 **정적 파티션**(**static partitioning**)과 **동적 파티션**(**dynamic partitioning**)을 사용할 수 있다. 정적 방식은 고정된 크기로 자원을 한 번만 나눈다. 예를 들어 메모리를 사용할 수 있는 두 사용자가 있다고 해 보자. 한 사용자에게 메모리의 고정된 일부를 준 후에 나머지 모두를 다른 사용자에게 줄 수 있다. 동적 방식은 좀 더 유동적이라서 때에 따라 다른 양의 자원을 나누어 줄 수도 있다. 예를 들어 한 사용자가 일정 시간 동안 높은 퍼센트로 디스크 대역폭을 쓰고 있다고 해 보자. 잠시 후에 시스템은 사용 가능한 디스크 대역폭 중 많은 부분을 다른 사용자에게 할당하기로 결정할 수도 있다.

각 접근법은 각자의 장점이 있다. 정적 파티션은 각 사용자가 자원의 일부를 나누어쓰도록 하기 때문에 일반적으로 예상 가능한 성능을 얻을 수 있고, 대체적으로 구현하기도 쉽다. 동적 파티션은 활용률 측면에서 더 좋기는 하지만(자원은 많이 써야 하는 사용자가 어차피 남아 놀고 있는 자원을 사용하도록 하기 때문에) 구현하기에 더 복잡할 수 있다. 또한, 사용자가 갖고 있던 놀고 있는 자원을 다른 사용자가 사용할 수 있도록 줬지만, 실제 필요로 할 때에는 돌려받기 위해서 오랜 시간이 걸릴 수 있기 때문에 자원을 제공한 사용자의 입장에서는 성능이 나빠지게 된다. 대부분 그렇듯 최선의 방법은 없다. 대신에 현재 주어진 문제를 잘 생각하여 어떤 방식이 적합한지 결정하여야 한다. 사실, 항상 그렇게 해야 하지 않을까?

읽기를 수행한다!

초기의 파일 시스템에서는 자주 사용되는 블럭들을 저장하기 위해서 캐시를 도입하였다. 가상 메모리에서 논의했듯이 **LRU**와 기타 다른 캐시 교체 정책들이 캐시에 어떤 블럭들을 남길지 결정해야 한다. 이 **고정 크기의 캐시**는 일반적으로 부팅 시에 할당이 되며 전체 메모리의 약 10%를 차지한다.

하지만, 이런 메모리의 **정적 기법**은 낭비가 많다. 어떤 특정 시점에 파일 시스템이 메모리의 10%가 필요하지 않은 경우에는 어떻게 할까? 앞서 언급한 고정 크기 방식을 사용하면 파일 캐시에서 사용되지 않은 페이지들은 다른 목적을 갖도록 용도 변경을 할 수 없으므로 낭비가 된다.

그에 반해 현대의 시스템은 **동적 파티션** 방식을 사용한다. 현대의 많은 운영체제는 가상 메모리 페이지들과 파일 시스템 페이지들을 통합하여 **일원화된 페이지 캐시**(**unified page cache**)를 만들었다 [Sil00]. 이렇게 하면 어느 한 시점에 어느 부분에 더 많은 메모리가 필요하냐에 따라 파일 시스템과 가상 메모리에 좀 더 융통성 있게 메모리를 할당할 수 있다.

캐싱을 하는 경우에 파일 열기에 대해 알아보자. 첫 번째 열기는 디렉터리 아이노드와 데이터로 인해서 많은 읽기 I/O를 발생시키겠지만, 그 뒤를 따르는 같은 파일(또는 같은 디렉터리의 파일들)에 대한 파일 열기의 경우 캐시에서 히트가 되기 때문에 추가 I/O가 필요 없다.

쓰기 캐싱에 대한 영향력도 같이 알아보자. 캐시가 충분히 크면 대부분의 읽기 I/O를

제거할 수 있다. 쓰기는 영속성을 유지하기 위해서 해당 블럭들을 디스크로 내려야 한다. 쓰기의 경우에는 캐시가 읽기에서와 같은 필터 역할을 할 수가 없다. 캐시는 쓰기 시점을 연기하는 역할을 한다. 이를 **쓰기 버퍼링**(**write buffering**, 가끔 이렇게 불림)이라 한다. 쓰기 버퍼링을 통해 얻을 수 있는 몇 가지 성능 이득이 있다. 하나는 쓰기 요청을 지연시켜 다수의 쓰기 작업들을 적은 수의 I/O로 **일괄처리**(**batch**)할 수도 있다. 예를 들어 파일이 생성될 때 아이노드 비트맵이 갱신되었다고 하자. 잠시 후에 다른 파일이 또 생성되면서 아이노드 비트맵은 다시 갱신되었다. 이러한 경우에 파일 시스템은 첫 번째 갱신에 대한 쓰기를 연기하여, 두 번째 아이노드 비트맵에 대한 연산과 병합함으로써 I/O를 한개 줄일 수 있다. 두 번째는 여러 개의 쓰기요청들을 모아둠으로써 다수의 I/O들을 **스케줄**하여 성능을 개선할 수 있다. 마지막으로 지연시키는 것을 통해 쓰기 자체를 피할 수도 있다. 예를 들어 응용 프로그램이 파일을 생성한 후 즉시 삭제한다고 해 보자. 컴파일러는 컴파일 과정에서 수많은 임시파일을 생성하고 삭제한다. 이 경우에 쓰기를 지연시키면 애초에 디스크에 파일을 생성할 필요가 아예 없게 된다. 이러한 경우에는 게으른 것(디스크에 블럭을 쓰는 것에 대한)도 미덕이 된다.

위와 같은 이유로 대부분의 파일 시스템들은 쓰기 요청을 메모리에 약 5초에서 30초 정도동안 버퍼링한다. 물론 문제가 없는 것은 아니다. 쓰기 요청들이 디스크에 기록되기 전에 시스템이 크래시되면 내용은 손실된다. 메모리에 쓰기 내용을 오래 유지함으로써, 일괄처리, 스케줄링을 통해 성능을 개선할 수 있으며 때로는 쓰기 자체가 필요 없어지기도 한다.

어떤 응용 프로그램들은 (데이터베이스 등) 버퍼링으로 인해 발생하는 문제점을 용납하지 않는다. 방금 예금을 했는데, 전원이 차단되어 내가 저금한 금액이 통장잔고에 반영이 않되고 돈이 손실되었다고 생각해 보자. 쓰기 버퍼링으로 인한 예기치 않은 데이터 유실을 피하기 위해서 `fsync()`를 사용한다. 이를 호출하면 갱신된 내용이 디스크에 강제적으로 기록한다. 캐시를 사용하지 않도록 **direct I/O** 인터페이스를 사용하거나 **디스크**(**raw disk**) 인터페이스를 사용하여 파일 시스템을 건너뛰고 직접 디스크에 기록하는 경우도 있다[6]. 대부분의 응용 프로그램은 파일 시스템이 제공하는 버퍼링 기능을 사용하지만, 원치 않을 경우 시스템을 원하는 식으로 설정하는 충분한 제어 기능들이 있다.

43.8 요약

파일 시스템 개발에 필요한 기본 기법들을 살펴보았다. 각 파일에 어떤 정보(메타데이터)가 필요한지 살펴보면서 대부분은 아이노드라고 부르는 자료 구조에 저장한다는 것을 보았다. 디렉터리는 특수 파일로서 파일 이름과 아이노드 번호 간의 연결정보를 저장한다. 다른 자료 구조들이 더 있다. 예를 들면 파일 시스템은 아이노드나 데이터 블럭의 할당과 해제 여부를 나타내는 비트맵과 같은 정보를 갖고 있다.

6) 데이터베이스 수업을 수강해서 전통적인 데이터베이스에 대해, 그리고 운영체제를 배제하고 데이터베이스가 모든 것을 제어해야 한다는 과거의 주장에 대해 배워보라. 하지만 조심하자! 데이터베이스 사람들은 운영체제에 대해서 늘 비방하려 든다. 창피하지 않은가 데이터베이스 분야의 사람들아. 창피하다.

팁: 영속성과 성능 간의 절충

저장 장치 시스템에서 데이터의 영속성과 시스템 성능은 동전의 양면이다. 절충이 필요하다. 기록된 데이터가 바로 지속되기를 원한다면, 시스템은 새로 쓰인 데이터를 디스크에 커밋해야 한다. 시스템이 느려진다. 하지만, 데이터는 안전하게 보장된다. 사용자가 어느 정도의 데이터 손실을 감수할 수 있다면 시스템은 쓰기 요청을 일정 시간 메모리에 버퍼링한 후에 백그라운드로 디스크에 기록한다. 이렇게 하면 쓰기가 빨리 완료된 것처럼 보이기 때문에 체감 성능이 향상된다. 크래시가 일어나면 메모리에만 있고 디스크에 아직 커밋되지 않은 데이터는 손실된다. 어느 것이 옳은 방법인가? 제일 바람직한 방법은 응용 프로그램의 요구 사항을 명확히 이해하고 이를 만족시키는 방법을 채택하는 것이 최선이다. 예를 들어, 웹 브라우저가 다운로드한 마지막 몇 개의 이미지는 잃어버려도 괜찮지만, 은행 계좌에 입금하는 데이터베이스 트랜젝션의 일부를 잃어버리는 것은 허용되지 않는다. 물론, 부자가 아니라면 그렇다는 말이다. 부자라면 마지막 1원까지 챙길 이유가 있을까?

파일 시스템 설계가 정말 재미있는 이유는 자유롭다는 것이다. 앞으로 다루게 될 파일 시스템들은 특정 부분을 자유롭게 최적화한다. 심도있게 다루지 못한 파일 시스템 정책에 관련된 내용들이 많이 있다. 예를 들어, 새로운 파일이 생성되었을 때 디스크 어디에 이 파일을 배치해야 할까? 이들을 다음 장들에서 다루게 될 것이다.[7]

7) 신비로운 음악을 틀어 본다면 파일 시스템의 주제에 대해서 더욱 흥미를 갖게 될 것이다.

참고 문헌

[Agr+07] **"A Five-Year Study of File-System Metadata"**
Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch
FAST '07, pages 31,45, February 2007, San Jose, CA
최신의 훌륭한 분석으로 파일 시스템들이 실제 어떻게 사용되는지를 나타낸다. 참고하고 있는 참고 문헌을 보고 1980년대 초반의 파일 시스템을 분석한 논문들의 흔적을 찾아 보자.

[BM07] **"ZFS: The LastWord in File Systems"**
Jeff Bonwick and Bill Moore
URL: `http://opensolaris.org/os/community/zfs/docs/zfs%20last.pdf`
가장 최근의 중요한 파일 시스템 중의 하나로 기능이 풍부하며 경탄할만한 것으로 가득하다. 이 내용을 주제로 하는 장이 필요하며 곧 그렇게 될 것이다.

[Bro02] **"The FAT File System"**
Andries Brouwer
URL: `http://www.win.tue.nl/.aeb/linux/fs/fat/fat.html`
FAT에 대해 깔끔하게 설명이 잘 되어 있다. 파일 시스템이지 베이컨에 있는 그런 것이 아니다. 솔직하게 베이컨의 FAT(지방)이 아마도 훨씬 맛있다는 것을 인정해야 할 것이다.

[Cus94] **"Inside the Windows NT File System"**
Helen Custer
Microsoft Press, 1994
NTFS에 대한 짧은 책이다. 어딘가에 좀 더 상세한 기법을 다룬 책이 있을 것이다.

[How+88] **"Scale and Performance in a Distributed File System"**
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J.West.
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
분산 파일 시스템의 고전이다. 앞으로 더 자세히 다루게 될 것이다. 걱정하지 마라.

[Poi07] **"The Second Extended File System: Internal Layout"**
Dave Poirier
URL: `http://www.nongnu.org/ext2-doc/ext2.html`
Berkeley Fast File System 또는 FFS 기반의 간단한 Linux 파일 시스템인 ext2에 대한 설명이다. 다음 장에서 이에 대해 더 읽어 볼 수 있다.

[RT74] **"The Unix Time-Sharing System"**
M. Ritchie and K. Thompson
CACM, Volume 17:7, pages 365-375, 1974
Unix에 관한 최초의 논문이다. 읽어서 현대 운영체제의 대부분을 이해하자.

[Sil00] **"UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD"**
Chuck Silvers
FREENIX, 2000
파일 시스템의 버퍼 캐시와 가상 메모리의 페이지 캐시를 통합한 NetBSD에 대한 좋은 논문이다. 많은 다른 시스템들이 이와 비슷한 일을 하고 있다.

[Swe+96] **"Scalability in the XFS File System"**

Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck
USENIX '96, January 1996, San Diego, CA
핵심 주제인 디렉터리 내에 백만 개의 파일을 포함하는 것과 같은 연산의 확장성을 만들기 위한 첫 시도이다. 개념을 극한으로 까지 밀어붙이는 아주 좋은 예이다. 이 파일 시스템의 핵심 아이디어는 모든 것이 트리라는 것이다. 이 파일 시스템에 대한 장도 필요하다.

숙제

`vsfs.py`라는 툴을 사용하여 여러 동작들을 처리할 때 파일 시스템의 상태가 어떻게 변하는지 살펴보자. 파일 시스템은 루트 디렉터리 외에는 비어 있는 상태로 시작한다. 시뮬레이션이 동작하면서 여러 작업들이 수행될 것이며 그에 따라 파일 시스템의 디스크 상의 정보의 상태가 천천히 변경될 것이다. README를 참고하자.

문제

1. 다른 랜덤 시드를 사용하여(17, 18, 19, 20으로) 시뮬레이터를 실행해 보자. 그리고 각 상태가 변할 때마다 어떤 연산이 수행되었는지 알아보자.
2. 이번에는 랜덤 시드를 (21, 22, 23, 24) 바꿔서 위와 똑같이 해 보자. 대신 `-r` 플래그를 사용하여 실행해 보자. 어떤 연산이 수행되었는지 출력되면 상태가 어떻게 바뀔지 예상해 보자. 아이노드와 데이터 블럭 할당 알고리즘이 어떤 블럭들을 할당하기를 선호할까라는 측면에서 내릴 수 있는 결론은 무엇인가?
3. 이번에는 파일 시스템의 데이터 블럭의 수를 줄여 보자. 아주 작은 수(2라고 해 보자)로 정하고 백여 개의 요청을 시뮬레이터에서 처리하여 보자. 이렇게 상당히 제약적인 배치에서 최종적으로 파일 시스템에 남는 파일의 종류는 무엇인가? 어떤 종류의 작업이 실패하게 될까?
4. 똑같이 하되 아이노드에 대해서 해 보자. 매우 적은 수의 아이노드를 사용할 때 어떤 종류의 작업들이 성공할 수 있을까? 어떤 종류가 대체적으로 실패 할까? 파일 시스템의 최종 상태는 어떻게 될까?