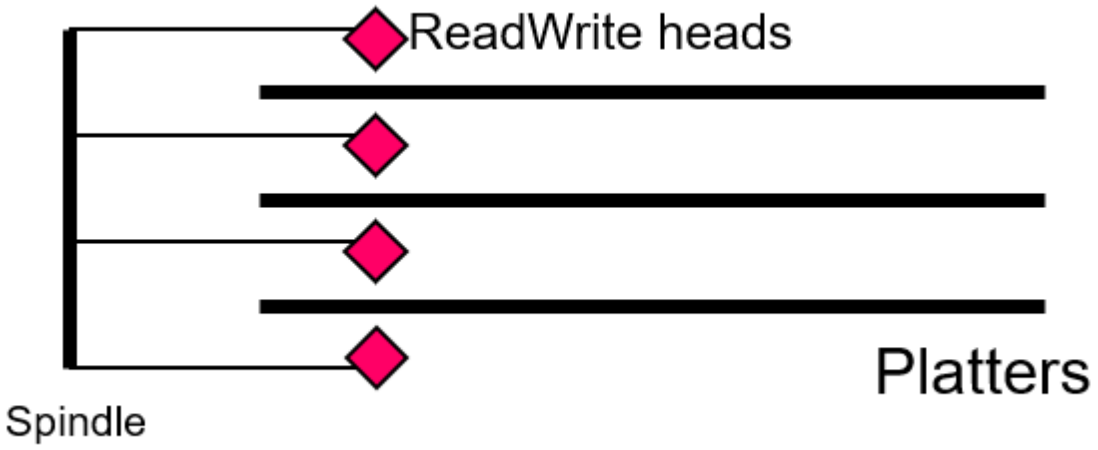
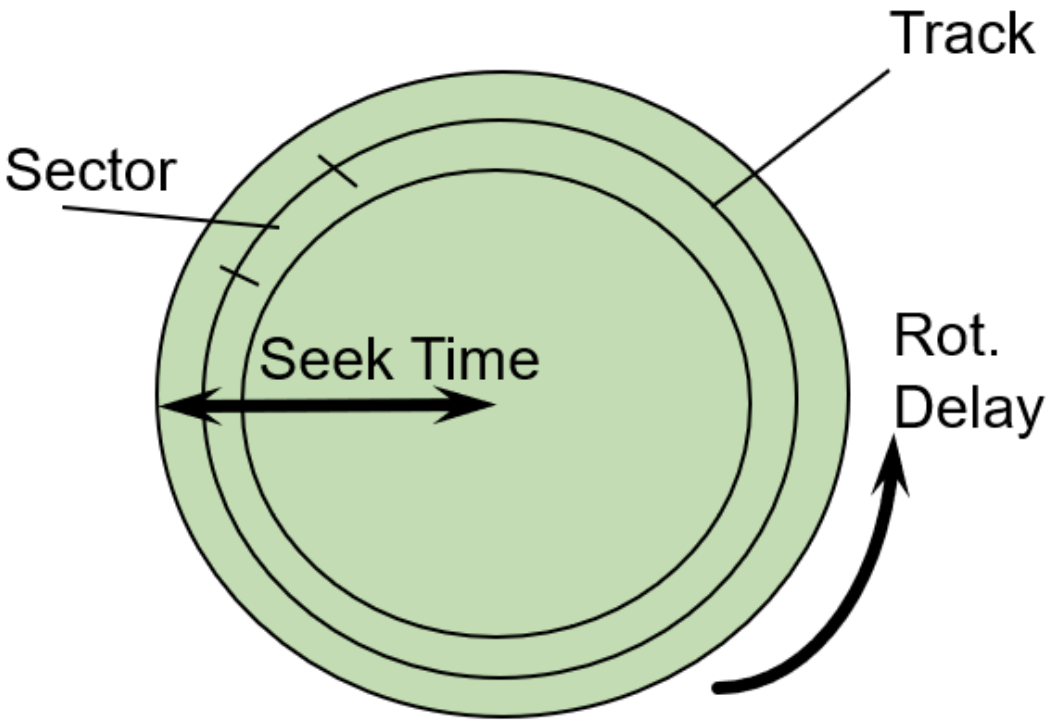
# 13. Mass storage management




- HD를 옆에서 본 그림
- `Spindle` 이 돌면서 head가 안쪽과 바깥쪽을 모두 확인할 수 있다.
  - 원판이 도는 것
- 각 head가 다른 platter 여도 같은 위치의 track 가리킴 → Cylinder: 여러 track의 데이터를 한번에 받아올 수 있다.
- 각 head가 위에서부터 1,2,3,4로 정해지면 각 track의 num도 정해진다




- HD를 위에선 본 그림
- 여러 개의 마그네틱 원판 (PLatter)
  - 양면으로 각각에 head가 하나씩 붙을 수 있다.
- 여기서는 3개의 flatter → 6개의 head가 가능
- `Track` : 헤더가 움직여서 생기는 원
  - 같은 Track에 있는 데이터는 한 번의 head 움직임 만으로 다 확인 가능하다.
- `seek time` **: Head를 움직여서 원하는 Track에 가는데 걸리는 시간**
- `Rot Delay` **: 원판이 돌면서 한 track 내에서 원하는 데이터가 있는 위치를 확인하는 데까지 걸리는 시간**
- `sector` : Disk에서 가장 작은 데이터 단위 (512 byte)
  - 물리적인 단위이다.
  - format의 단위
  - format 후에 sevtor 8개를 block으로 묶는 것은 소프트웨어적인 행동

아래의 것들이 정해져야 HW가 동작한다.

CHS (Cylinder, Head, sector)

Head 위치 결정 → Cylinder 형성Track의 번호를 head의 번호로 확인 가능 → 어느 track을 읽을지 정해지면 → 그 track에서 sector number에 따라 어떤 sector를 읽을지 결정

- HW는 CHS 체계이다.
    - 위 세 개가 정해지면 하나의 sector가 정해진다.

Transfer size

- 한꺼번에 이동할 수 있는 데이터의 양
- Bandwidth를 결정

Memory address

- 읽은 데이터를 메모리에 저장하기 위해

OS는 Hard disk의 구조를 모르고 그냥 저장장치라고 알고 있다.

- OS는 하드웨어 공간을 Logical block address (LBA)로 알고있다.
- **OS는 LBA로 하드웨어 공간을 다루고 HW가 이를 CHS 로 바꾸어서 동작**

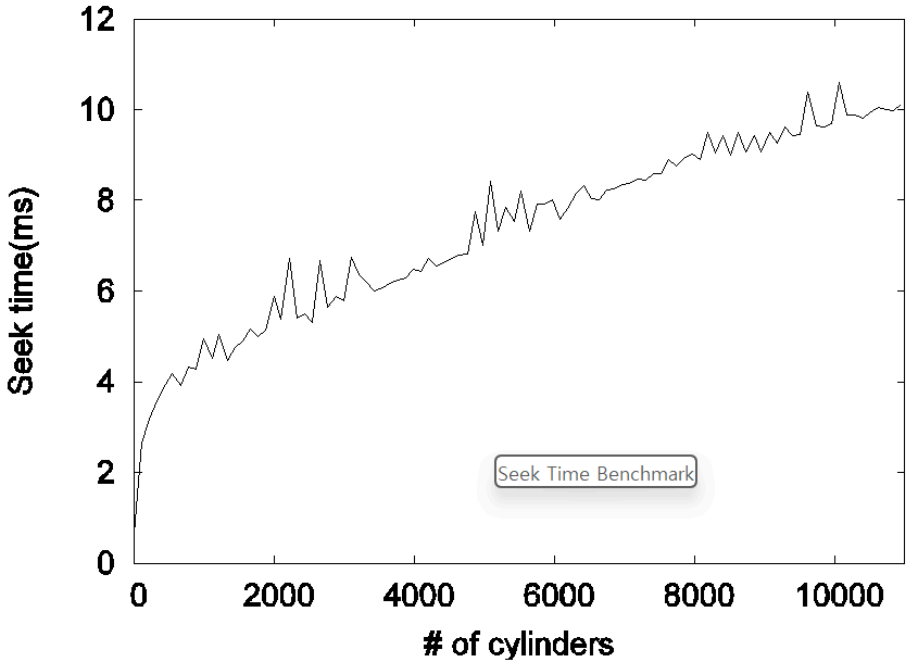
그럼 주소 변환을 어떻게??

- Sector 0: 가장 바깥쪽 track의 가장 위 Head에 해당
- 이후 Sector num을 정하는 거는 HW에 따라 다르다.

# Disk scheduling

Access time: HD에서 데이터를 읽고 쓰는데 걸리는 시간

- Rot delay : RPM이 클수록 빨리 돈다.
  - 빨리 돌수록 좋다.
- Seek time
  - head를 움직임
  - 가장 많이 소요
  - **Seek distance에 비례 (HEAD의 움직임)**




*Seek time model: $a + b\sqrt{d}$ (a, b: constants, d: distance in # of cylinders)*

- seek time이 distance에 따라 어떻게 변하는지를 시각화
- 가로축이 distance
- distance의 루트에 비례

1. **기본적으로 seek을 했을 때, 4ms는 소요된다.**
   - 이때는 seek 자체가 발생하지 않아야 절약 가능
2. **다른 부분은 seek distance를 줄여야 한다.**

**OS는 LBA로 요청을 한다.**

- LBA가 연속이면 HD에서도 연속이다.
- LBA 간격이 짧으면 HD사이 거리도 짧다 라고 믿고 싶다.

LBA가 연속적이 되도록, 연속적이지 않으면 가까운 것부터 처리하자

- 요청의 순서를 조정하여 이를 지원하자

**Disk scheduling**

Disk에 i/o 를 요청할 때, 순서대로 보내지 말고 요청 순서를 조정하자.

이때, LBA가 cylinder num에 매핑이 되어있다고 가정하자

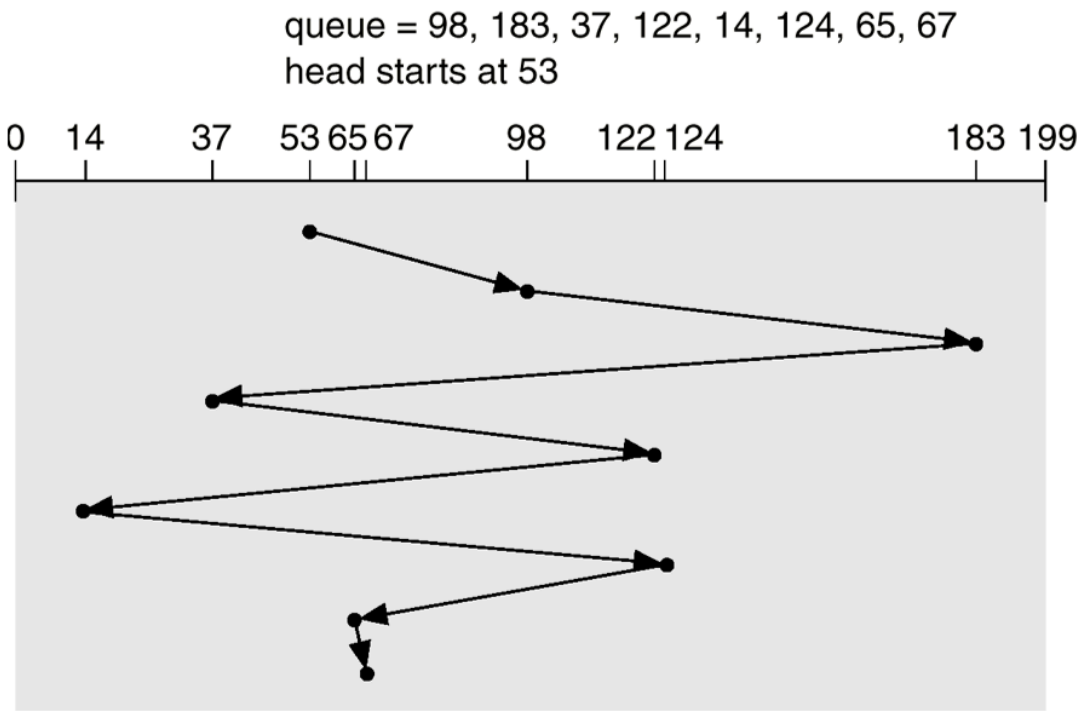
We illustrate them with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

Current head position: 53

- 현재 cylinder 위치가 53이라고 가정하자.

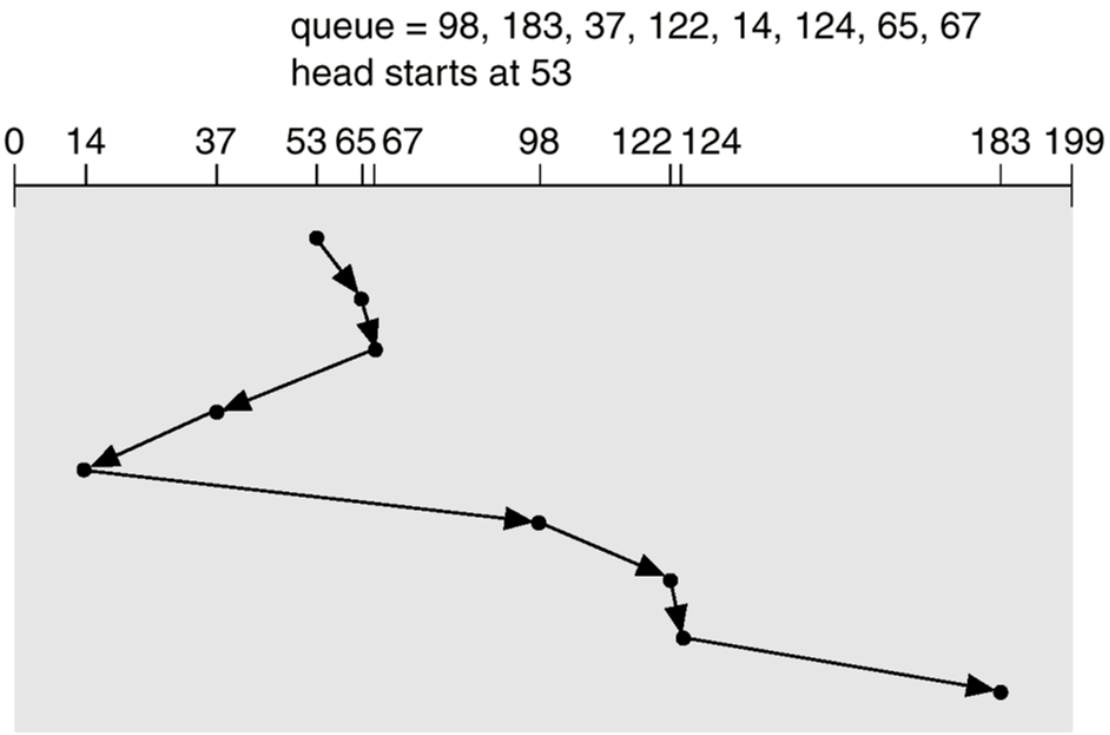
# 1. FCFS




- I/O 요청 온 순서대로 해보자

## 2. SSFF (Shortest Seek Time First)

- seek time이 작은 것부터 먼저 처리
- 현재 head 위치에서 가장 가까운 것을 먼저
- 멀리 있는 것은 `Starvation` 가능

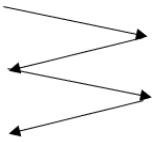



- Fairness 가 큰 문제
- **내가 언제 서비스를 받을 지에 대한 Upper bound가 존재하지 않아서 사용할 수 없다.**

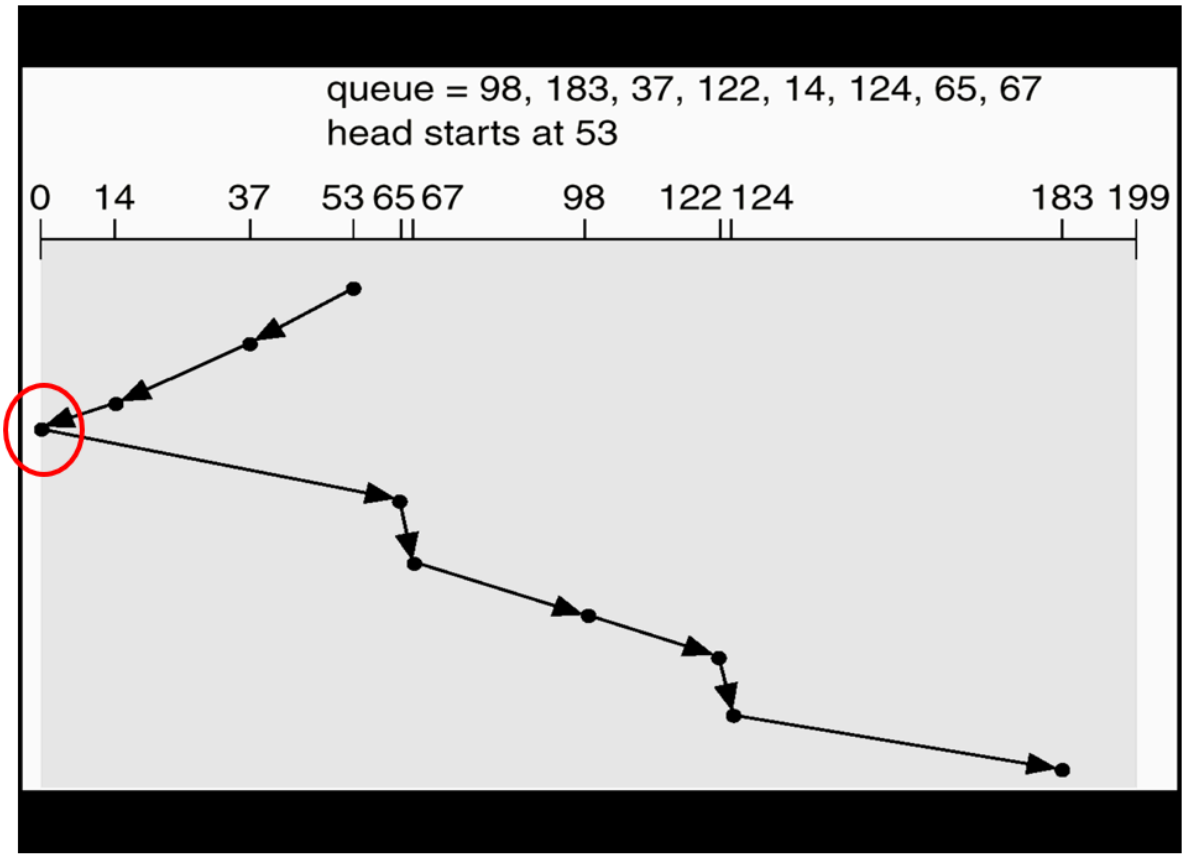
## 3. SCAN

Sometimes called the *elevator algorithm*

- First, service all requests while going up
- Then service all requests while going down



한번 head의 이동 방향이 정해지면 그 방향으로 이동해야 하는 요청을 모두 처리할 때까지 방향을 바꾸지 않는다.

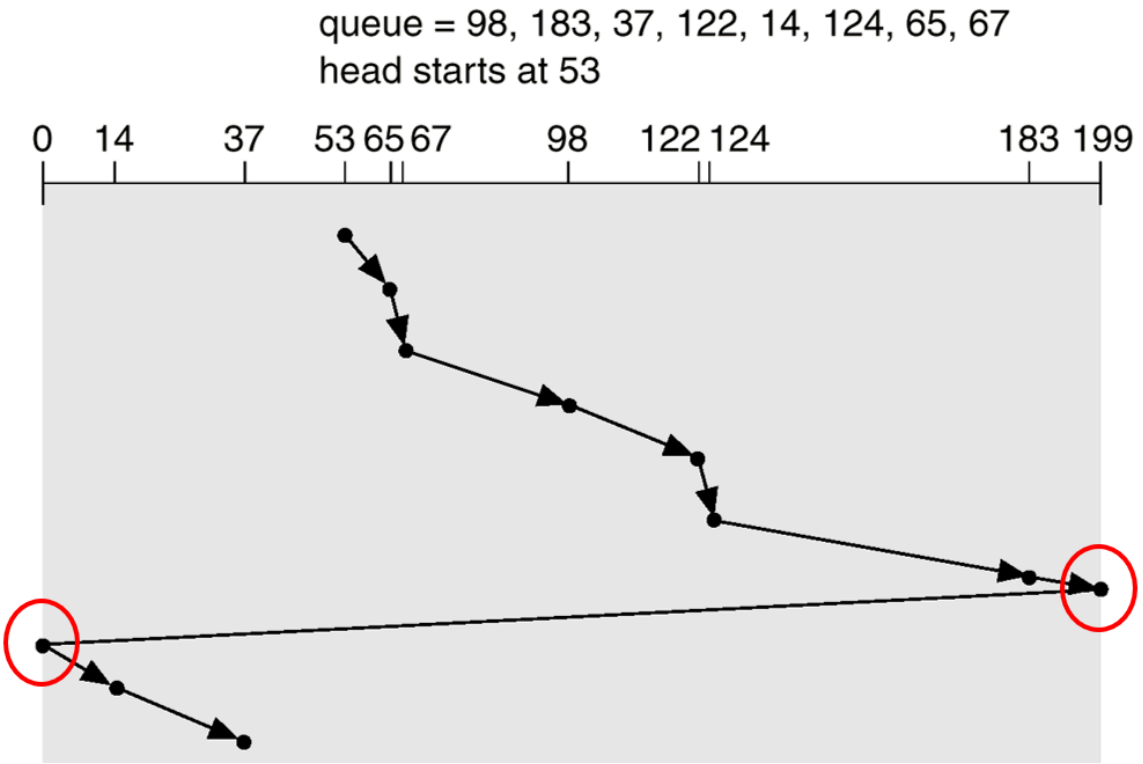



- 처음을 왼쪽으로 이동한다고 가정하자.
- Starvation이 생기지 않음
  - 한번 왔다갔다 하는 동안 반드시 서비스를 받을 수 있다.
  - Upper bound를 제공한다.
- 문제점
  - request가 없는 경우에도 0까지 간다.
  - 양 끝단은 왔다갔다 하는 동안 1번만 서비스 받는다.
    - 양 끝쪽에는 평균 대기 시간이 길다.

## 4. C-SCAN (Circular-SCAN)

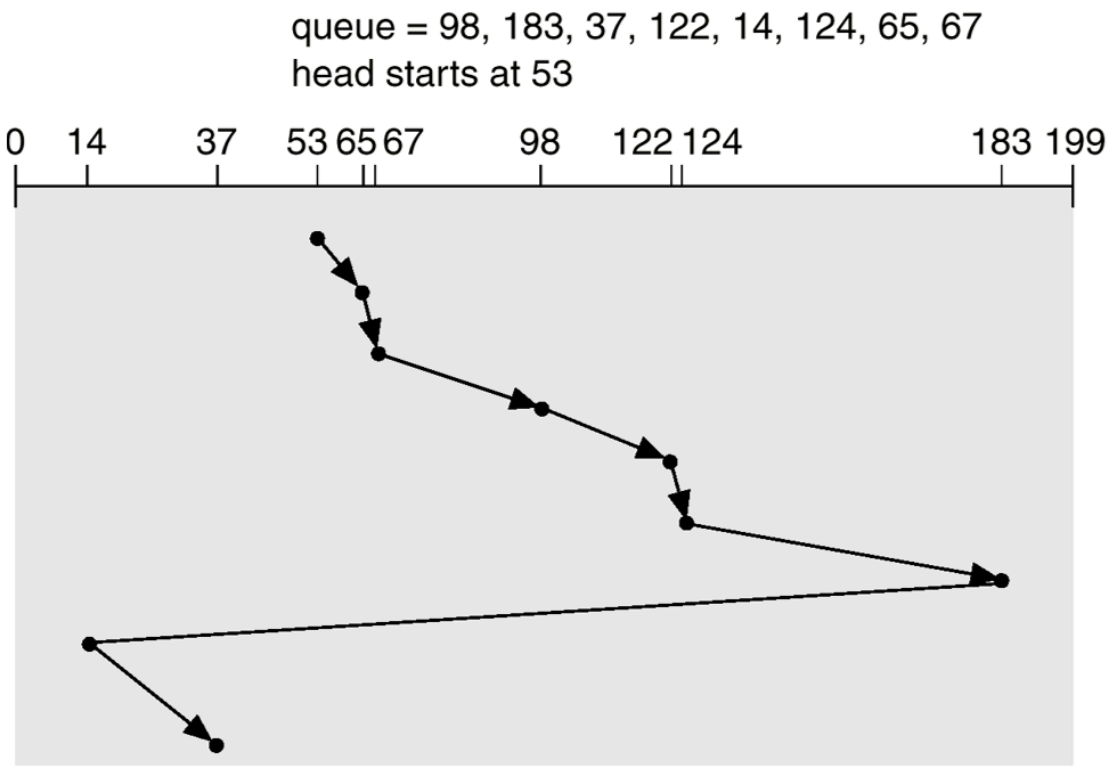
- 양쪽 끝단이 한번만 서비스를 받는 문제를 해결

- 한쪽 방향으로 이동할 때만 서비스를 해주고, 다른 방향으로는 서비스가 아니라 빨리 돌아옴
  - 빨리 돌아올 수 있다.
  - 한쪽 방향으로 가는 것만 서비스를 기다리면 되므로, worst wating time이 적다.
  - 한쪽 끝 찍고 바로 돌아오니까 시작과 끝이 붙어있는 것처럼 생각할 수 있다.
- 그러나 돌아올때 의미 없이 seek time을 가진다.
  - throughput이 SCAN에 비해 낮을 수 있다.




## 5. C-LOOK

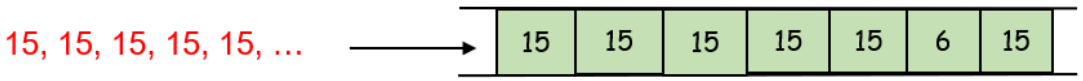
- 한 쪽 방향으로 이동할 때, 전진하는 방향에 현재 request가 없으면 처음으로 바로 돌아오겠다.
- 돌아올때도 이전 방향에 request가 있는 곳까지만 돌아온다.




## 6. LOOK

- SCAN에 대응되는 lokk

## Arm stickiness problem




- FCFS를 제외하고, 15번째가 계속 들어오면 6번은 계속 대기해야 한다.

## Physical formatting

- HD 공간을 전부 Sector 단위로 자른다

- 각 Sector 는 Head + 512 KB + Tailer
  - Head나 tailer는 ECC
  - ECC는 error detect 후 error가 있다면 bad sector로 이동
- Sparse sector : LBA에 매핑되지 않는 HD 공간
  - BAD sector가 한번 발생하면 이 공간은 계속 Bad sector가 될 가능성이 크다.
  - 따라서 Bad sector는 사용하지 않도록 해야한다.
  - Bad sector는 LBA mapping 해제하고, Sparse sector 중 하나가 기존 LBA에 매핑되도록 한다.

# Partition

physical disk → logical disk

- 하나의 HD인데 이걸 partition하면 c drive, d drive 로 나누는 등의 행위가 가능하다
- 디스크가 두개가 있는것처럼 OS가 관리할 수 있다.

# Power-up

- ROM에 small bootstrap loader가 실행되고, Sector 0를 load
- Sector 0에는 full bootstrap loader program이 동작
  - 이 위치는 OS 설치될 때 정해짐

# RAID

- **Redundant Array of Inexpensive (Independent) Disks**
- 값싼 디스크를 여러 개를 묶어서 **하나의 Disk처럼 보이게 만드는 방법**
- 속도 빠르고, 용량 크다

문제점)

- 신뢰성 문제가 생긴다.

- **N개의 disk 쓸 때, 그 중 하나의 DIsk만 고장나도 전체가 고장난다.**

The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail

- E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
- Techniques for using redundancy to avoid data loss are critical with large numbers of disks

- MTTF: HD가 고장이 나는 시간 간격

해결)

- Data를 Redundant 하게 여러 디스크에 복제해서 저장할 수 있도록 해야 한다.

데이터를 그럼 어떻게 분할? 어떤 DISK에 저장?

Redundancy를 어떻게 구현?

Raid level: 데이터 분할 방식 + Redundancy 구현 방식에 따라 조합이 달라진다.

- 이를 Level에 따라 다르게 구현

**Redundancy를 구현하는 가장 쉬운 방법**

mirroring : 하나의 디스크하고 똑같은 디스크를 하나 더 두자

- **전체 용량의 반밖에 사용하지 못 하지만 신뢰성은 크게 높일 수 있다.**

Mean time to data loss depends on mean time to failure, and mean time to repair

- E.g. MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of $500*10^6$ hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

- Mirroring까지 된 두 디스크가 모두 고장이 나게 되려면 57,000년이 소요

- 하나가 고장나고, 10시간 내에 다른 것도 고장나야한다.

**데이터를 어떻게 분할할까?**

- file별로 데이터를 분할하는 것은 의미가 없다
- 파일 내부에 있는 데이터를 잘라야 한다.

우선 아래 두 방법을 쓰면 동시에 N개의 Disk를 접근할 수 있기에 병렬성은 좋다.

Bit-level striping : Bit 단위로 자른다.

- bit 별로 읽을 때마다 seek 이 발생하면 seek overhead 너무 큼
- 사용하지 않는다.

Block-level striping : Block 단위로 자른다.

- 첫 번째 block은 첫 번째 디스크에, 두 번째 block은 두 번째 디스크에 저장하는 방식으로 효율성을 높임
- 여러 개의 block 요청이 오면 한 번에 여러 디스크에서 한 번에 읽어올 수 있다.

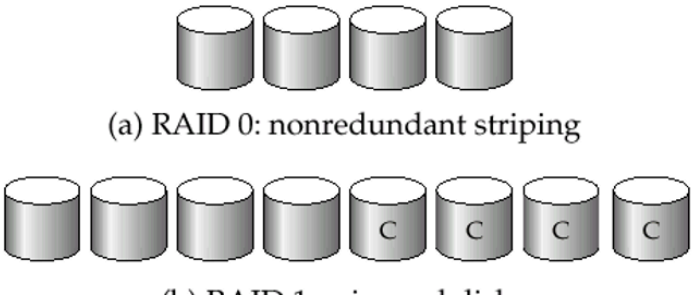
# Raid level

RAID Level 0: Block striping, non-redundancy

- Used in high-performance applications where data loss is not critical

RAID Level 1: Mirrored disks with block striping

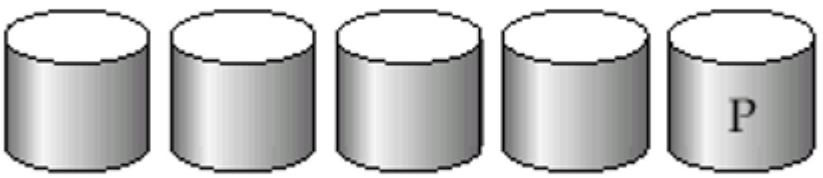
- Offers best write performance
- Popular for applications such as storing log files in a database system




(a) RAID 0: nonredundant striping

- Level 0는 오직 성능만을 위해, block striping 사용
  - redundancy는 사용하지 않음
- Level 1은 block + mirroring
  - Write preformance에 최적화

## RAID Level 4: Block-interleaved parity, block-level striping

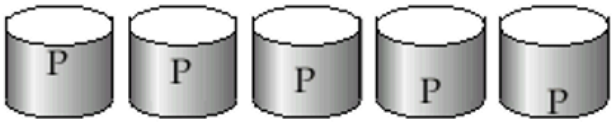



(e) RAID 4: block-interleaved parity

- Parity disk를 하나를 추가로 보유한다.
- Even parity면 이전 4개의 1의 개수가 짝수면, P에서 parity bit가 0이 되어야 한다.

- 만약 4개의 disk 중 하나가 깨지면, 전체적인 Even parity가 맞지 않는다.
  - 이 경우에 Parity disk를 이용해서 복구한다.
  - XOR 이용
- 이 때, 쓰고 있을 때, worst case에는write 시에는  read 2번, write 2번 발생
  - Parity block이 업데이트 되는 경우, 이를 확인하기 위해 기존 Parity block의 값을 읽어야 함.
  - 만약 변경되었다면, Parity disk에서의 Parity 값도 확인해야한다. (Read)
- wirte 시, parity disk에는 반드시 write 해줘야함
  - Parity block이 bottleneck이 된다.

## RAID Level 5: Block-interleaved distributed parity, block-level striping




(f) RAID 5: block-interleaved distributed parity

| | | | | |
|---|---|---|---|---|
| P0 | 0 | 1 | 2 | 3 |
| 4 | P1 | 5 | 6 | 7 |
| 8 | 9 | P2 | 10 | 11 |
| 12 | 13 | 14 | P3 | 15 |
| 16 | 17 | 18 | 19 | P4 |

- 5번째 parity disk도 block striping

- 4번째보다 성능은 좋은데, 단점은 없음
- **Level 4 never used** since it is subsumed by level 5

**RAID 0 is used only when data safety is not important**

- 성능이 중요한 경우에 사용
- E.g. data can be recovered quickly from other sources

결국 남는 것은 1번, 5번 뿐

Level 1: Mirroring

- Update 시, write 2번
- 용량이 적어진다는 단점

Level 5

- Update 시, read 2번, write 2번

Update overhead → Level 5 > Level 1

현대 시점에 DISK가 용량이 부족한 경우는 거의 없다. 따라서 Mirroring에 의해  용량이 부족해진다는 점을 신경 쓰지 않아도 된다.

High update 의 경우에는 Level 1

- 일반적인 경우에도 동일

Low update + Large data에는 Level 5