40

하드 디스크 드라이브

이전 장에서 I/O 장치에 대한 전반적인 개념을 다루었고 운영체제가 I/O 장치라는 괴물과 어떻게 상호작용하는지 살펴보았다. 이 장에서 특정 장치에 대해서 좀 더 자세히 살펴보기로 한다. 바로 **하드 디스크 드라이브**이다. 이런 드라이브들이 수 세기 동안 컴퓨터 시스템의 영구적인 데이터 저장소였으며 (곧 살펴볼) 파일 시스템 기술은 거의 대부분 하드 디스크 드라이브의 동작에 기반을 두고 개발되었다. 그러므로 디스크를 관리하는 파일 시스템 소프트웨어를 구현하기 전에 디스크의 상세한 동작을 이해하는 것이 중요하다. 여기서 논의하는 대부분의 내용은 Ruemmler와 Wilkes [RW94] 그리고 Anderson, Dykes와 Riedel [ADR03]이 쓴 탁월한 논문에 나와 있다.

핵심 질문: 디스크에 있는 데이터를 어떻게 저장하고 접근하는가
현대 하드 디스크 드라이브는 어떻게 데이터를 저장하는가? 인터페이스는 무엇인가? 데이터는 실제로 어떻게 배치되고 접근되는가? 디스크 스케줄링은 어떻게 성능을 개선시킬 수 있는가?

40.1 인터페이스

현대 디스크 드라이브의 인터페이스를 이해하는 것부터 시작해 보자. 모든 현대 드라이브의 기본적인 인터페이스는 단순하다. 드라이브는 읽고 쓸 수 있는 매우 많은 수의 섹터(512 byte 블럭)들로 이루어져 있다. 디스크 위의 n 개의 섹터들은 0부터 n-1까지의 이름이 붙어 있다. 그렇기 때문에 디스크를 섹터들의 배열로 볼 수 있으며 0부터 n-1이 드라이브의 **주소 공간**이 된다.

멀티 섹터 작업도 가능하다. 사실 많은 파일 시스템들이 한 번에 4 KB(또는 그 이상)를 읽거나 쓴다. 하지만 드라이브 제조사는 하나의 512 byte 쓰기만 **원자적**(즉, 온전히 모두 완료되거나, 온전히 모두 실패함)이라고 보장한다. 그러므로 갑작스럽게 전력 손실이 발생한다면 대량의 쓰기 중에 일부만 완료될 수 있다(때로는 이런 현상을 **찢긴 쓰기**(**torn write**)라고 부른다).

디스크 드라이브 사용자들은 몇 가지 가정을 하지만 그것들이 인터페이스에 직접적으로 명시되어 있지는 않다. Schlosser와 Ganger는 이것을 디스크 드라이브의 "계약 불문율"이라고 부른다 [SG04]. 구체적으로 말하자면, 드라이브의 주소 공간에서 가깝게 배치되어 있는 두 개의 블럭을 접근하는 것은 멀리 떨어져 있는 두 개의 블럭을 접근하는 것보다 빠르다고 가정한다. 또 다른 가정은 연속적인 청크의 블럭을 접근하는 것(순차 읽기 또는 쓰기)이 가장 빠르며 어떤 랜덤 접근 패턴보다 매우 빠르다는 것이다.

40.2 기본 구조

현대 디스크의 주요 요소들을 이해해 보자. 먼저 **플래터(platter)**라는 것을 살펴보자. 원형의 딱딱한 표면을 갖고 있는 플래터에 자기적 성질을 변형하여 데이터를 지속시킨다. 디스크는 하나 또는 그 이상의 플래터를 갖고 있으며 각각은 2개의 **표면(surface)**을 갖고 있다. 이런 플래터는 대체적으로 단단한 물질(알루미늄과 같은)로 만들어지며 드라이브의 전원이 나가더라도 비트를 드라이브에 영구적으로 저장하기 위해 얇은 자성층이 입혀져 있다.

플래터들은 **회전축(spindle)**이라는 것으로 고정되어 있는데, 이 축은 모터와 연결이 되어 있어서 (드라이브에 전원이 인가된 경우) 플래터를 일정한 (고정된) 속도로 회전시킨다. 회전의 속도는 **분당 회전 수(rotation per minute, RPM)**로 측정되며 일반적인 값은 7,200 RPM에서 15,000 RPM 사이에 있다. 대체로 우리가 관심을 갖는 것은 플래터가 한 바퀴 회전할 때 걸리는 시간이라는 것을 유의하자. 예를 들면 10,000 RPM의 속도로 드라이브가 회전할 때 한 바퀴 회전하는 데 걸리는 시간은 6 msec(ms)이다.

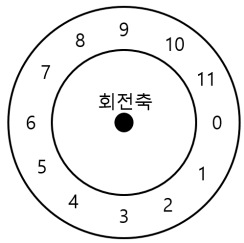
각 표면에 동심원을 따라 배치되어 있는 섹터들 위에 데이터는 부호화된다. 이때 동심원 하나를 **트랙(track)**이라고 한다. 표면에는 수많은 트랙들이 서로 촘촘하게 붙어 있다. 수백 개의 트랙들이 모여야 사람의 머리카락 두께 정도가 된다.

표면 위를 읽거나 쓸 때에는 디스크의 자기적 패턴을 감지하거나(읽거나) 변형을 유도하는(쓰는) 기계적 장치가 필요하다. 읽기와 쓰기 동작은 **디스크 헤드(disk head)**라는 것을 통해 할 수 있으며 각 표면마다 그런 헤드가 하나씩 존재한다. 디스크 헤드는 **디스크 암(disk arm)**에 연결이 되어 있으며 이것을 통해서 헤드가 원하는 트랙 위에 위치하도록 이동시킬 수 있다.

40.3 간단한 디스크 드라이브

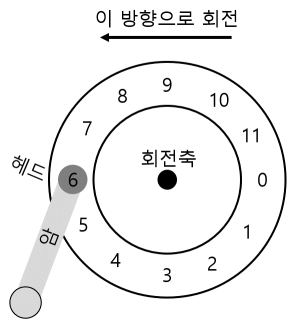
디스크가 어떻게 동작하는지 이해하기 위해 한 트랙씩 모형을 만들어 보자. 그림 40.1에서 나타난 것과 같이 하나의 트랙으로만 이루어진 간단한 디스크를 생각해 보자.

이 트랙에는 12개의 섹터가 있고 각 섹터는 512 byte의 크기를 갖고 있으며(앞서 이야기 했듯이 이것이 섹터의 보편적인 크기이다), 주소 영역은 0부터 11까지로 이루어져 있다. 모터에 연결된 회전축을 중심으로 플래터가 회전한다. 물론, 트랙 자체로는



〈그림 40.1〉 **트랙이 하나만 존재하는 디스크**

흥미롭지 않기 때문에 섹터에 무엇인가를 읽거나 쓰고 싶을 것이다. 그렇기 때문에 그림 40.2와 같이 디스크 암에 붙어 있는 디스크 헤드가 필요하다.



〈그림 40.2〉 **트랙 하나와 헤드**

그림에서 보는 것과 같이 디스크 헤드는 디스크 암의 끝에 붙어 있으며 섹터 6번 위에 위치해 있다. 그리고 표면은 시계 반대 방향으로 회전한다.

단일 트랙 지연 시간: 회전 지연

트랙이 하나 뿐인 이 간단한 디스크에서 요청이 어떻게 처리되는지 이해하기 위해서 블럭 0번을 읽는다고 가정해 보자. 디스크가 이 요청을 어떻게 처리해야 할까?

우리가 사용하는 간단한 디스크의 경우 그렇게 많은 일을 할 필요가 없다. 디스크 헤드 아래에 원하는 섹터가 위치하기를 기다리면 된다. 이러한 기다림은 현대 드라이브에서도 흔하게 발생하며 I/O 서비스 시간에서 중요한 요소이기 때문에 **회전형 지연**(**rotational delay**, 이상하게 들릴 수도 있겠지만 때로는 **회전 지연**(**rotation delay**)이라고 불림)이라는 특별한 이름을 갖고 있기도 하다. 예제에서 만약 한 바퀴를 다 회전하는 데 걸리는 회전 지연이 R이라고 하면 디스크는 (6에서 시작한 경우) 읽거나 쓰려는 디스크 헤드가 0에 위치하기 위해서는 $\frac{R}{2}$이 필요하다. 트랙이 하나 있을 때의 최악의 경우는 헤드가 섹터 5번에 있을 때가 될 것이다. 거의 한 바퀴를 다 돌아야 요청을 처리할 수 있게 된다.

멀티 트랙: 탐색 시간

지금까지의 디스크는 트랙이 하나만 존재하는 비현실적인 경우를 살펴보았다. 현대 디스크들은 당연하겠지만 수백만 개의 트랙을 갖고 있다. 이제 아주 조금 더 현실적인 디스크 표면을 생각해 보자. 그림 40.3의 좌측에 트랙이 세 개가 있는 그림을 살펴보자.

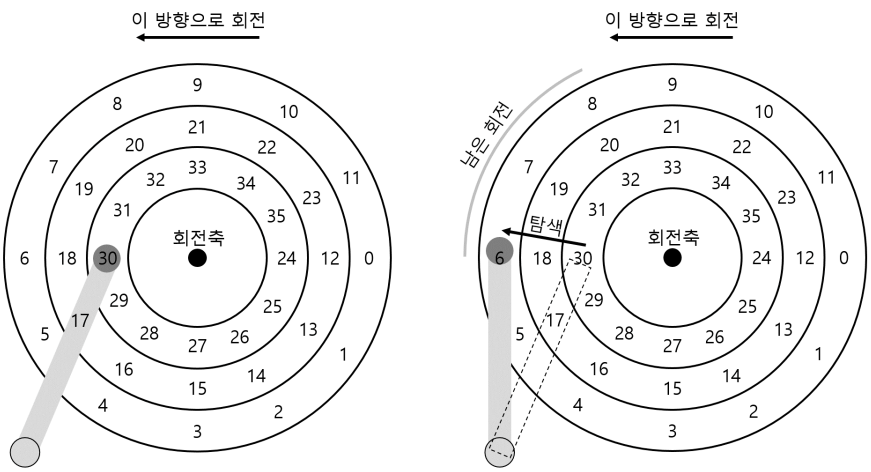
이 그림에서 헤드는 가장 안쪽의 트랙(섹터 24번부터 35번까지)에 위치하고 있다. 그 다음 트랙은 다음의 섹터 집합(12번부터 23번까지)을 갖고 있으며 가장 바깥쪽의 트랙에는 선두의 섹터들(0번부터 11번까지)이 존재한다.

드라이브가 지정된 섹터들을 접근하는 방식을 이해하기 위한 예로 섹터 11번을 읽는 경우처럼 멀리 떨어져 있는 섹터에 대한 요청을 받은 때를 살펴보겠다. 이 읽기 요청을 처리하기 위해서 드라이브는 디스크 암을 먼저 올바른 트랙 위에 위치시킨다(이 경우에는 가장 바깥쪽 트랙이다). 이 과정을 **탐색(seek)**이라고 한다. 회전과 더불어서 탐색은 가장 비싼 디스크 동작 중 하나이다.

탐색은 여러 단계로 되어 있다는 것에 유의해야 한다. 첫 번째는 *가속* 단계로 디스크의 암이 움직이기 시작한다. 디스크 암이 최고 속도로 움직이는 *활주* 단계를 지나고, 디스크 암의 속도가 줄어드는 *감속* 단계 이후에 *안정화* 단계에서 정확한 트랙 위에 헤드가 조심스럽게 위치하게 된다. 드라이브가 정확한 트랙 위에 위치했는지 확실하게 해야 하기 때문에(정확하지 않고 거의 근접한 경우를 생각해 보라!) **안정화 시간(settling time)**은 매우 중요하며 0.5에서 2 msec 정도로 오래 걸린다.

탐색 이후에 디스크 암은 올바른 트랙 위에 헤드를 위치시킨다. 탐색 동작이 그림 40.3에 묘사되어 있다.

그림에 나타난 것과 같이 탐색 과정에서 암이 원하는 트랙 위로 이동을 하는 동안에 당연하게 플래터 역시 회전하였다. 이 경우 3개의 섹터만큼 이동하였다. 섹터 9번이 디스크 헤드 아래로 막 지나가고 있기 때문에 약간의 회전 지연 후에 전송을 완료할 수 있다.

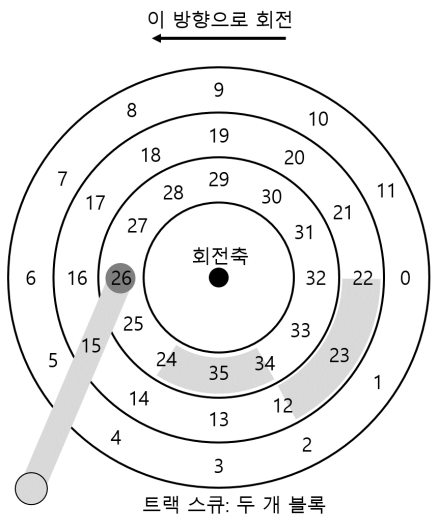


〈그림 40.3〉 **세 개의 트랙과 헤드 (우측: 탐색을 포함)**

섹터 11번이 디스크 헤드를 지나가게 되면 I/O의 마지막 단계인 **전송**이 이루어져 표면 위의 데이터를 읽거나 쓰게 된다. 이제 I/O 시간은 탐색과 회전 지연 동안 기다린 후 전송한다는 전체 윤곽이 그려졌다.

그 외의 세부 사항

아주 자세히는 아니지만, 하드 드라이브 동작에 대한 몇 가지 흥미로운 내용을 살펴보자. 많은 드라이브는 **트랙 비틀림**(**track skew**)이라 불리는 기술을 채용하여 트랙의 경계를 지나서 순차적으로 존재하는 섹터들을 올바르게 읽을 수 있게 한다. 우리가 사용하는 디스크에서 트랙 비틀림은 그림 40.4와 같이 나타낼 수 있다.



〈그림 40.4〉 **세 개의 트랙: 트랙 스큐 2**

한 트랙에서 다른 트랙으로 전환하는 경우에, 바로 인접한 트랙으로 전환되는 경우에도, 디스크의 헤드를 다시 위치시키기 위한 시간이 필요하다. 이와 같은 비틀림이 없다면 헤드가 다음 트랙으로 넘어 갔을 때 다음에 읽어야 하는 블럭이 이미 헤드를 지나쳤을 수도 있기 때문에 다음 블럭을 접근하기 위해 거의 한 바퀴에 해당하는 회전 지연을 감수해야 한다.

또 다른 현실 상황은 바깥 측에 공간이 더 많다는 구조적인 이유 때문에 바깥 측 트랙들에는 안쪽 트랙들보다 더 많은 섹터들을 갖고 있다. 이러한 트랙들을 흔히 **멀티 구역**(**multi-zoned**) 디스크 드라이브라고 부른다. 디스크들은 여러 구역으로 나뉘어 있으며 한 구역은 표면 위에 연속적으로 존재하는 트랙들의 집합이다. 각 구역 내의 트랙은 같은 수의 섹터들을 포함하고 있으며 바깥 측 구역의 트랙에는 안쪽 구역의 트랙보다 많은 수의 섹터들을 갖고 있다.

마지막으로 현대 디스크 드라이브의 가장 중요한 부분은 **캐시**(**cache**)로서, 역사적인 이유로 때로는 **트랙 버퍼**(**track buffer**)라고도 부른다. 이 캐시는 일반적으로 8 또는

16 MB 정도의 작은 크기의 메모리로 드라이브가 디스크에서 읽거나 쓴 데이터를 보관하는 데 사용한다. 예를 들어, 디스크에서 하나의 섹터를 읽을 때 드라이브가 그 트랙 위의 모든 섹터를 다 읽어서 캐시에 저장할 수도 있다. 이렇게 하면 같은 트랙의 섹터에 대한 이후의 요청에 빠르게 응답할 수 있게 된다.

쓰기의 경우 드라이브는 선택할 수 있다. 메모리에 데이터가 기록된 시점에 쓰기가 완료되었다고 할지, 디스크에 실제로 기록되었을 때 완료가 되었다고 할지를 정할 수 있다. 전자는 **write-back** 캐싱(또는 **즉시 보고**(**immediate reporting**)라고 함)이라고 부르고, 후자는 **write-through**라고 부른다. 때로는 Write-back 캐싱을 사용할 경우 드라이브가 "더 빠른" 것처럼 보이지만 위험할 수 있다. 만약 파일 시스템이나 응용 프로그램이 정확함을 위해 특정 순서로 디스크에 기록해야 한다고 할 때 write-back을 사용하면 문제가 될 수 있다(파일 시스템 저널링을 설명하는 장을 읽어보라).

여담: 차원 분석

화학 시간에 단위를 잘 정리해 놓고 상쇄시키다 보면 거의 대부분의 문제들이 풀리기도 하고 그 자체가 답이 되는 경우를 기억하는가? 이 기법을 멋있게 표현하여 **차원 분석**(**dimensional analysis**)이라고 한다. 컴퓨터 시스템 분석에서도 이 방법은 유용하다.

차원 분석이 어떻게 동작하는지 예제를 통해 살펴보고 왜 유용한지 알아보자. 디스크가 한 번 회전하는 데 얼마나 오랜 시간이 걸리는지를 msec 단위로 계산한다고 해보자. 불행하게도 디스크의 **RPM** 또는 **분당 회전수**만을 알고 있다. 10 K RPM 디스크(분당 10,000번 회전)를 예로 들어보자. msec당 회전수를 알려면 어떻게 차원 분석 식을 작성해야 할까?

먼저 식의 좌변에 원하는 단위를 기록하는 것으로 시작한다. 우리는 회전 한 번에 소요되는 시간을 msec 단위로 알고 싶어 하기 때문에 우리가 원하는 바를 그대로 $\frac{Time(ms)}{1\ Rotation}$ 라고 쓴다. 그리고 우리가 아는 모든 것을 쓰되 상쇄할 수 있는 것들을 찾아 상쇄한다. 먼저 좌변에서 회전을 분모로 놓은 것처럼 우변에도 $\frac{1\ minute}{10{,}000\ Rotation}$ 이라고 쓴다. 그리고 분을 초로 바꾸어 $\frac{60\ seconds}{1\ minute}$ 라고 쓰고 다시 msec로 바꾸어 $\frac{1000\ ms}{1\ second}$ 라고 쓴다. 최종 결과는 다음과 같다(단위들이 자연스레 상쇄된다).

$$\frac{Time\ (ms)}{1\ Rot.} = \frac{1\ \cancel{min}}{10{,}000\ Rot.} \cdot \frac{60\ \cancel{sec}}{1\ \cancel{min}} \cdot \frac{1000\ ms}{1\ \cancel{sec}} = \frac{60{,}000\ ms}{10{,}000\ Rot.} = \frac{6\ ms}{Rotation}$$

이 예제에서 보는 것과 같이 차원 분석은 직관적인 것처럼 보이는 과정을 간단한 반복 과정으로 바꾼다. RPM 계산 외에도 I/O를 분석하는 데 자주 유용하게 활용된다. 예를 들면 디스크의 전송 속도가 100 MB/s와 같이 주어졌을 때 512 KB 블럭을 전송하는 데 몇 msec가 소요되는가와 같은 질문을 쉽게 만날 수 있다. 차원 분석을 사용하면 간단하다.

$$\frac{Time\ (ms)}{1\ Request} = \frac{512\ \cancel{KB}}{1\ Request} \cdot \frac{1\ \cancel{MB}}{1024\ \cancel{KB}} \cdot \frac{1\ \cancel{sec}}{100\ \cancel{MB}} \cdot \frac{1000\ ms}{1\ \cancel{sec}} = \frac{5\ ms}{Request}$$

40.4 I/O 시간 계산

이제 추상화된 디스크 모델을 만들었으니 간단한 분석을 통해 디스크의 성능을 구할 수 있다. 세 개의 항으로 이루어진 다음의 식을 통해 I/O 시간을 나타낼 수가 있다.

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \tag{40.1}$$

드라이브 간의 비교를 쉽게 하기 위해 주로 사용되는 I/O의 속도(rate, $R_{I/O}$)는 시간을 사용하여 다음의 식과 같이 간단하게 나타낼 수가 있다.

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}} \tag{40.2}$$

I/O 시간에 대한 이해를 돕기 위해 계산을 해 보자. 두 개의 워크로드가 있다고 가정하자. 하나는 **랜덤** 워크로드로 디스크에 4 KB의 작은 읽기 요청을 발생시킨다. 랜덤 워크로드는 데이터베이스 관리 시스템과 같은 많은 중요 응용 프로그램에서 흔하게 사용된다. 두 번째는 **순차** 워크로드로서 헤드의 이동 없이 디스크에 연속되어 있는 여러 개의 섹터를 단순히 읽는 것이다. 순차 접근 패턴 역시 흔하기 때문에 마찬가지로 중요한 워크로드이다.

랜덤과 순차 워크로드의 성능 차이를 이해하기 위해 먼저 디스크 드라이브에 대한 몇 가지 가정을 해야 한다. Seagate 사의 디스크들을 예로 사용해 보자. 하나는 Cheetah 15K.5 [S09b]로 고성능 SCSI 드라이브이고, 다른 하나는 용량을 위해 주로 쓰이는 Barracuda [S09a]이다. 상세 명세는 그림 40.5에 나타나 있다.

	Cheetah 15K.5	**Barracuda**
용량	300 GB	1 TB
RPM	15,000	7,200
평균 탐색 시간	4 msec	9 msec
최대 전송량	125 MB/s	105 MB/s
플래터	4	4
캐시	16 MB	16/32 MB
연결 방식	SCSI	SATA

〈그림 40.5〉 **디스크 드라이브 명세 : SCSI 대 SATA**

보는 것과 같이 드라이브들은 상당히 다른 특성을 갖고 있고 디스크 드라이브 시장의 두 가지 중요한 요소를 잘 요약하고 있다. 하나는 "고성능" 드라이브 시장으로 가능한 빠르게 회전하도록 설계되어서 낮은 탐색 시간과 빠른 데이터 전송 속도를 갖고 있다. 두 번째는 "용량" 위주의 시장으로 바이트당 가격이 가장 중요한 측면이라서 드라이브 속도는 낮지만 주어진 공간에 가능한 많은 비트를 저장한다.

팁: 디스크를 순차적으로 사용하자

데이터를 디스크로 전송하거나 전송받을 때에는 가능하면 순차적인 방식으로 해야 한다. 순차적으로 전송하는 것이 불가능하면 최소한 큰 청크 단위로 데이터를 전송할 수 있는 방법을 생각해야 한다. 청크의 크기가 클수록 더 좋다. 만약 I/O가 작은 임의의 크기 단위로 처리된다면 I/O의 성능은 극적으로 나빠질 것이다. 또한, 사용자도 고통받게 될 것이다. 당신도 생각 없이 랜덤 I/O를 발생시켰기 때문에 고생하고 있다는 것을 알게 되면 고통받게 될 것이다.

그림에 나타난 드라이브의 값들을 사용하여 앞에서 정리한 두 개의 워크로드가 얼마나 잘 동작하는지를 계산해 볼 수 있다. 먼저 랜덤 워크로드의 경우를 살펴보자. 랜덤한 디스크의 위치에서 4 KB씩 읽기가 발생한다고 했을 때 Cheetah에서 각 읽기가 얼마나 오래 걸릴지를 다음의 식처럼 계산할 수 있다.

$$T_{seek} = 4 \text{ ms}, \ T_{rotation} = 2 \text{ ms}, \ T_{transfer} = 30 \ us \tag{40.3}$$

평균 탐색 시간(4 msec)은 제조사가 명시하고 있는 평균 시간을 사용하였다. 전체 탐색(표면의 한쪽 끝에서 반대편 끝까지 이동)은 대체로 두 배에서 세 배 가량 더 긴 시간이 필요하다는 것에 유의하라. 평균 회전 지연은 RPM에서 직접적으로 계산해낼 수 있다. 15,000 RPM은 250 RPS(초당 회전수)로 나타낼 수 있으므로 한 번의 회전은 4 msec가 걸린다. 평균적으로 디스크는 반 바퀴를 회전을 할테니 평균 회전 지연 시간은 2 msec가 된다. 마지막으로 전송 시간은 전송된 데이터 크기를 최대 전송 속도로 나눈 값이다. 여기서는 없는 것이나 다름없는 작은 값이다(30 μs, 1,000 μs가 있어야 1 ms가 된다!).

위의 식에 따라 Cheetah의 $T_{I/O}$는 약 6 msec가 된다. I/O 속도를 계산하기 위해서 전송 데이터의 크기를 평균 시간으로 나눈다. 랜덤 워크로드를 처리하는 Cheetah의 $R_{I/O}$는 0.66 MB/s가 된다. Barracuda도 같은 방식으로 계산하면 $T_{I/O}$는 약 13.2 msec로 두 배 이상이나 느리고 전송 속도는 약 0.31 MB/s가 된다.

이제 순차 워크로드를 살펴보자. 아주 긴 시간의 전송 전에 한 번의 탐색과 회전이 있다고 가정해 보자. 논의하기 쉽도록 전송할 데이터의 크기는 100 MB라고 하자. 그러면 Barracuda와 Cheetah의 $T_{I/O}$는 각각 800 ms와 950 ms가 된다. I/O의 속도는 드라이브의 최고 전송 속도인 125 MB/s와 105 MB/s와 거의 비슷하게 된다. 그림 40.6에 이 수치들이 요약되어 있다.

	Cheetah 15K.5	**Barracuda**
$R_{I/O}$ 랜덤	0.66 MB/s	0.31 MB/s
$R_{I/O}$ 순차	125 MB/s	105 MB/s

〈그림 40.6〉 **디스크 드라이브 성능: SCSI 대 SATA**

이 그림은 몇 가지 중요한 사실을 알려준다. 첫 번째 가장 중요한 사실은 랜덤 워크로드와 순차 워크로드의 드라이브 간 성능 차이가 크다는 것이다. Cheetah의 경우에는 거의 200배 이상 차이 나고, Barracuda의 경우 300배 이상 차이가 난다. 이렇게 컴퓨터 역사 상 가장 분명한 디자인 팁에 이르게 된다.

두 번째는 좀 더 미묘한데 "성능" 위주의 드라이브와 저사양의 "용량" 위주의 드라이브 간의 성능 차이가 상당히 크다는 것이다. 이러한 이유로(또한 다른 이유로) 전자를 위해서는 비싼 돈을 들이는 데 주저하지 않으면서도 후자를 구하기 위해서는 가능한 싸게 사려고 한다.

40.5 디스크 스케줄링

I/O의 비용이 크기 때문에 역사적으로 운영체제는 디스크에게 요청되는 I/O의 순서를 결정하는 데에 중요 역할을 담당했다. 구체적으로 이야기 하자면 I/O 요청이 주어졌을 때 **디스크 스케줄러**는 요청을 조사하여 다음에 어떤 I/O를 처리할지 결정하였다 [SCO90; JJ91].

각 작업의 길이가 얼마나 될지 알 수 없는 작업 스케줄링과 다르게 디스크 스케줄링의 경우, 디스크 요청 작업이 얼마나 길지를 꽤 정확히 예측할 수 있다. 요청의 탐색과 회전 지연의 정도를 예측하면 각 요청이 얼마나 오래 걸릴지 디스크 스케줄러가 알 수 있기 때문에 (greedy 방식으로) 처리할 수 있는 가장 빠른 요청을 선택할 수 있다. 이와 같이 디스크 스케줄러는 **SJF**(**shortest job first**, **짧은 작업 우선**)의 원칙을 따르려고 노력한다.

SSTF: 최단 탐색 시간 우선

초기의 디스크 스케줄링 접근 방법은 **최단 탐색 시간 우선**(**Shortest-seek-time-first**, **SSTF**)을 사용하였다(또는 **최단 탐색 우선**(**shortest-seek-first**, **SSF**)이라고도 불림). SSTF는 트랙을 기준으로 I/O 요청 큐를 정렬하여 가장 가까운 트랙의 요청이 우선 처리되도록 한다. 예를 들어 현재 헤드의 위치가 안쪽 트랙에 위치해 있다고 하자. 이때에 가운데 있는 트랙의 섹터 21번과 바깥 측의 섹터 2번에 대한 요청을 받으면, 21번 요청을 먼저 처리하고 완료되기를 기다렸다가 요청 2번을 처리한다(그림 40.7).

이 예제의 경우에 SSTF는 중간의 트랙으로 탐색했다가 바깥 측 트랙을 탐색하기 때문에 잘 동작한다. 하지만 다음과 같은 이유로 SSTF가 만병통치약은 아니다. 첫째로 드라이브의 구조는 호스트 운영체제에게 공개되어 있지 않으며 운영체제는 그저 블럭들의 배열로만 인식한다. 다행스러운 것은 이 문제는 제법 쉽게 해결될 수 있다. 운영체제는 SSTF를 사용하는 대신 **가장 가까운 블럭 우선**(**Nearest-block-first**, **NBF**) 방식을 사용하면 된다. 이 방식은 가장 가까운 블럭 주소에 접근하는 요청을 다음에 처리하도록 스케줄한다.

두 번째는 더 중요한 **기아 현상**(**starvation**)에 대한 문제이다. 위의 예제에서 현재 헤드가 위치하고 있는 안쪽 트랙에만 지속적으로 요청이 발생되는 상황을 생각해 보자.

여담 : "평균" 탐색 시간 계산하기

많은 책들과 논문에서 평균 디스크 탐색 시간을 전체 탐색 시간의 대략 삼분의 일 정도로 말하는 것을 볼 수 있을 것이다. 이 수치는 어디에서 나오는 것일까?

평균 탐색 시간이 아니라 평균 탐색 거리를 기준으로 간단한 계산을 해보면 확인할 수 있다. 트랙이 0부터 N개가 있는 디스크를 가정해 보자. 두 개의 트랙 x와 y간의 탐색 거리는 $|x-y|$로 x와 y의 차의 절대치로 계산할 수 있다.

평균 탐색 거리를 계산하기 위해서 할 일은 가능한 모든 탐색 거리들을 다 더하는 것이다.

$$\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y| \tag{40.4}$$

그리고 모든 가능한 서로 다른 탐색들의 수인 N^2로 나누는 것이다. 다 더하기 위해서는 적분식을 사용하면 된다.

$$\int_{x=0}^{N}\int_{y=0}^{N}|x-y|\,dy\,dx \tag{40.5}$$

안쪽의 적분을 위해서 절대치를 다음과 같이 두 구간으로 나눈다.

$$\int_{y=0}^{x}(x-y)\;dy+\int_{y=x}^{N}(y-x)\;dy \tag{40.6}$$

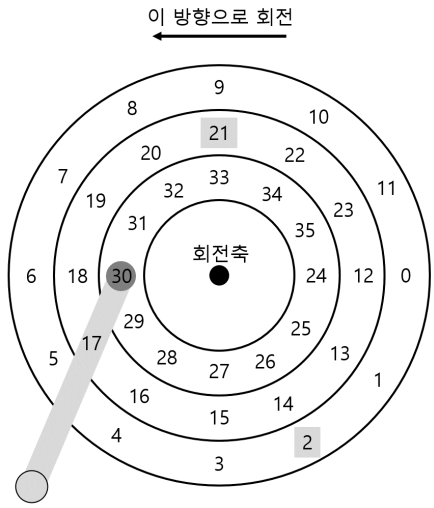
이 식을 계산하면 $\left(xy-\frac{1}{2}y^2\right)|_0^x+\left(\frac{1}{2}y^2-xy\right)|_x^N$을 얻을 수 있으며 간단하게 만들면 $\left(x^2-Nx+\frac{1}{2}N^2\right)$을 얻는다. 이제 바깥쪽의 적분을 계산해야 한다.

$$\int_{x=0}^{N}\left(x^2-Nx+\frac{1}{2}N^2\right)\;dx \tag{40.7}$$

계산하면 다음과 같은 결과를 얻는다.

$$\left(\frac{1}{3}x^3-\frac{N}{2}x^2+\frac{N^2}{2}x\right)\Bigg|_0^N=\frac{N^3}{3} \tag{40.8}$$

전체 탐색의 횟수(N^2)로 위의 식을 나눠야 평균 탐색 거리를 얻을 수 있다는 것을 기억하라. 따라서 계산하면 $\left(\frac{N^3}{3}\right)/\left(N^2\right)=\frac{1}{3}N$을 얻는다. 그러므로 디스크의 평균 탐색 거리는 모든 가능한 탐색을 따져보았을 때 전체 거리의 삼분의 일이 된다. 이제 평균 탐색이 어떻게 전체 탐색의 삼분의 일이 되는지 알았을 것이다.



〈그림 40.7〉 **SSTF: 21과 2 요청의 스케줄링**

순수한 SSTF 방식에서는 다른 트랙에 있는 요청들은 완전히 무시될 것이다. 그러므로 다음과 같은 질문을 할 수 있다.

핵심 질문: 디스크 요청의 기아 현상을 어떻게 처리할까
SSTF와 유사한 스케줄링을 구현하면서 어떻게 기아 현상을 피할 수 있을까?

엘리베이터(SCAN 또는 C-SCAN이라고도 함)

이 질문에 대한 해법은 꽤 오래전에 개발되었으며(그 중 하나는 [CKR72]를 참고하자), 비교적 자명하다. 최초에는 **SCAN**이라고 불렸던 이 알고리즘은 트랙의 순서에 따라 디스크를 앞뒤로 가로지르며 요청을 서비스한다. 디스크를 한 번 가로지르는 것을(밖에서 안으로 또 안에서 밖으로) 스위프(`sweep`)라고 부르자. 따라서 어떤 요청이 이번 스위프에 이미 지나간 트랙에 대해 들어온다면 바로 처리되지 않고 다음 번 스위프에(반대 방향) 처리되도록 큐에서 대기한다.

SCAN은 몇 가지 변종이 있는데 모두가 비슷하게 동작한다. 예를 들면 Coffman 등은 스위프하는 동안에는 큐를 동결시키는 **F-SCAN**이라는 방법을 소개하였다 [CKR72]. 디스크를 스위프 하는 동안에 새로운 요청이 도착하면 다음 번 서비스 될 큐에 삽입한다. 이와 같이 현재 요청과 가까이 있지만 늦게 도착한 요청들의 처리를 지연시켜 멀리 떨어져 있는 요청에 대한 기아 현상을 없앴다.

C-SCAN은 또 다른 일반적인 변종으로 **Circular SCAN**의 약자이다. 디스크를 한 방향으로 스위프하는 대신 이 알고리즘은 밖에서 안으로 그리고 다시 안에서 밖으로 스위프한다.

이제는 확실하게 이해할 수 있는 이유로 이 SCAN 알고리즘(그리고 변종들)은 현재 위치와 가까운 층 위주로 이동하는 것이 아니라 위로 또는 아래로 이동하는 엘레베이터와 같다하여 **엘리베이터** 알고리즘이라고 불린다. 당신이 10층에서 1층으로 내려가고 있는 중에 3층에서 탄 사람이 4층을 눌렀고 4층이 1층보다 가깝다는 이유로 위로 올라갔다면 얼마나 짜증이 나겠는가! 이와 같이 엘리베이터 알고리즘을 실제 생활에 적용하면 엘리베이터 안에서 일어날 수 있는 싸움을 예방할 수 있다. 디스크에 적용하면 기아 현상을 예방할 수 있다.

불행하게도 SCAN과 그와 유사한 기법들은 최고의 스케줄링 기술을 의미하지 않는다. 그 이유는 SCAN(또는 SSTF 까지도) 방식은 SJF의 원칙을 지키기 위해 최선을 다하지 않기 때문이다. 구체적으로는 이 기법들은 회전을 무시한다. 그러므로 다음과 같은 질문이 생긴다.

핵심 질문: 디스크 회전 비용을 고려하려면 어떻게 해야 할까

어떻게 하면 탐색과 회전을 모두 고려하여 SJF를 가장 근접하게 모방하는 알고리즘을 만들 수 있을까?

SPTF: 최단 위치 잡기 우선

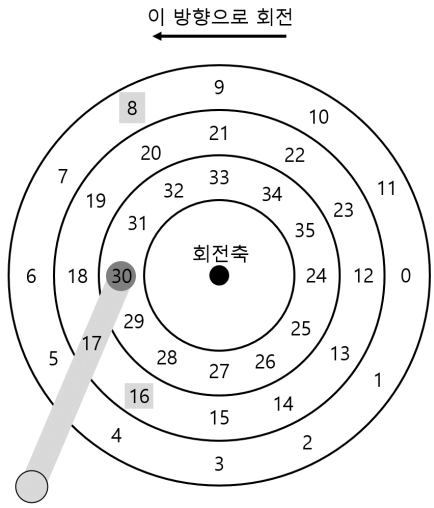
문제에 대한 해법인 **최단 위치 잡기 우선(shortest positioning time first)** 또는 **SPTF** 스케줄링(때로는 **최단 접근 시간 우선(shortest access time first)** 또는 SATF라고도 함)을 논의하기 이전에 문제 자체를 제대로 이해했는지 확인하고 넘어가자. 그림 40.8의 예를 살펴보자.

이 예에서는 헤드가 현재 가장 안쪽의 트랙의 섹터 30번 위에 위치해있다. 스케줄러는 다음의 요청을 처리하기 위해 중간 트랙의 섹터 16번으로 이동할지 바깥 트랙의 섹터 8번으로 이동할지 결정해야 한다. 다음의 차례는 어떤 것이 되어야 할까?

대답은 당연히 "상황에 따라 다르다." 공학 분야의 거의 대부분의 질문에 "상황에 따라 다르다."라고 대답할 수 있다. 왜냐하면 공학도에게는 절충이 삶의 일부이기 때문이다. 상사의 질문에 대한 대답을 모르는 때처럼 곤경에 빠졌을 때 이 격언이 도움이 될 수도 있으니 사용해 보자. 하지만 *왜* 상황에 따라 다른지를 이해하는 것이 중요하다. 이제 우리도 이해해 보자.

여기서 상황에 의존적인 이유는 탐색에 걸리는 시간과 회전에 걸리는 시간이 다르기 때문이다. 이 예제에서 탐색 시간이 회전 지연보다 더 크다면 SSTF(와 그 변종)는 잘 동작한다. 하지만 탐색이 회전보다 훨씬 빠르다고 생각해 보자. 그러면 더 *먼* 탐색을 통해 바깥 트랙에 있는 8번의 요청을 먼저 처리하는 것이 짧게 탐색하여 가까운 중간의 트랙으로 이동한 후 16번 요청을 처리하는 것보다 낫다. 왜냐하면 16번 섹터가 디스크 헤드 아래를 지나가려면 거의 한 바퀴를 다 돌아야하기 때문이다.

현대 드라이브들은 앞서 본 것과 같이 탐색과 회전이 거의 비슷하고 따라서(요청이 무엇이냐에 따라 다르기는 하다) SPTF가 유용하고 성능을 개선시킬 수 있다. 하지만



〈그림 40.8〉 **SSTF만으로는 충분하지 않다**

팁: 항상 상황에 따라 다르다(Livny's Law)

Miron Livny가 늘 말하듯이 거의 모든 질문에 "상황에 따라 다르다."라고 답할 수 있다. 하지만 주의하여 사용하여야 한다. 너무 많은 질문에 이런 식으로 대답하면 당신에게 더 이상 질문하지 않게 될 것이다. 예를 들어, 누군가 "점심 먹으러 갈래?"라고 질문을 했을 때 당신은 "상황에 따라 다른데, *너*가 같이 가는 거야?"라고 대답할 수 있다.

트랙의 경계가 어디인지 현재 디스크 헤드가 어디에 있는지(회전의 관점에서)를 정확히 알 수 없기 때문에 운영체제에서 이것을 구현하기는 매우 어렵다. 그러므로 SPTF는 아래에서 설명한 것처럼 드라이브 내부에서 실행된다.

다른 스케줄링 쟁점들

기본적인 디스크 동작, 그리고 간략하게 다룬 스케줄링과 관련된 주제에 대해 논의할 때 하지 않은 여러 많은 쟁점들이 있다. 그 쟁점들 중 하나는 다음과 같다. 디스크 스케줄링은 현대 시스템에서 *어느* 부분이 담당해야 하는가? 예전 시스템의 경우 운영체제가 모든 스케줄을 결정하였다. 대기 중인 요청들의 집합을 살펴보고 운영체제가 최선의 요청을 선택하여 디스크에게 명령을 내렸다. 요청 처리가 완료되면 다음 요청을 다시 선택하는 식이다. 그때만 해도 디스크도 인생도 간단했다.

현대 시스템에서 디스크는 대기 중인 여러 개의 요청들을 수용할 수 있으며 복잡한 내부 스케줄러를 자체적으로 갖고 있다(이 스케줄러는 디스크 컨트롤러 내부에 있기 때문에 헤드의 정확한 위치도 알 수 있을 뿐만 아니라 그 외의 필요한 정보들도 알 수 있다. 그래서 SPTF 방식을 정밀하게 구현할 수 있다). 그렇기 때문에 운영체제의 스케줄러는

최선이라고 보이는 몇 개의 요청(16번이라고 하자)을 선택하여 모두 디스크에 내려보낸다. 디스크는 상세한 트랙 배치 정보와 헤드의 위치에 대한 내부 지식을 사용하여 최선의 (SPTF) 순서로 정렬한다.

디스크 스케줄러가 수행하는 중요한 또 다른 관련 작업은 **I/O 병합**(**I/O merging**)이다. 그림 40.8에서처럼 블럭 33번, 8번, 그리고 34번을 읽는 연속된 요청이 있다고 하자. 그런 경우라면 스케줄러는 블럭 33번과 34번 요청을 병합하여 두 블럭 길이의 요청으로 만든다. 병합된 요청을 반영하기 위하여 스케줄러는 해당 요청들을 재배치한다. 디스크로 내려보내는 요청의 개수를 줄이면 오버헤드를 줄일 수 있기 때문에 운영체제에서 병합은 특히 중요하다.

현대 스케줄러가 해결해야 하는 마지막 문제는 다음과 같다. 디스크로 I/O를 내려보내기 전에 시스템은 얼마나 기다려야 하는가? 단 하나의 I/O만 있더라도 디스크로 즉시 내려보내야 한다고 순진하게 생각할 수도 있다. 이와 같은 방식은 처리할 요청이 있는 한 디스크는 유휴 상태가 되지 않도록 하는 **작업 보전**(**work-conserving**) 방식이다. 하지만 **예측 디스크 스케줄링**(**anticipatory disk scheduling**) 연구에 따르면 때로는 잠시 기다리는 것이 더 좋다는 것을 보였으며 [ID01], 이를 **작업 비보전**(**non-work-conserving**) 방식이라고 부른다. 기다리다 보면 새로운 "좀 더 좋은" 요청이 디스크에 도달할 수 있으므로 전체적인 효율이 좋아지게 된다. 물론, 언제 기다리고 얼마나 기다리는 것을 결정하는 것은 까다로울 수 있다. 상세하게 알고 싶다면 논문을 참고하거나 (좀 더 야망이 있는 학생이라면) 이런 개념을 실제로 어떻게 구현하고 있는지 알아보기 위해 Linux 커널을 살펴보라.

40.6 요약

이 장에서는 디스크 동작에 대한 정리를 제시하였다. 이 정리는 기능적 모델에 대한 것이기 때문에, 실제 드라이브를 설계하는 데 고려해야 하는 물리나 전자 그리고 재료과학의 놀라운 기술들은 설명하고 있지 않다. 그런 특성에 관한 상세 정보가 흥미롭다면 전공을 바꾸기를 권한다(또는 부전공으로 선택하기를). 이 기능적 모델로 만족한다면 다행이다! 이제 이 놀라운 장치 위에 흥미로운 시스템을 만드는 데 계속해서 이 모델을 사용할 것이다.

참고 문헌

[ADR03] **"More Than an Interface: SCSI vs. ATA"**
Dave Anderson, Jim Dykes, and Erik Riedel
FAST '03, 2003
현대 디스크 드라이브가 실제로 어떻게 동작하는지를 설명하는 최신이라고 할 수 있는 참고 문헌 중 하나이다. 이 분야에 대해 좀 더 알고 싶은 사람이라면 꼭 읽어봐야 한다.

[CKR72] **"Analysis of Scanning Policies for Reducing Disk Seek Times"**
E.G. Coffman, L.A. Klimko, and B. Ryan
SIAM Journal of Computing, September 1972, Vol 1. No 3.
디스크 스케줄링 분야의 초기 연구 중의 하나이다.

[ID01] **"Anticipatory Scheduling: A Disk-scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O"**
Sitaram Iyer and Peter Druschel
SOSP '01, October 2001
잠시 기다리는 것이 디스크 스케줄링의 성능을 높일 수 있다는 것을 보인 멋진 논문이다. 더 좋은 요청이 오고 있어!

[JJ91] **"Disk Scheduling Algorithms Based On Rotational Position"**
D. Jacobson and J.Wilkes
Technical Report HPL-CSP-91-7rev1, Hewlett-Packard (February)
디스크 스케줄링에 관한 좀 더 현대적인 해석. 저자들보다 Seltzer 등[SCO90]이 먼저 출간하는 바람에 기술 문서로만 남아 있다(논문으로 게재되지는 않았다).

[RW94] **"An Introduction to Disk Drive Modeling"**
C. Ruemmler and J. Wilkes
IEEE Computer, 27:3, pp. 17-28, March 1994
기본적인 디스크 동작에 대한 훌륭한 소개 논문이다. 몇몇 정보는 구닥다리가 되었지만, 대부분의 기본은 그대로 사용되고 있다.

[S09a] **"Barracuda ES.2 data sheet"**
URL: `http://www.seagate.com/docs/pdf/datasheet/disc/ds/cheetah/15k/5.pdf`
기술 명세서, 각오하고 읽어야 한다. 어떤 각오? 지루함.

[S09b] **"Cheetah 15K.5"**
URL: `http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf`
참고 문헌 [S09a]에 대한 설명을 참고하자.

[SG04] **"MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?"**
Steven W. Schlosser and Gregory R. Ganger
FAST '04, pp. 87-100, 2004
이 논문에서 MEMS 부분은 아직 큰 영향을 발휘하고 있지는 못하지만 디스크와 파일 시스템 간의 계약에 관한 설명은 훌륭하며 그 공헌은 지금까지 인정받고 있다.

[SCO90] **"Disk Scheduling Revisited"**
Margo Seltzer, Peter Chen, and John Ousterhout

USENIX 1990
디스크 스케줄링 분야에서 회전을 고려하는 것이 중요하다는 것을 말하고 있는 논문이다.

문제

이 문제들에서는 disk.py를 사용하여 현대 하드 드라이브의 동작 방식에 대해서 익숙해질 수 있도록 한다. 여러 다양한 옵션이 있으며, 다른 시뮬레이터와 달리 디스크가 어떻게 동작하는지를 보여주는 애니메이션을 포함하고 있다. 상세 설명은 README를 참고하시라.

1. 다음의 요청들에 대해서 탐색, 회전, 그리고 전송 시간을 계산하라: -a 0, -a 6, -a 30, -a 7, 30, 8 그리고 -a 10, 11, 12, 13.

2. 같은 요청들을 사용하되 탐색 속도로 다른 값을 사용해 보자: -S 2, -S 4, -S 8, -S 10, -S 40, -S 0.1. 시간은 어떻게 달라지는가?

3. 역시 같은 요청들을 사용하고 회전 속도를 바꿔보자. -R 0.1, -R 0.5, -R 0.01. 이때 시간은 어떻게 달라지는가?

4. 어떤 요청 흐름들은 FIFO보다 더 좋은 정책을 사용하면 좋겠다는 생각이 들었을 것이다. 예를 들어 -a 7, 30, 8 이라는 요청 흐름의 경우, 어떤 순서로 요청들을 처리해야 할까? 최단 탐색 시간 우선(SSTF) 스케줄러를 같은 워크로드에 대해 실행하여 보자 (-p SSTF). 각 요청을 처리하는 데 얼마나 오래 걸릴까(탐색, 회전, 전송)?

5. 이번에 다시 같은 작업을 할 때에는 최단 접근 시간 우선(SATF) 스케줄러를 사용해 보자(-p SATF). -a 7, 30, 8로 명시된 요청의 집합의 경우 어떤 변화가 있는가? SSTF 대비 눈에 띄게 SATF가 더 빠르게 서비스하는 요청들의 집합을 찾아보자. 어떤 조건에서 눈에 확연한 차이가 생겨나는가?

6. -a 10, 11, 12, 13의 요청 흐름이 디스크에서 특히 잘 서비스되지 않는 것을 보았을 것이다. 그 이유는 무엇인가? 트랙 스큐를 사용하여 이 문제를 해결할 수 있는가 (-o skew, 이때 skew는 양의 정수)? 디폴트 탐색 속도를 사용한다면 어떤 스큐 값을 사용하여야 전체 서비스 시간을 최소화할 수 있겠는가? 탐색 속도를 바꿔본다면 어떻게 되겠는가(-S 2, -S 4)? 탐색 속도와 섹터 배치 정보를 안다고 했을 때, 스큐 값을 계산할 수 있는 공식을 만들 수 있는가?

7. 멀티 구역 디스크는 바깥측의 트랙에 더 많은 섹터들을 넣는다. 그런 구성을 원한다면 -z 플래그를 사용하면 된다. 디스크의 구역을 -z 10, 20, 30으로 설정하여 몇 개의 요청에 대해 실행시켜 보자. 각 값은 트랙마다 섹터들이 차지하는 공간의 크기를 각으로 정의한 것으로, 이 예에서는 바깥 트랙에 10도 위치마다 섹터가 있고, 중간에는 20도마다, 그리고 안쪽 트랙에는 30도마다 섹터가 위치한다. 임의의 요청을 발생시켜 보자(예, -a -1 -A 5, -1, 0, 여기서 -a -1 플래그는 임의의 요청을 발생시키기 위해 사용되었으며 0부터 최댓값 사이의 다섯 개의 값이 선택된다). 그리고 탐색과 회전 그리고 전송 시간을 계산할 수 있는지 알아보자. 다른 랜덤 시드를 발생시켜서 해 보자 (-s 1, -s 2 등). 바깥 측, 중앙, 그리고 안쪽 트랙의 대역폭(기준 시간 당 섹터의 수)은 무엇인가?

8. 스케줄링의 창(window)은 다음에 처리할 요청을 결정하기 위해 디스크가 한 번에 검사할 수 있는 요청의 개수를 결정한다. 랜덤 워크로드를 아주 많이 발생시켜라(예, -A 1000, -1, 0, 그리고 랜덤 시드도 다른 것을 사용해 보자). 스케줄링의 창 1에서 요청의 전체 수만큼 증가시키면서(예, -w 1과 -w 1000을 사용해 보고 그 사이의 값들도 사용해 보자). 최고의 성능을 얻기 위해서는 스케줄링의 창이 얼마나 커야 하는가? 그래프를 그려서 확인해 보자. 힌트: 더 빨리 실행시키려면 -G 플래그를 사용하여 그래픽 기능을 끄고 -c 플래그를 사용하라. 스케줄링 창이 1로 설정되었을 때 사용하는 정책이 중요한가?

9. 스케줄러에서는 기아 현상을 막는 것이 매우 중요하다. SATF와 같은 기법을 사용할 때 특정 섹터의 처리가 매우 지연되는 경우를 생각할 수 있는가? 지정된 요청에 대해 제한된 SATF(**bounded SATF**) 또는 **BSATF** 스케줄링을 사용하면 어떻게 동작하는가? 이 실험을 할 때에는 스케줄링의 창(예, -w 4)과 BSATF 정책(-p BSATF)을 명시한다. 이렇게 하면 스케줄러는 현재 창의 모든 요청이 다 처리된 후에 다음 창으로 이동한다. 이 접근법이 기아 현상을 해결하는가? SATF 대비 성능은 어떠한가? 일반적으로 기아 현상 회피와 성능 사이에서 디스크는 어떤 절충을 해야 하는가?

10. 지금까지 살펴본 스케줄링 정책들은 **greedy** 방식으로서 요청 집합 중에서 최적의 스케줄을 찾는 대신 차선으로 좋은 선택을 하고 있다. 이러한 greedy 방식이 최적이 아닌 요청 집합을 찾을 수 있는가?