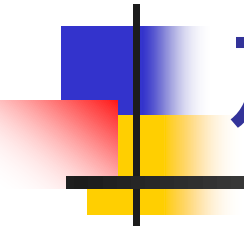
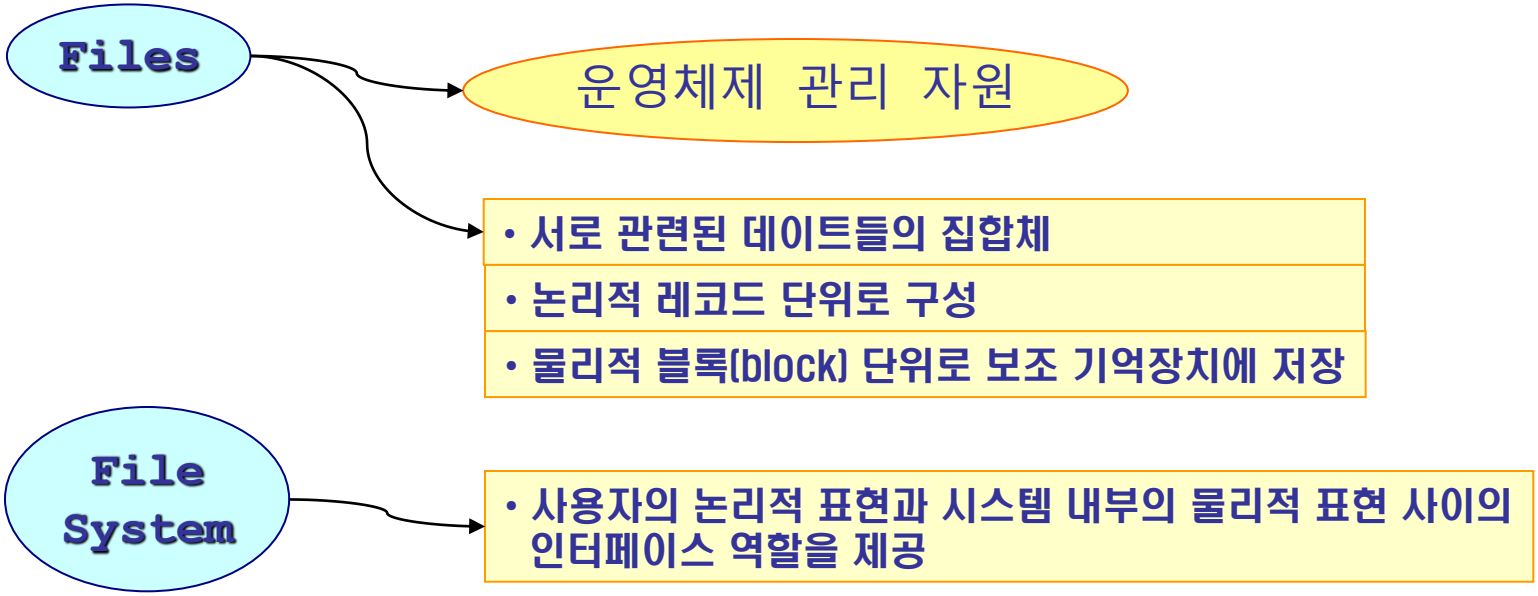


제6장 파일 시스템

제6장 파일 시스템

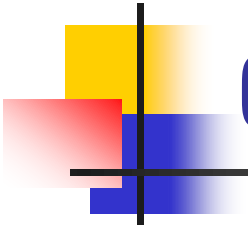


6.1 개요

- 파일 시스템 기능
  - ① 파일 생성(creation)과 삭제(deletion)
  - ② 파일 열기(open)와 닫기(close)
  - ③ 파일 읽기(read)와 쓰기(write)
  - ④ 파일 공유(sharing)와 보호(protection)

- 사용자의 요청에 따라 디스크와 같은 보조 기억장치에 파일의 내용을 저장하고, 저장된 파일의 내용을 다시 사용할 수 있도록 사용자와 보조 기억장치 사이의 인터페이스 기능을 제공함

- 사용자의 논리적 단위인 파일과 보조 기억장치의 물리적 단위인 블록 사이의 효율적인 매핑(mapping) 과정이 요구됨



6.1.1 파일 개념

- 관련을 갖는 논리적인 레코드(record)의 집합
  - 물리적인 레코드 혹은 블록 단위로 보조 기억장치에 저장
  - 논리적 레코드 크기와 물리적 블록 크기는 상이할 수 있음
  - 논리적 연속성이 물리적 연속성을 보장하지는 않음
- 일반 사용자들이 자료를 저장하기 위한 논리적인 단위로 사용
  - 임의의 파일명을 부여하여 데이터를 저장하고, 파일명을 사용하여 참조
- 속성: 파일명, 파일 종류, 크기, 작성 날짜 및 시간 등

- 파일의 레코드들을 보조 기억장치에 배치하는 방법
  - 보조 기억장치의 특성과 밀접한 관련
- 파일의 레코드들을 접근하는 방법
  - 보조 기억장치에 배치된 방법에 따라 다름

6.1.1 파일 개념

(1) 순차 접근(sequential access)

- 파일을 구성하는 임의의 레코드를 순차적으로 접근하는 방식
- 자기 테이프와 같이 순차적인 저장 특성을 갖는 보조 기억장치에 적용
- 파일을 구성하는 레코드들을 논리적 순서와 동일하게 보조 기억장치에 연속적으로 배치

☺레코드 배치가 단순하며, 저장된 순서에 따라 연속적으로 접근하기 때문에 다음 레코드 접근 속도가 빠름

☹특정 레코드를 접근할 경우 항상 파일의 첫 번째 레코드부터 찾아야 하므로 시간 소모적임

☹파일의 레코드들을 연속적으로 배치해야 하므로 보조 기억장치의 단편화 현상이 발생할 수 있음

☹파일 내용을 추가 혹은 삭제할 경우 파일 전체의 레코드를 재배치해야 하는 어려움이 존재

6.1.1 파일 개념

(2) 직접 접근(direct access)

- 파일을 구성하는 임의의 레코드를 직접 접근하는 방식
  - 레코드들에 대한 논리적 순서와 무관하게 보조 기억장치에 불연속적으로 접근 가능 → 다음 레코드에 대해 직접 접근 요구
- 디스크와 같이 직접 접근이 가능한 보조 기억장치에 적용

☺파일 레코드들을 불연속적으로 배치할 수 있으므로, 보조 기억장치의 빈 공간을 효과적으로 이용할 수 있음

☺파일 내용의 추가 혹은 삭제할 경우 파일 전체의 레코드를 재배치할 필요가 없음

☹파일의 레코드가 불연속으로 배치된 경우, 다음 레코드를 검색하는 오버헤드 발생

6.1.1 파일 개념

(3) 색인 접근(indexed access)

- 파일을 구성하는 임의의 레코드에 대해 색인을 통해 순차 혹은 직접 접근하는 방식
- 색인은 파일의 레코드가 저장된 보조 기억장치의 주소를 가짐
- 파일을 구성하는 레코드들은 연속 혹은 불연속적으로 배치 가능
  - 연속 배치 파일: 색인 순차 접근
    - 자기 테이프와 같이 순차적인 저장 특성을 갖는 보조 기억장치에 적용
  - 불연속 배치 파일: 색인 직접 접근
    - 디스크와 같이 직접 접근이 가능한 보조 기억장치에 적용

6.1.2 디렉토리 구조(Directory Structure)

- 디렉토리 (or folder ?)
  - 파일의 한 종류로, 보통 파일(ordinary file)과 구별
  - 파일 관리에 필요한 정보(파일명, 속성, 파일 주소 등)를 유지
  - 보통 파일(ordinary file)의 접근은 디렉토리 참조를 통해 수행되며, 디렉토리의 구조 및 내용은 운영체제에 따라 상이함

① 1-단계 디렉토리(single-level directory)

② 2-단계 디렉토리(two-level directory)

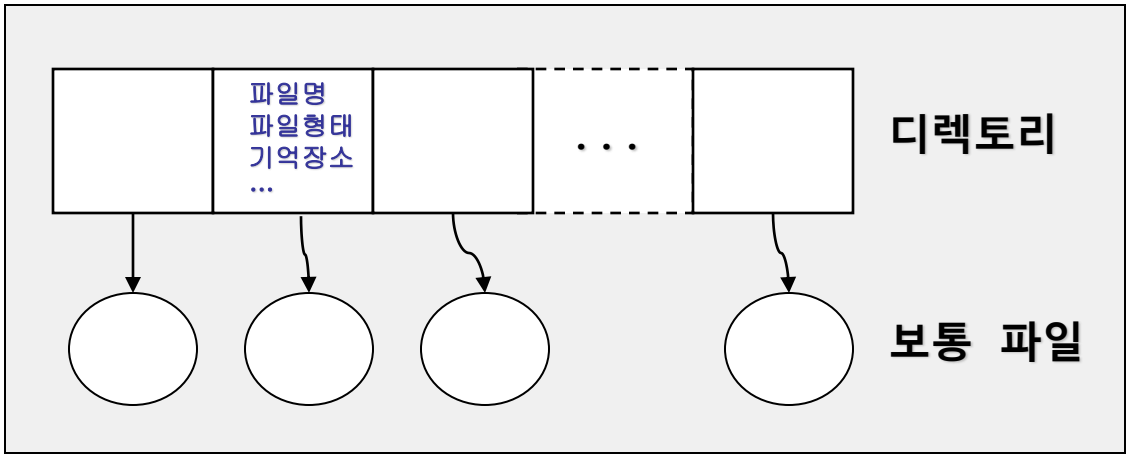
③ 트리 구조 디렉토리(tree-structured directory)

④ 그래프 구조 디렉토리(graph-structured directory)

6.1.2 디렉토리 구조(Directory Structure)

(1) 1-단계 디렉토리(single-level directory)

- 파일 시스템 내부에 오직 하나의 디렉토리만을 유지하는 구조
- 모든 보통 파일의 접근은 한번의 디렉토리 참조로 가능
- 구조는 매우 간단하지만 파일의 수가 증가할 경우 모든 파일명을 다르게 부여해야 함

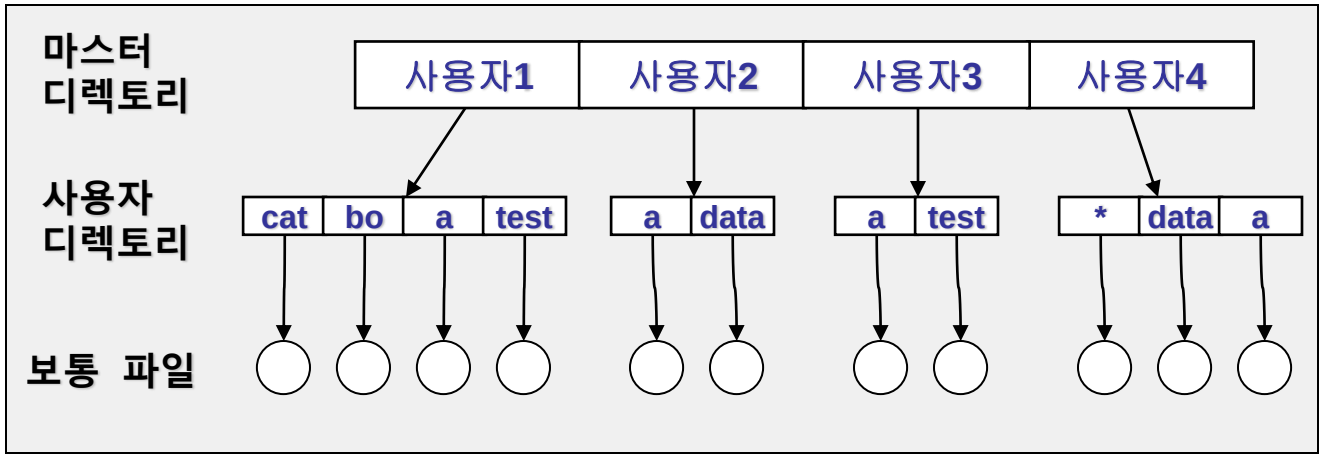


[그림 6.1] 1-단계 디렉토리 구조

6.1.2 디렉토리 구조(Directory Structure)

(2) 2-단계 디렉토리(two-level directory)

- 마스터(master) 디렉토리와 사용자(user) 디렉토리를 유지
  - 마스터 디렉토리: 모든 사용자 디렉토리에 대한 정보를 포함
  - 사용자 디렉토리: 각 사용자의 파일에 관한 정보를 포함
- 파일 접근
  - 사용자명과 파일명을 반드시 명시해야 함
  - 항상 두 번의 디렉토리 참조 요구

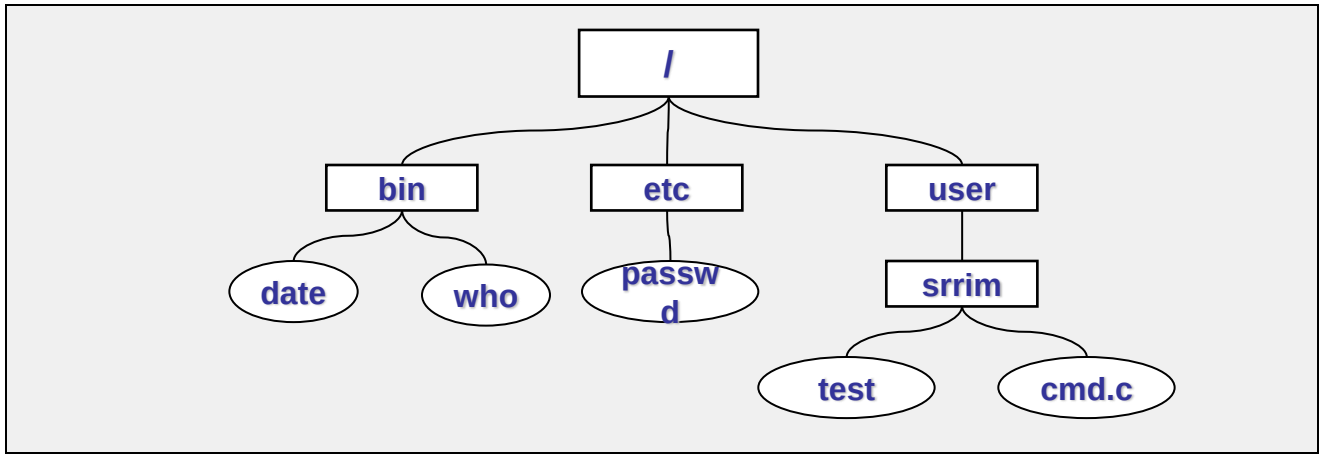


[그림 6.2] 2-단계 디렉토리 구조

6.1.2 디렉토리 구조(Directory Structure)

(3) 트리 구조 디렉토리(tree-structured directory)

- 루트(root) 디렉토리와 서브(sub) 디렉토리로 구성
- 모든 파일은 고유의 경로명(path name)을 가짐
  - 절대 경로: 루트 디렉토리로부터 특정 파일까지의 경로
  - 상대 경로: 현재(current) 디렉토리로부터 특정 파일까지의 경로
    - 현재 디렉토리(.), 부모 디렉토리(..)

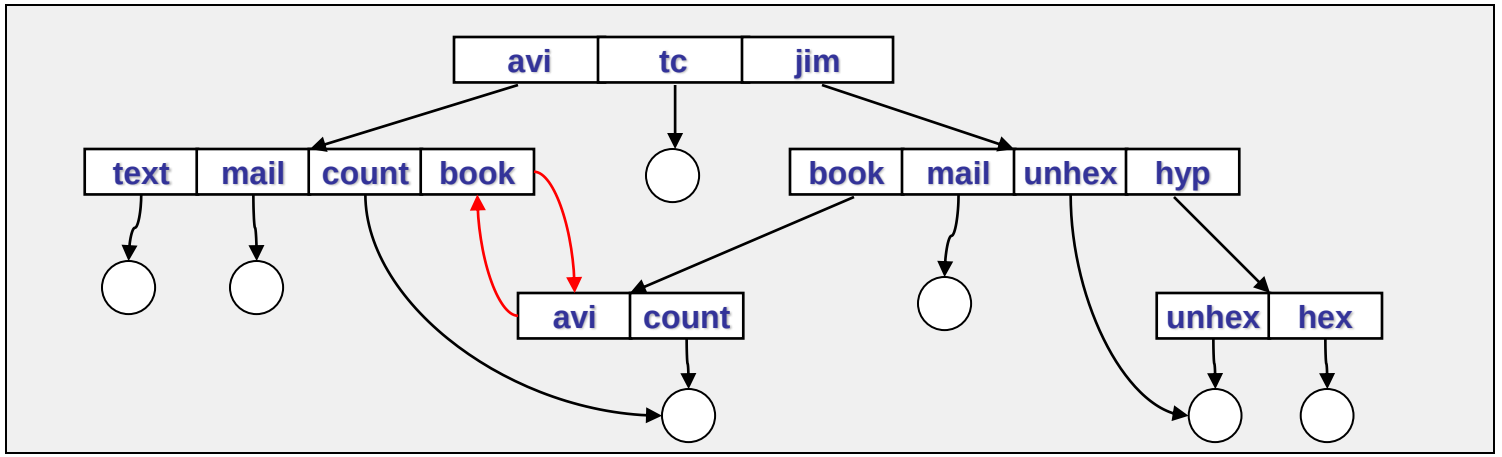


[그림 6.3] 트리 구조 디렉토리

6.1.2 디렉토리 구조(Directory Structure)

(4) 그래프 구조 디렉토리(graph-structured directory)

- 모든 서브 디렉토리와 파일의 공유가 허용
- 동일한 파일을 서로 다른 경로명으로 지정할 수 있음
- 디렉토리 혹은 파일을 공유할 경우 이들에 대한 경로명이 축약될 수 있으므로, 파일에 대한 접근이 용이해질 수 있음
- 서브 디렉토리 혹은 파일들이 순환(cycle) 구조로 연결될 경우 이들에 대한 탐색(traversal) 과정에서 무한 루프에 빠질 수 있음
  - 비순환(acycle) 그래프 구성으로 무한 루프 방지 가능



[그림 6.4] 그래프 구조 디렉토리

6.1.3 파일 보호

- 물리적인 손상과 불법적인 접근으로부터 보호가 필요
  - 물리적 손상 방지
    - 파일 내용에 대한 사본 유지
    - 사용자 → 직접 수행, 시스템 → 자동 수행
- 공유 파일에 대한 보호 방법
  - 파일명(file name)에 의한 방법
    - 사용자가 자신만이 알 수 있는 파일명을 부여
    - 다른 사용자가 파일명을 알 수 없고 추측하기 어렵다는 가정에 근거
    - 사용자 자신은 기억이 용이한가? 파일 자체에 대한 접근은 허용?
  - 암호(password)에 의한 방법
    - 사용자가 자신의 파일에 암호를 부여
    - 사용자가 부여한 암호에 대한 암호화 & 해독 알고리즘 필요

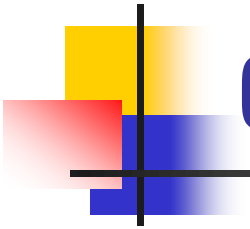
6.1.3 파일 보호

- 공유 파일에 대한 보호 방법(계속)
  - 접근 제어 행렬(access control matrix)에 의한 방법
    - 각 파일에 대한 사용자의 접근 권한을 부여
    - 파일 접근이 허용된 사용자에게만 공유 가능 → 접근 권한 제어 가능

	사용자-1	사용자-2	사용자-3
파일-1	R,W	R	R
파일-2	R,X	R,W,X	R,X
파일-3	R	R	R,W

[그림 6.5] 접근 제어 행렬

  - 모든 파일과 사용자에 대한 접근 제어 행렬이 필요
  - 임의 파일에 대한 사용자 접근 권한 여부를 검색하는 오버헤드 존재
  - UNIX, LINUX: 사용자 구분 → (소유자,그룹,기타),
    접근 권한 구분 → (읽기,쓰기,실행)



6.2 파일 공간 관리

- 파일 내용을 저장하기 위해서는 보조 기억장치의 빈 공간을 파악하고, 빈 공간을 할당하는 기법이 요구됨

- 디스크와 같은 보조 기억장치의 빈 공간을 관리하는 기법
- 빈 공간 할당하는 기법

6.2.1 빈 공간 관리

- 파일 시스템은 새로운 파일 생성을 위해 보조 기억장치의 빈 공간 위치와 크기를 관리
- 빈 공간 관리 기법

  (1) 비트맵(bit map)

  - 보조 기억장치의 모든 블록에 대해 각 블록의 사용 여부를 비트 단위로 표시하는 기법
  - 사용중인 블록 → 1, 미사용 → 0
    - 20개 블록으로 구성된 디스크에서 블록번호 0,1,2,10,11,15,16,17이 사용중인 경우

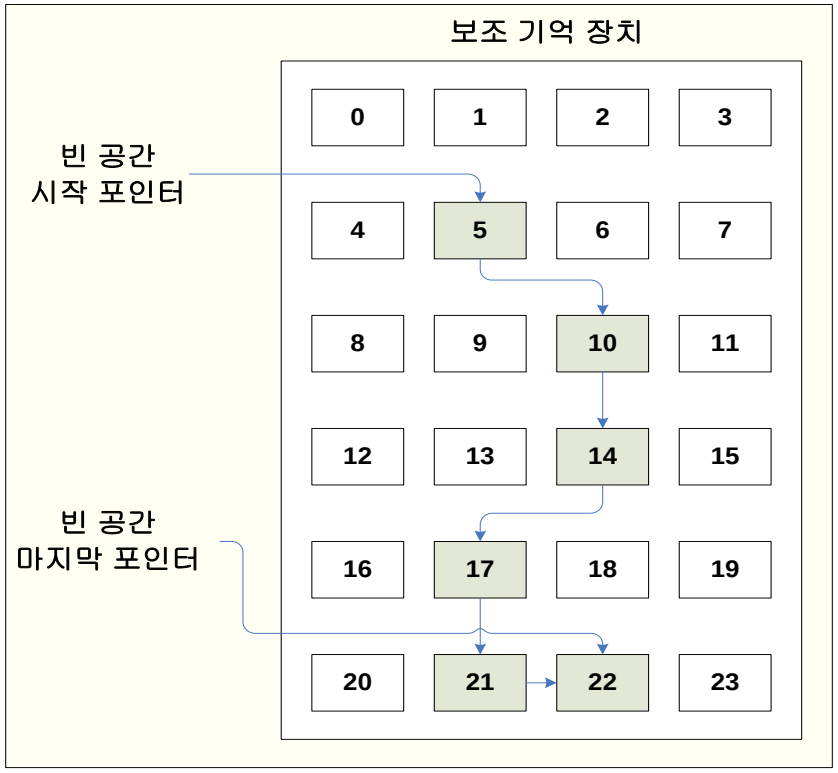
  - 파일 시스템은 비트 맵의 내용을 메모리에 적재하여 사용
  - 구현 용이 & 여러 개의 연속적인 빈 공간을 요구할 경우 효과적
  - 대용량의 보조 기억장치에는 부적합

6.2.1 빈 공간 관리

- 빈 공간 관리 기법(계속)

(2) 연결 리스트(linked list)

- 보조 기억장치의 모든 빈 공간을 연결 리스트 구조로 관리하는 기법



- 빈 공간 시작포인터와 마지막 포인터 유지
- 보조 기억장치 용량에 무관하게 적용 가능
- 단점:
  - 다음 빈 공간 탐색 및 빈 공간 추가 시 보조기억장치 접근 오버헤드 존재
- 개선 방안
  - 계수**(counting)** 기법
    - 보조 기억장치의 빈 공간이 연속적으로 존재할 경우 첫 번째 빈 공간의 주소와 연속된 빈 공간의 계수**(count)**를 지정하여 다음 빈 공간의 탐색 시간을 최소화함
    - **but,** 불연속적으로 존재하는 빈 공간들은 포인터로 연결
  - 그룹핑**(grouping)** 기법
    - 첫 번째 빈 공간에 빈 공간들의 주소를 지정하여 다수의 빈 공간을 한 번에 탐색할 수 있도록 함
    - 빈 공간 수가 너무 많아 첫 번째 빈 공간이 부족할 경우 또 다른 빈 공간을 포인터로 연결하여 나머지 빈 공간의 주소가 지정

[그림 6.6] 연결 리스트를 이용한 빈 공간 관리

6.2.2 빈 공간 할당

- 새로운 파일 생성을 위해 빈 공간의 크기와 위치를 확인한 후 빈 공간을 할당

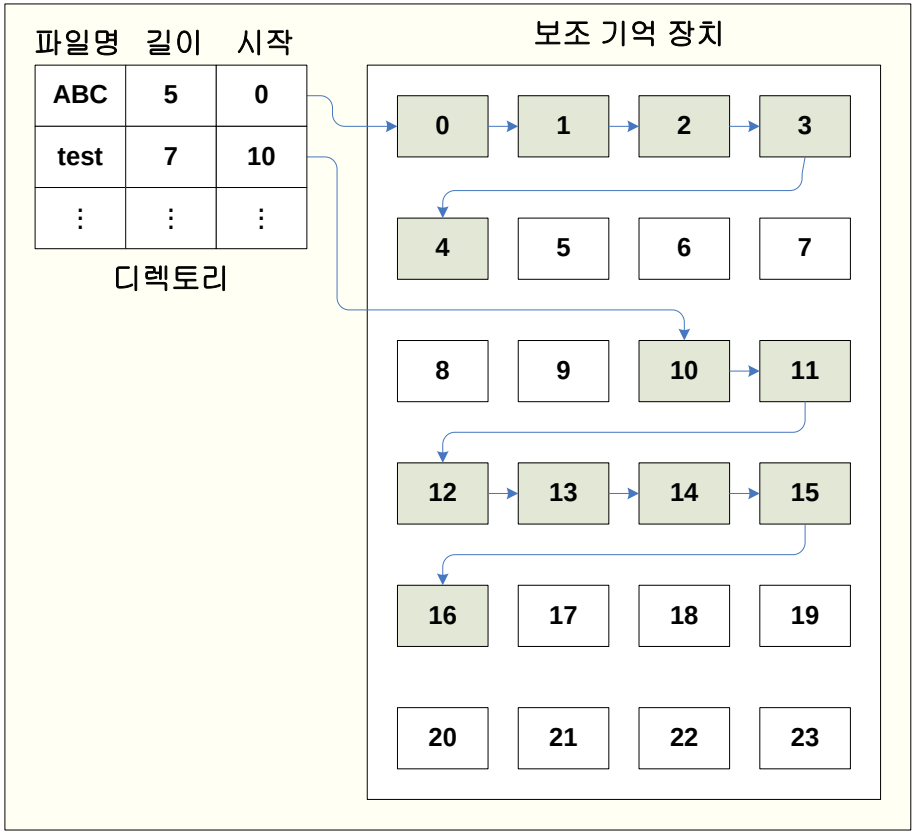
- 가변 분할 다중 프로그래밍에서 메모리 할당 기법과 유사

- 연속 할당(contiguous allocation)
- 불연속 할당(noncontiguous allocation)

6.2.2 빈 공간 할당

(1) 연속 할당(contiguous allocation)

- 파일을 구성하는 레코드들을 인접된 빈 공간에 연속적으로 할당



- 디렉토리 정보
  - 각 파일 이름 / 시작 주소 / 길이

- 불연속 할당 기법보다 빠른 레코드 접근 가능

- 파일 생성과 삭제가 반복되면 단편화**(fragmentation)** 현상 발생
- 빈 공간 합병**(coalescing)** 혹은 집약**(compaction)** 요구

[그림 6.7] 연속 할당(contiguous allocation)

6.2.2 빈 공간 할당

(2) 불연속 할당(noncontiguous allocation)

- 연속 할당에 의한 단편화 현상을 근본적으로 해결
- 파일을 구성하는 레코드들을 보조 기억장치의 임의 공간에 불연속적으로 할당하는 기법

① 연결 리스트(linked list)

② 색인 블록(indexed block)

6.2.2 빈 공간 할당

① 연결 리스트(linked list)를 이용한 방법

- 하나의 파일을 구성하는 레코드들이 보조 기억장치에 연결 리스트(linked list)의 형태로 저장되는 구조

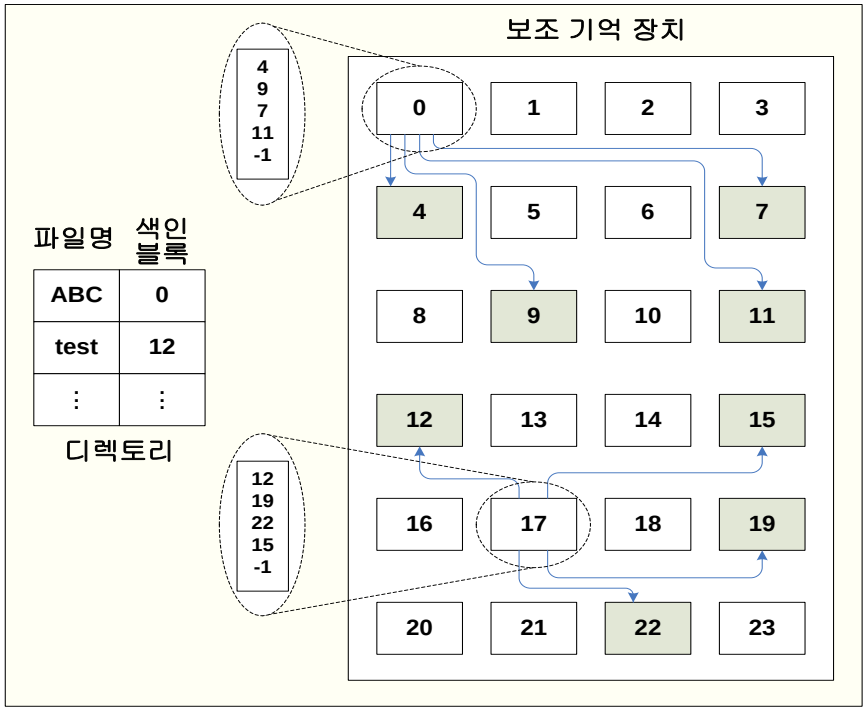
- 디렉토리 정보
  : 파일명, 시작 주소, 끝 주소
- 단순한 디렉토리 구조
- 파일 수정에 의한 레코드 추가/삽입/삭제 용이
- 포인터 연결 구조로 인해 임의의 레코드를 를 접근할 때, 첫 레코드부터 검색해야 함
  → 파일 탐색 시간이 증가
- 포인터 활용으로 인한 저장 공간 감소

[그림 6.8] 연결 리스트를 이용한 불연속 할당

6.2.2 빈 공간 할당

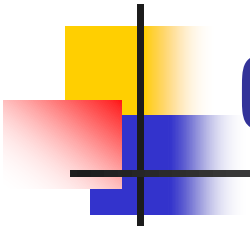
② 색인 블록(indexed block)을 이용한 방법

- 색인 블록을 통해 파일을 구성하는 모든 레코드들에 대한 보조 기억장치의 주소를 가리키는 포인터를 유지
- 하나의 색인 블록이 부족하면 임의 빈 공간을 연결하여 색인으로 사용



- 디렉토리 정보
  : 파일명, 색인 블록 주소
- 디렉토리 구조 간단
- 임의 레코드를 접근하기 위해 보조 기억장치의 주소를 빠르게 탐색 가능
- 색인 블록 공간 낭비
- 파일 수정에 의한 레코드의 삽입 및 삭제 시 색인 블록을 수정해야 함
- 다수의 색인 블록에 대해 포인터 연결

[그림 6.9] 색인 블록을 이용한 불연속 할당



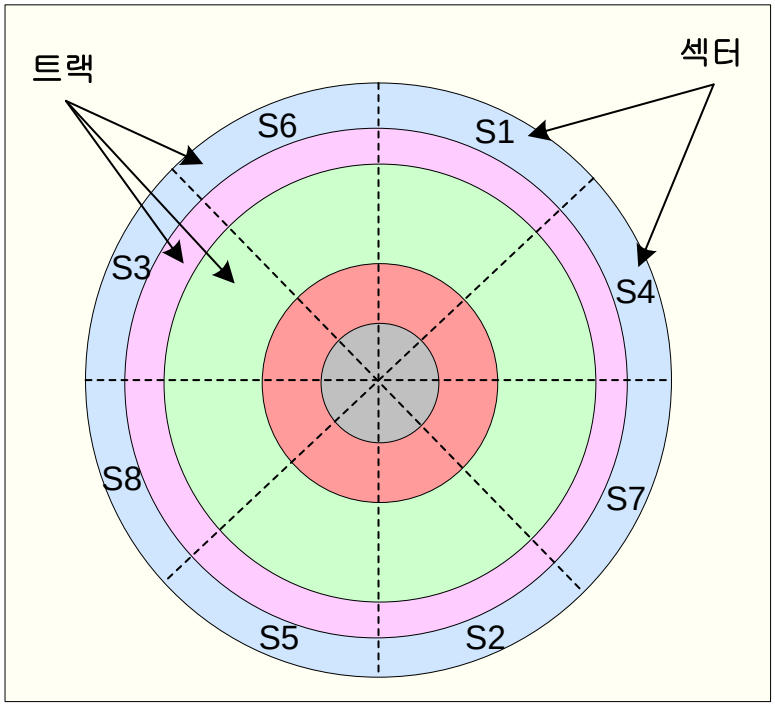
6.3 디스크 관리

- 파일은 디스크 상의 블록 단위로 저장
- 하나의 파일은 여러 개의 블록으로 구성
- 하나의 파일을 참조하기 위해서는 파일을 구성하는 모든 블록에 대한 접근이 요구

- 디스크 접근 속도는 물리적으로 제한
- 시스템 성능 향상을 위한 운영체제의 디스크 관리 기법

6.3.1 디스크 구조 및 접근 시간(access time)

- 디스크: 다수 원판(platter), 헤드(head)
- 헤드에 의해 원판의 표면에 데이터가 저장
- 원판: 트랙(track), 섹터(sector)



[그림 6.10] 디스크의 트랙**(track)**과 섹터**(sector)**

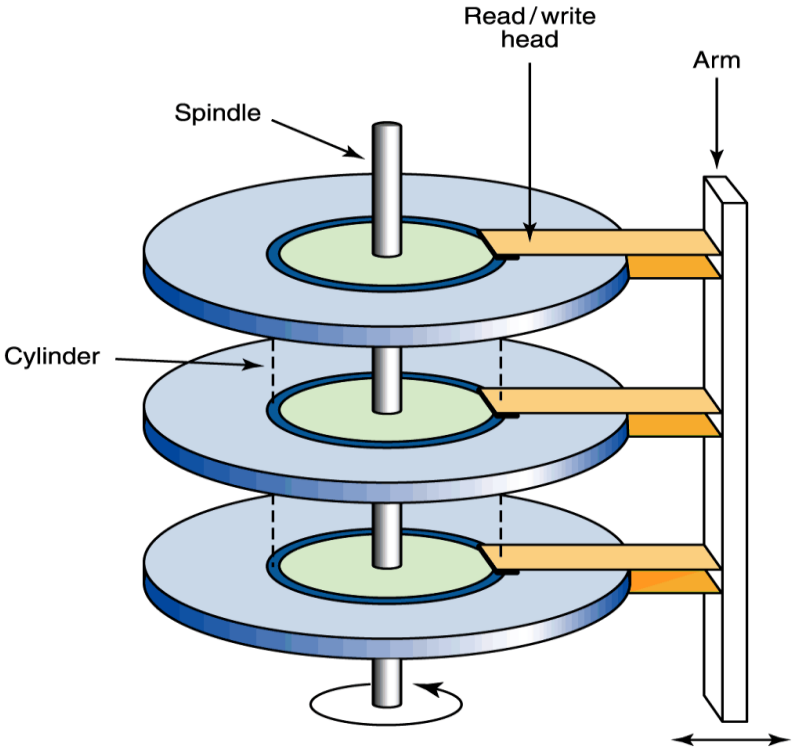
- 트랙 길이는 서로 다름
- 트랙 당 섹터 수는 일정
- 트랙 및 섹터 수는 디스크 특성에 의해 물리적으로 결정

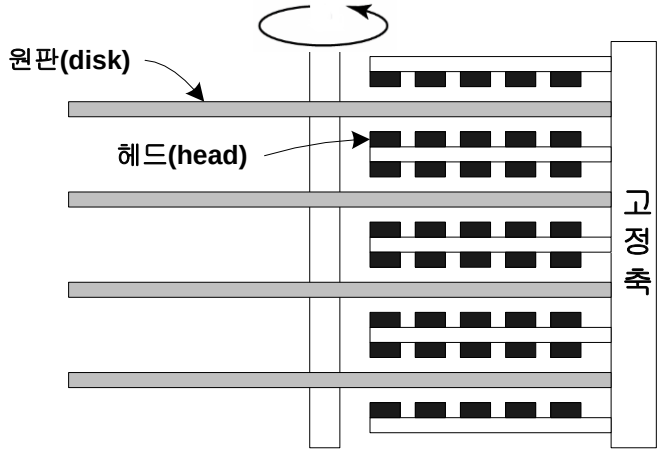
- 하나의 원판 → **500 ~2,000**개 트랙
- 하나의 트랙 → **10 ~ 100**개 섹터
- 섹터**:** 디스크의 데이터 저장 단위
- 섹터 크기**: 32B ~ 4KB(**일반적으로 **512B)**

- 섹터 번호가 불연속**???**
  - 섹터 데이터 읽기 → 전송 → 읽기 → **…**
  - 전송하는 사이 원판은 회전함
- **디스크 인터리빙(disk interleaving)**
  : 한 섹터의 데이터를 전송하는 데 소요되는 시간만큼의 간격을 두고 다음 섹터 번호를 할당하는 것
- **인터리빙 계수(interleaving factor)**
  : 디스크 인터리빙 간격의 정도

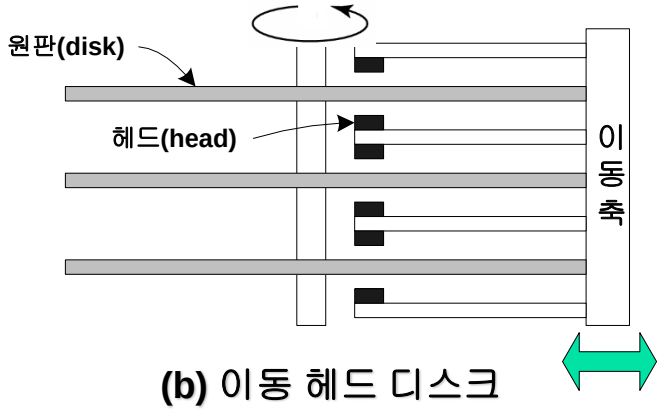
6.3.1 디스크 구조 및 접근 시간(access time)

- 디스크 헤드 유형
  - 고정 헤드(fixed head) vs. 이동 헤드(moving head)





(a) 고정 헤드 디스크



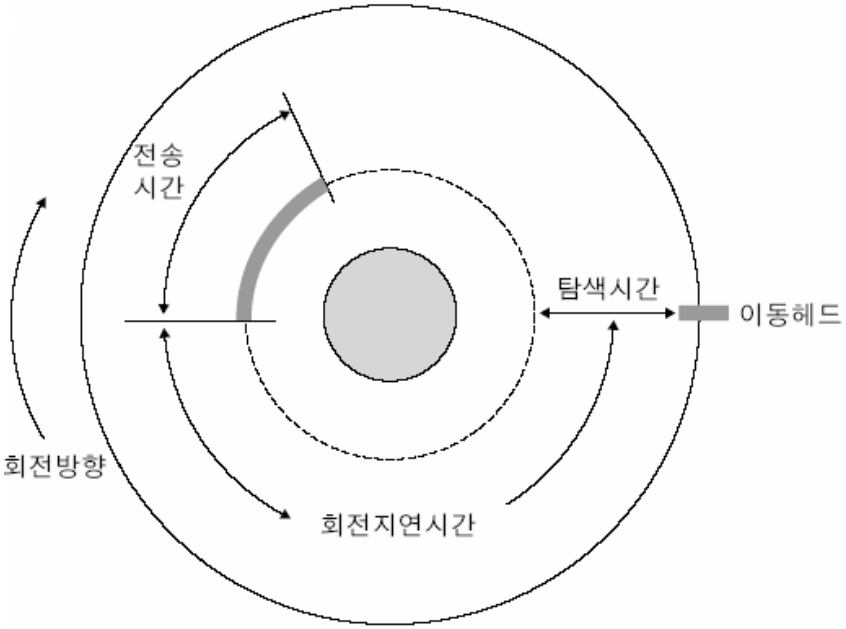
(b) 이동 헤드 디스크

[그림 6.11] 디스크 헤드의 유형

6.3.1 디스크 구조 및 접근 시간(access time)

- 디스크 접근 시간

접근 시간(access time) = 탐색 시간(seek time)
+ 회전 지연시간(latency time)
+ 전송 시간(transmission time)



[그림 6.12] 디스크 접근 시간

- 탐색시간 & 회전 지연시간
  → 디스크 접근 시간을 단축할 수 있는 시간

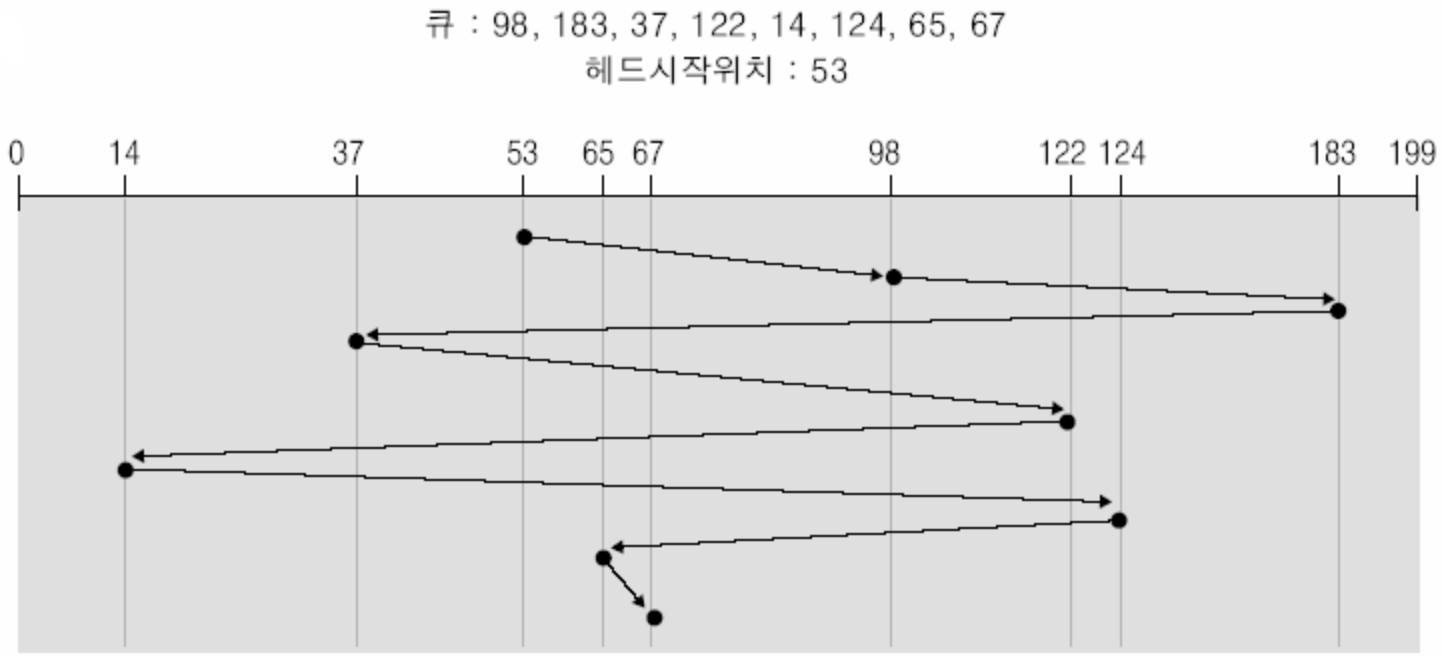
- 디스크 회전 속도: **5,000rpm ~ 10,000rpm**
- 탐색 시간은 회전 지연시간의 **10**배 이상

→ 디스크 스케줄링은 회전 지연시간보다 탐색 시간을 단축하기 위한 기법이며, 이동 헤드 디스크 유형에 적용

6.3.2 디스크 스케줄링

- 디스크 큐(queue): 디스크 입출력 요청 대기
- 디스크 스케줄링: 디스크 큐의 요청 선정 정책

(1) 선입 선처리(FCFS: First-Come First-Served)



•공평성 유지
•탐색 시간 최소화 **X**

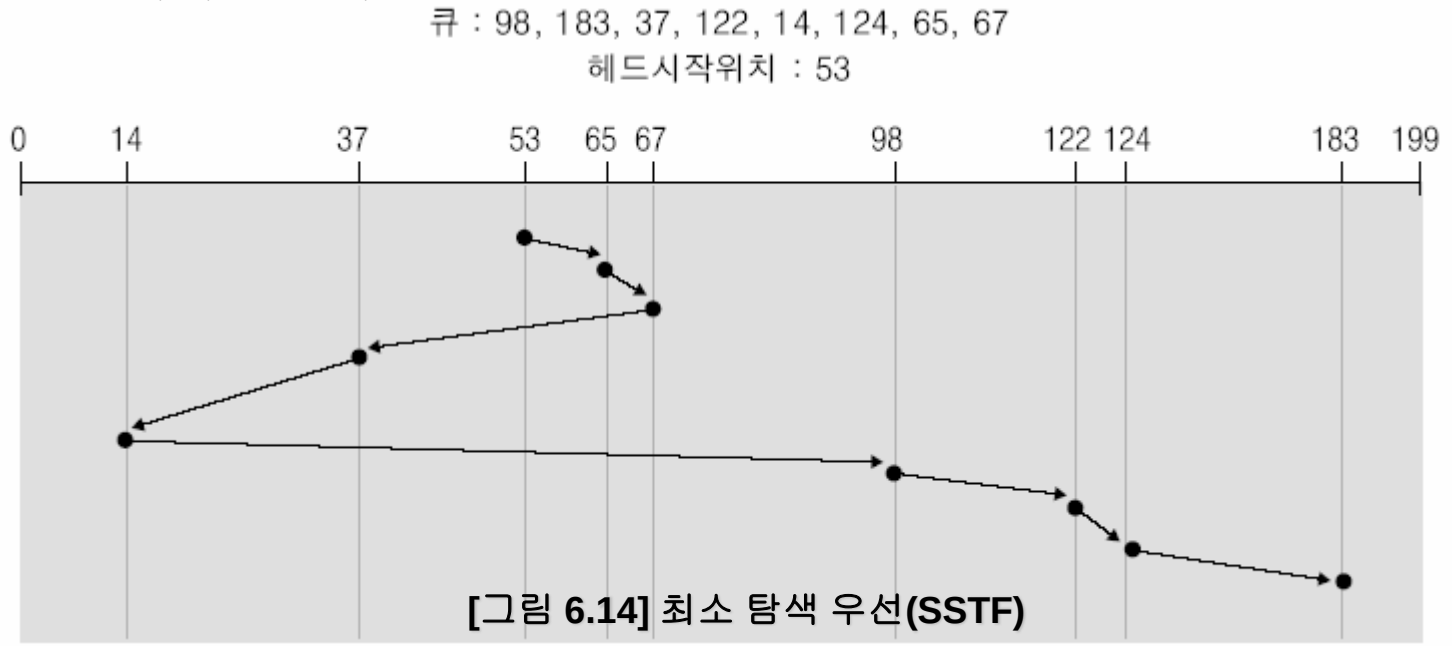
• 디스크 요청↑
→ 탐색시간 ↑↑

[그림 6.13] FCFS

6.3.2 디스크 스케줄링

(2) 최소 탐색 우선(SSTF: Shortest Seek Time First)

- 현재 헤드의 위치에서 가장 가까운 트랙에 해당하는 요청을 우선적으로 처리하는 기법



[그림 6.14] 최소 탐색 우선(SSTF)

- FCFS 보다 탐색 시간 개선
- 현재 헤드 위치에서 가까운 트랙을 검색하는 시간 소요
- 현재 헤드 위치로부터 멀리 떨어져 있는 블록 → 기아현상 발생 가능

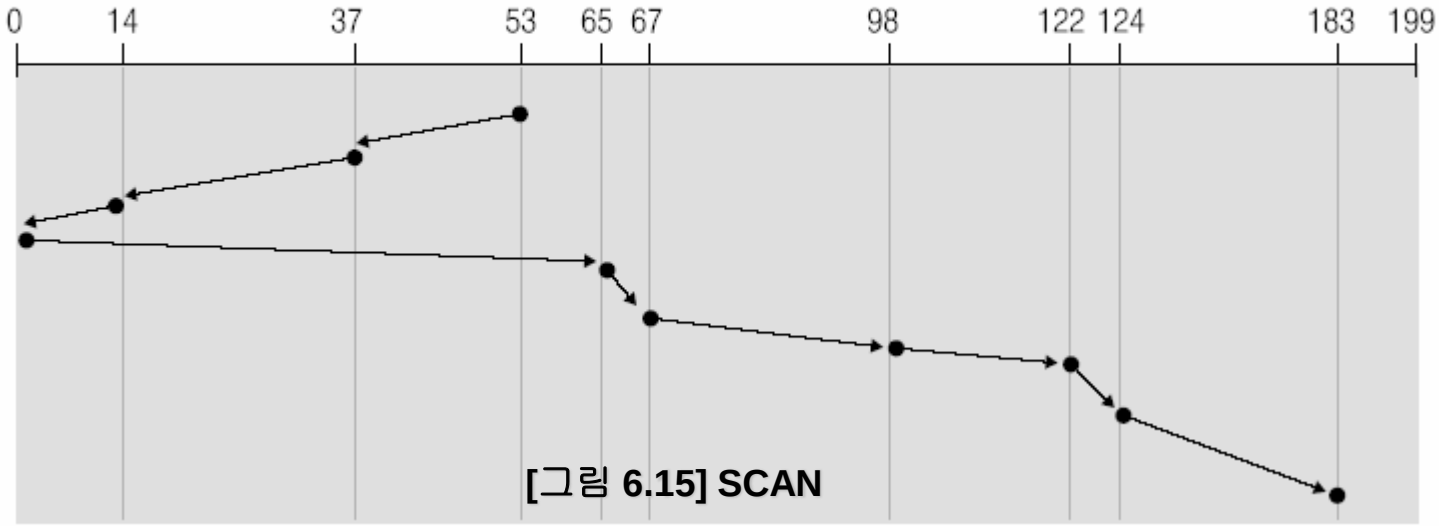
6.3.2 디스크 스케줄링

(3) SCAN 및 C-SCAN (Circular SCAN)

- SCAN
  - 디스크 헤드가 안쪽과 바깥쪽을 왕복하면서 디스크 큐에서 대기중인 블록들을 처리하는 기법, 엘리베이터(elevator) 방식

헤드가 트랙 0 방향으로 이동

큐 : 98, 183, 37, 122, 14, 124, 65, 67
헤드시작위치 : 53



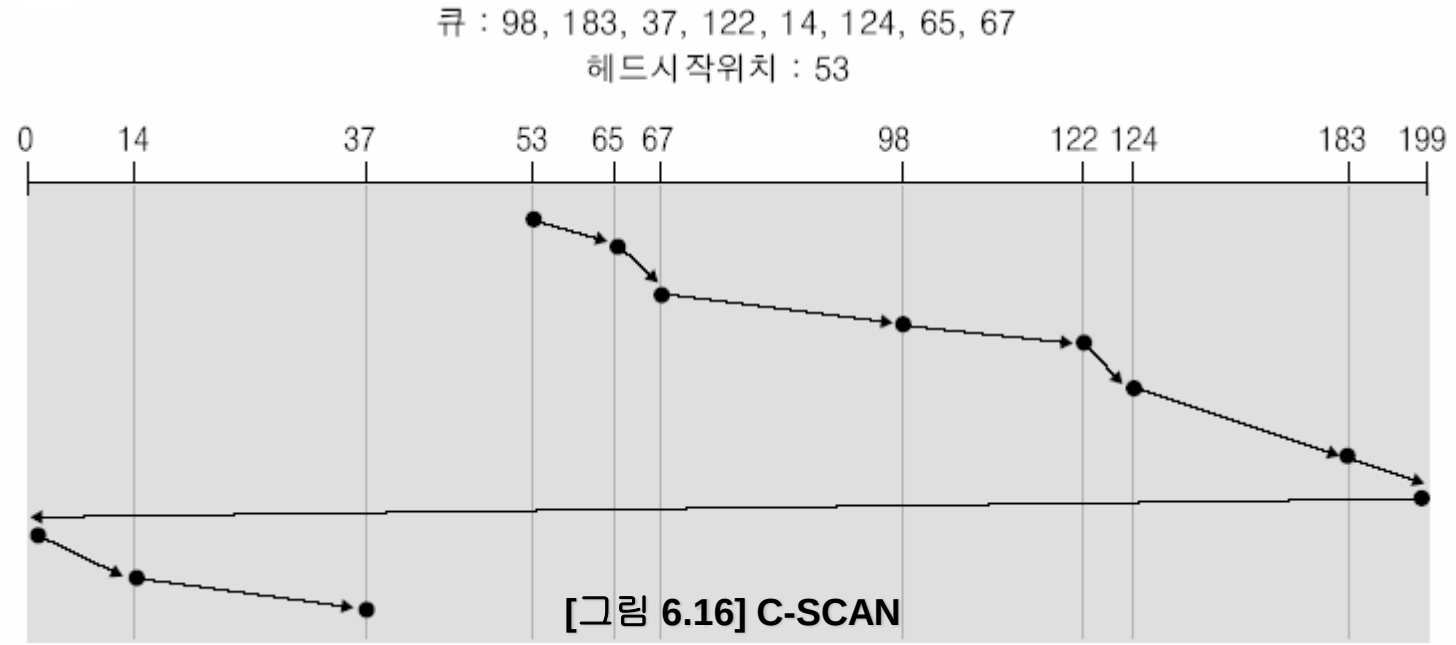
[그림 6.15] SCAN

• 양쪽 끝의 트랙에 위치한 블록은 가운데 트랙에 비해 상대적으로 대기 시간이 길다.

6.3.2 디스크 스케줄링

(3) SCAN 및 C-SCAN (Circular SCAN)

- C-SCAN
  - 균등한 대기 시간을 구현하기 위해 디스크 헤드가 한쪽 방향으로 진행하면서 처리하는 기법



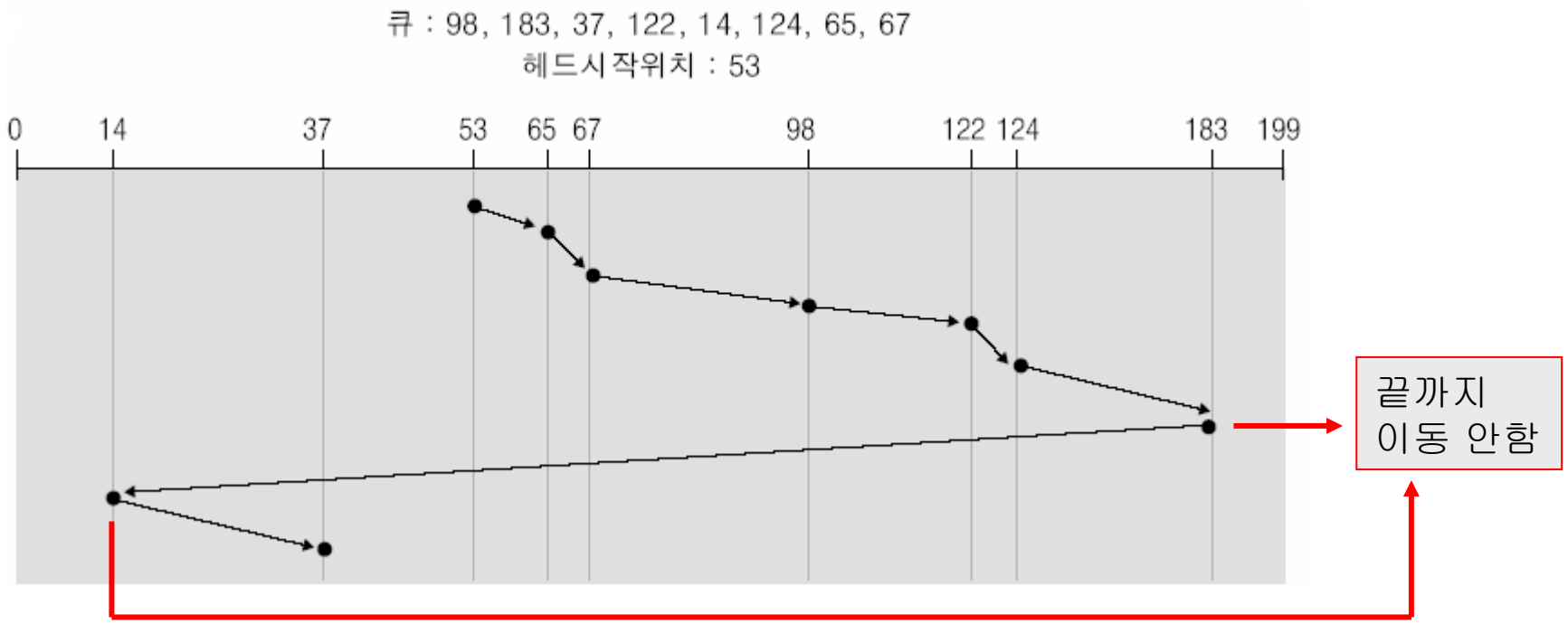
[그림 6.16] C-SCAN

- 디스크 헤드가 한쪽 끝에 다다르면 반대방향으로 진행하면서 처리하는 것이 아니라 헤드를 처음 위치로 이동한 후 다시 처리함

6.3.2 디스크 스케줄링

(4) LOOK 및 C-LOOK (Circular LOOK)

- LOOK: 현 진행방향에서 더 이상 처리할 블록이 없으면 진행방향을 바꿈
- C-LOOK: 현 진행방향에서 더 이상 처리할 블록이 없으면 진행하지 않음

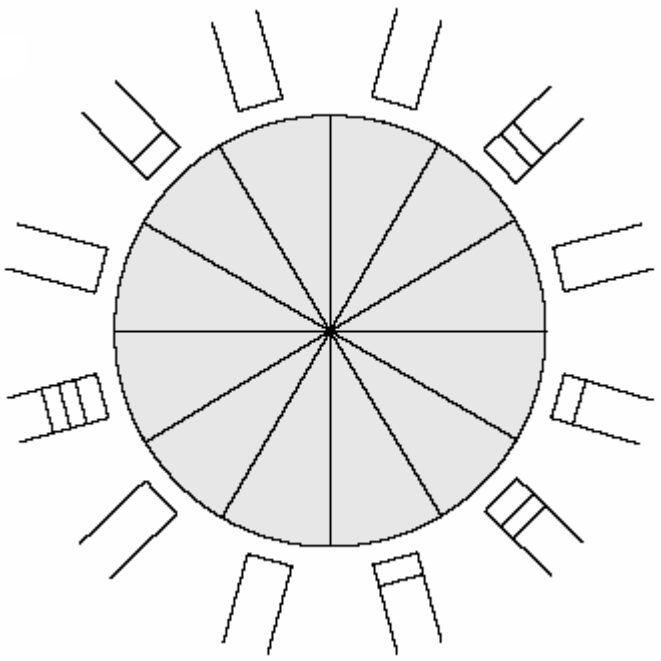


[그림] LOOK 및 C-LOOK (circular LOOK)

6.3.2 디스크 스케줄링

(5) 최소 지연 우선(SLTF: Shortest Latency Time First)

- 디스크 헤드가 고정(헤드 이동 불필요)된 경우 회전 지연 시간을 단축하기 위한 기법



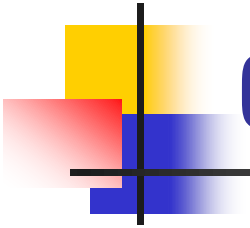
[그림 6.17] 최소 지연 우선(SLTF)

⇒ 섹터 큐잉**(sector queueing)** 알고리즘 사용

6.3.3 디스크 캐쉬 (Disk Cache)

- 디스크 접근 속도의 향상을 위해 사용
- 메인 메모리와 디스크 사이에 위치하는 메인 메모리의 일부분
- 디스크 내용을 블록 단위로 유지
  - 프로세스로부터 디스크의 특정 블록에 대한 입력 요청이 발생
  - 운영체제는 해당 블록의 존재 여부를 디스크 캐쉬에서 확인
    - 존재: 디스크 캐쉬의 내용을 프로세스 주소 공간으로 복사
    - 미존재: 해당 블록을 디스크로부터 디스크 캐쉬로 복사한 후, 디스크 캐쉬의 내용을 프로세스 주소 공간으로 복사

- 디스크 캐쉬의 성공률(hit ratio)
  - 디스크 접근 속도 향상 → 성공률 극대화
  - 요구 혹은 예상 페이징 기법(4장)에서 페이지 부재율 최소화 전략과 동일

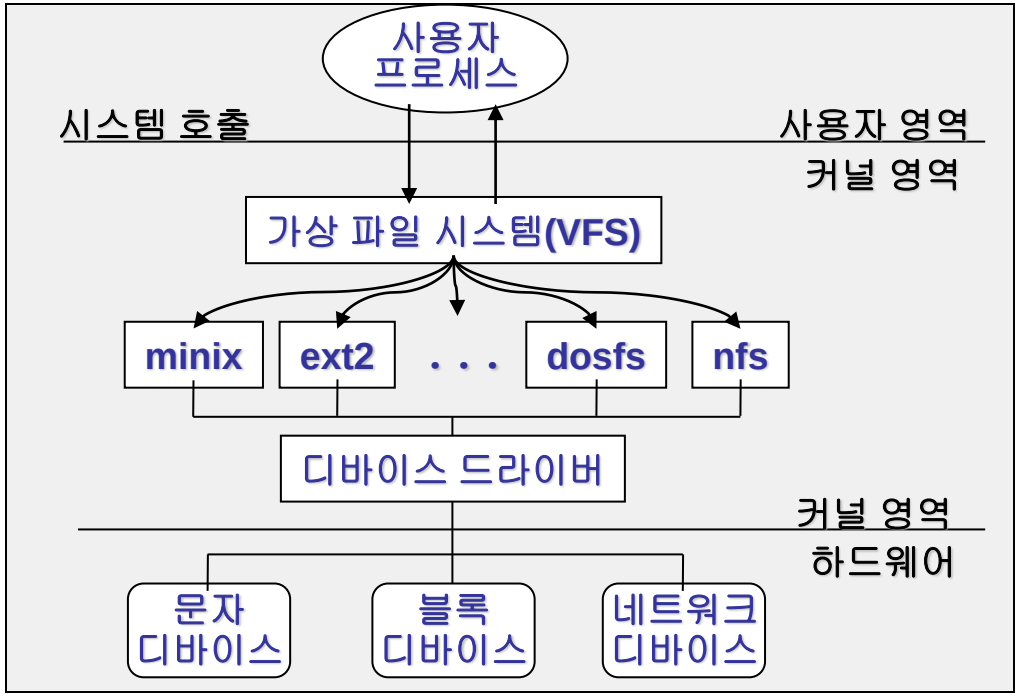


6.4 리눅스 파일 시스템

- 리눅스 가상 파일 시스템
- MS-DOS
- Ext2

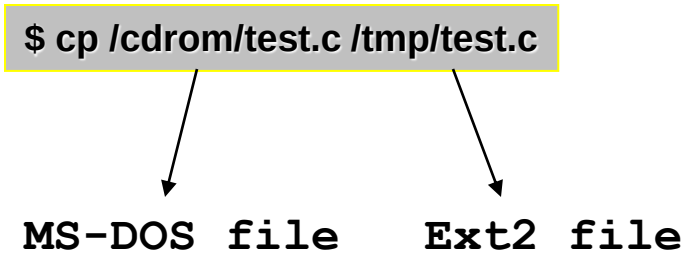
6.4.1 가상 파일 시스템

- 특정 파일 시스템들의 상위 계층에 위치
- 각각의 파일 시스템을 추상화함
- 사용자 투명성 제공
  - 사용자 프로세스의 파일 접근 시스템 호출 → 가상 파일 시스템 처리 → 특정 파일 시스템에 해당되는 루틴을 호출



[그림 6.18] 가상 파일 시스템

- 리눅스 커널의 기본 파일 시스템 → **Ext2**
- 리눅스에서는 **VFS**를 통해 다양한 운영체제 **(MS-DOS, UNIX, MINX, Windows NT)**에서 사용하는 파일 시스템을 지원



6.4.2 MS-DOS 파일 시스템

- MS-DOS 파일 시스템 구조
  - 부트 섹터 (boot sector)
  - FAT (File Allocation Table)
  - 루트 디렉토리
  - 데이터 영역

[그림 6.19] MS-DOS 파일 시스템 구조

6.4.2 MS-DOS 파일 시스템

(1) 부트 섹터(boot sector)

- 시스템이 부팅될 때 MS-DOS 운영체제를 디스크로부터 메모리에 적재시켜주는 부트스트랩 루틴(bootstrap routine)을 포함
- 부트스트랩 루틴: ROM-BIOS에 의해 적재된 후 실행
- 부트 섹터 정보는 디스크가 포맷될 때 항상 디스크의 첫 번째 섹터에 존재

- 부트 섹터 다음에는 MS-DOS에서 예약한 섹터들이 존재
  - 부트스트랩 루틴을 하나의 섹터에 저장할 수 없는 경우 예약해 놓은 섹터들을 연결하여 사용함
  - 부트 섹터의 오프셋 0Eh에 정보를 기록
  - 부트 섹터가 하나의 섹터에 모두 저장될 경우에는 0Eh에 1을 기록

6.4.2 MS-DOS 파일 시스템

<표 6.1> 부트 섹터의 구조

offset	info	Length(byte)
00h	부트스트랩 루틴으로 jump하는 명령어	3
03h	DOS 버전과 생산자 이름	8
0Bh	섹터 당 바이트 수	2
0Dh	클러스터 당 섹터 수	1
0Eh	DOS에서 예약된 섹터 수	2
10h	FAT의 개수	1
11h	루트 디렉토리의 엔트리 수	2
13h	전체 섹터 수	2
15h	디스크 타입에 대한 정보	1
16h	FAT에 사용되는 섹터 수	2
18h	트랙 당 섹터 수	2
1Ah	R/W 헤드의 개수	2
1Ch	숨겨진 섹터의 수	2
1Eh	부트스트랩 루틴의 시작 주소	?

6.4.2 MS-DOS 파일 시스템

(2) FAT (File Allocation Table)

- 새로운 파일 생성 또는 기존 파일 확장을 위한 디스크 공간 관리
- FAT 섹터는 MS-DOS 예약 섹터 다음에 위치
- 디스크에 존재하는 FAT의 수는 부트 섹터의 offset 10h에 기록
- 클러스터
  - 디스크 공간의 효율적인 관리를 위해 연속된 섹터들을 하나의 클러스터로 정의하여 디스크를 사용하는 최소 단위로 사용
  - 부트 섹터 offset 0Dh에 기록
- FAT 엔트리(entry)
  - 클러스터(cluster)와 1:1 관계

<표 6.2> FAT 엔트리 내용과 의미

내용	의미
0000h	사용되고 있지 않은 클러스터
FFF0h ~ FFF6h	**MS-DOS**에 의해 예약된 클러스터
FFF7h	고장난 클러스터
FFF8h ~ FFFFh	파일에 할당된 마지막 클러스터
xxxxh	파일에 할당된 다음 번째 클러스터 번호

6.4.2 MS-DOS 파일 시스템

(3) 루트 디렉토리(root directory)

- FAT 다음 섹터에 위치
- 루트 디렉토리 엔트리 수는 부트 섹터의 offset 11h에 기록
- 디렉토리 엔트리 크기: 32bytes

<표 6.3> 디렉토리 엔트리의 구조

offset	info	length(bytes)
00h	파일 이름	8
08h	확장자 이름	3
0Bh	파일 속성(attribute)	1
0Ch	reserved	10
16h	변경된 시간	2
18h	변경된 날짜	2
1Ah	파일이 저장된 첫 번째 클러스터 번호	2
1Ch	파일 크기	4

파일명→8자 제한

확장자→3자 제한

6.4.2 MS-DOS 파일 시스템

- 파일 속성은 1바이트로 표시
- 파일에 대한 속성은 각 비트들의 조합으로 표현

<표 6.4> 속성 바이트의 구조

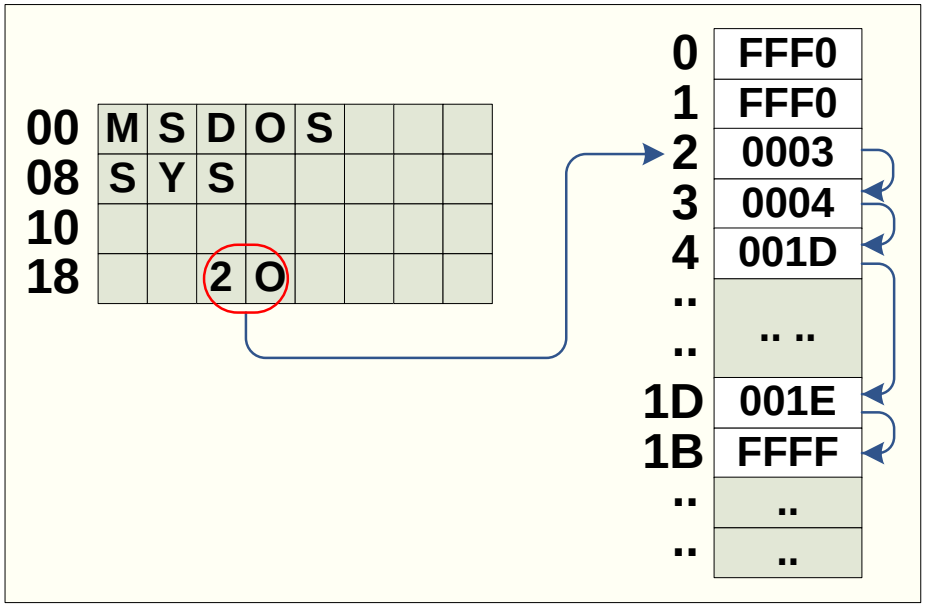
비트	의미
0	1 = 읽기 전용(read only)
	0 = 읽기/쓰기(read/write)
1	1 = 파일명 숨기기(hidden file)
2	1 = 시스템 파일(system file)
3	1 = 볼륨 이름(volume name)
4	1 = 디렉토리(directory)
5	아카이브 비트(archive bit)
6-7	reserved

백업에 의해 파일이 저장된 경우 1이라는 값을 유지

6.4.2 MS-DOS 파일 시스템

- 사용자 프로그램 혹은 데이터와 서브 디렉토리를 저장
- 예) MS-DOS 파일 시스템에 "/MSDOS.SYS" 파일이 클러스터 번호 2부터 저장된 경우 FAT와 루트 디렉토리의 관계

MSDOS.SYS → 부트섹터 체크 및 부팅에 직접적인 역할을 수행 (기본 주변드라이브 제어, 부팅시퀀스 및 프로세싱 등등)



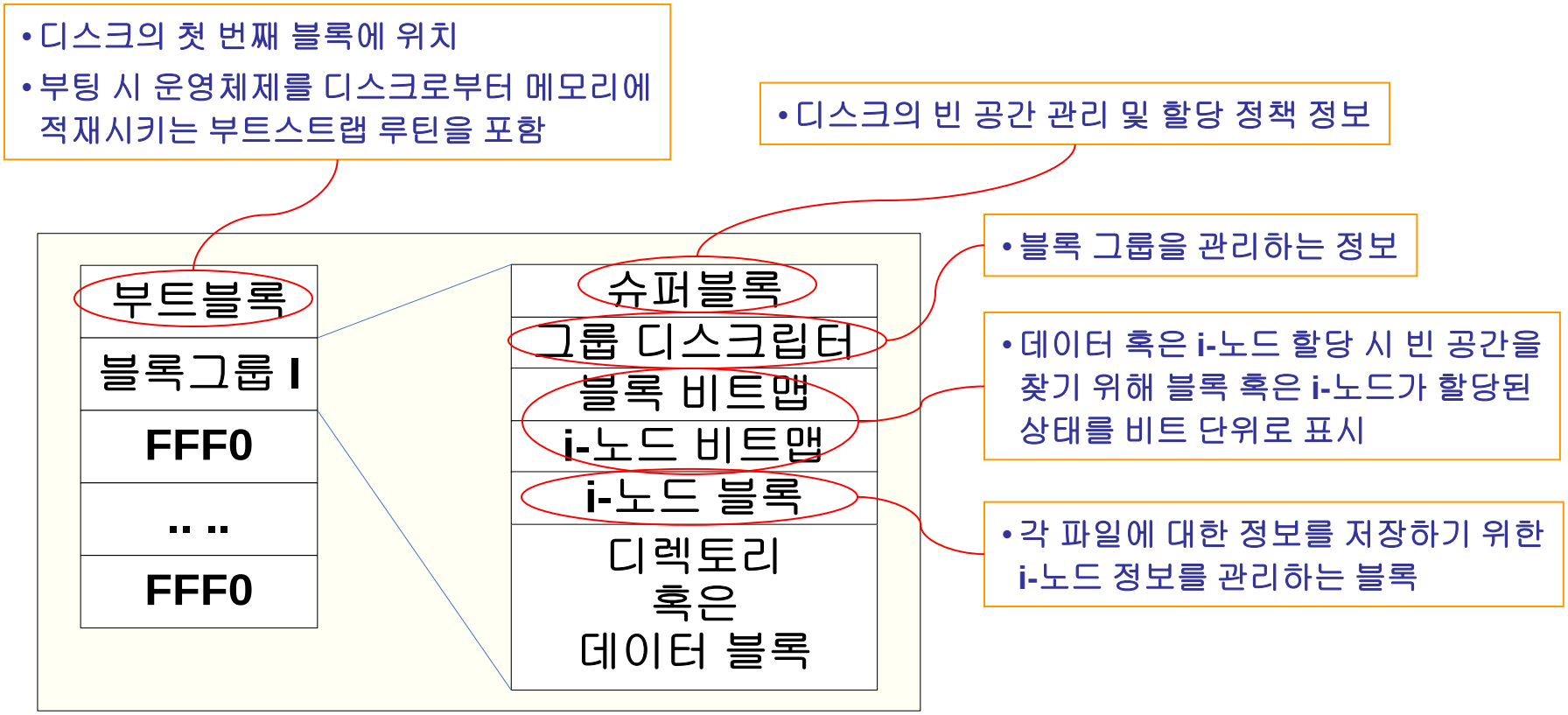
①루트 디렉토리를 검색하여 파일명 확인
②파일명에 해당하는 첫번째 클러스터(02) 검색
③FAT 엔트리(02)로부터 다음 클러스터의 번호 (03)을 검색
④동일한 방법으로 마지막 클러스터 번호(1B)까지 검색

- 새로운 파일 생성 시 FAT를 검색하여 빈 클러스터를 할당하고, 다음 클러스터 번호를 연결
- 기존 파일 삭제 시 디렉토리 엔트리의 파일명의 첫 번째 바이트를 "E5h"로 변경(삭제파일 복구가능)

[그림 6.20] FAT와 디렉토리 관계

6.4.3 Ext2 파일 시스템

- Ext2(Second Extended): 리눅스 기본 파일 시스템



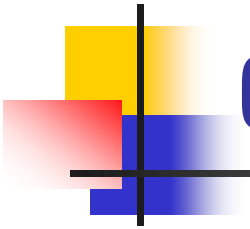
[그림 6.21] Ext2 파일 시스템 구조

6.4.3 Ext2 파일 시스템

(2) 슈퍼 블록(super block)

- 디스크 빈공간 관리와 할당 정책 관련 정보
  - ①총 i-노드 수 및 사용 가능한 i-노드 수
  - ②총 블록 수 및 사용 가능한 블록 리스트
  - ③블록 크기
  - ④그룹당 i-노드 및 블록 수
  - ⑤파일 시스템 상태
  - ⑥생성한 운영체제
  - ⑦파일 시스템이 마지막으로 갱신된 날짜 및 시간

  - 파일 생성/삭제 과정에서 파일 시스템에 의해 갱신
  - 새로운 파일 생성시 사용 가능한 디스크 블록을 빠르게 검색하기 위해 슈퍼 블록의 내용은 부팅 시 메모리에 복사하여 사용
  - 슈퍼 블록의 내용이 변경되면 그 내용과 갱신된 날짜 및 시간을 주기적으로 디스크에 복사하여 디스크의 내용과 일치시킴



6.4.3 Ext2 파일 시스템

(3) 블록 그룹 디스크립터

- 각 블록 그룹을 관리하는 정보
  - 블록 비트 맵 블록 번호
  - i-노드 비트 맵 블록 번호
  - i-노드 테이블 블록 번호
  - 빈 블록 수
  - 빈 i-노드 수
  - 디렉토리 수

6.4.3 Ext2 파일 시스템

(4) 블록 혹은 i-노드 비트 맵

- 데이터 혹은 i-노드를 할당할 때 빈 공간을 찾기 위해 블록 혹은 i-노드가 할당된 상태를 비트 단위로 표시
  - 비트 → 0: 블록 혹은 i-노드가 할당되어 있지 않음
  - 비트 → 1: 블록 혹은 i-노드가 할당되어 사용중임

  - 블록 혹은 i-노드 비트 맵은 각각 한 개의 블록을 사용하므로 블록 크기에 따라 표시할 수 있는 블록 혹은 i-노드의 개수가 결정됨

6.4.3 Ext2 파일 시스템

(5) i-노드 블록

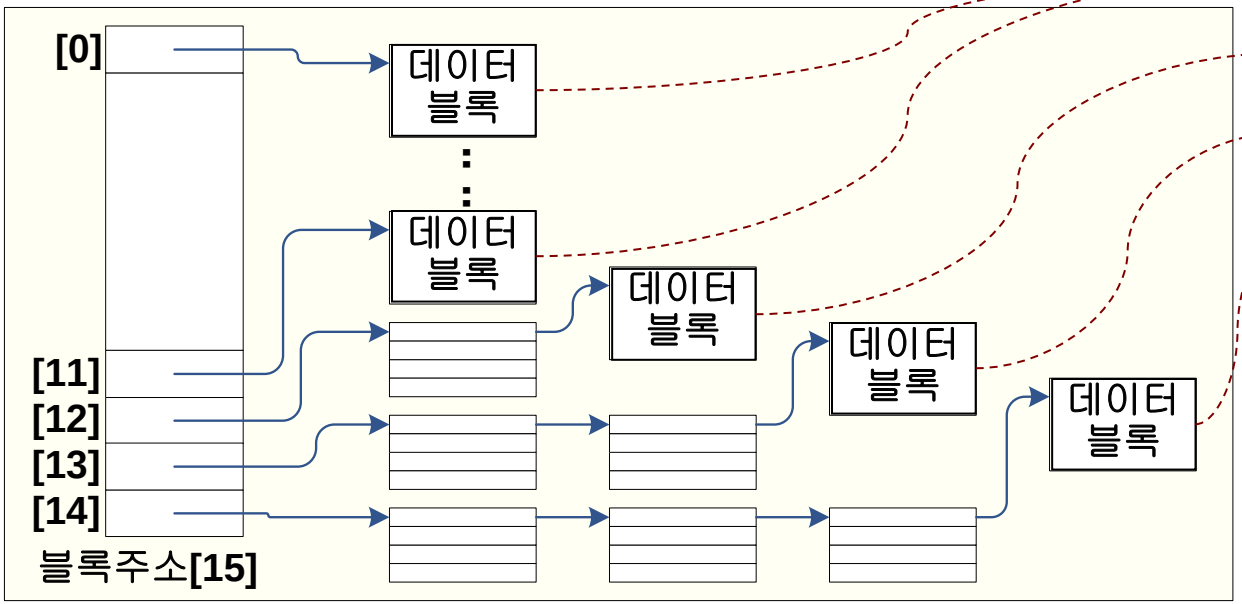
- i-노드
  - 각 파일에 대한 정보를 저장하기 위한 자료구조
  - 각 파일은 하나의 i-노드를 가짐
  - i-노드마다 고유의 번호가 할당
- i-노드 블록
  - 여러 개의 블록으로 구성되며, 아래 정보를 가짐

① 파일 종류(일반 파일, 디렉토리 파일, 장치 파일 등)
② 파일의 링크 수
③ 파일 소유자(uid, gid)
④ 파일의 크기
⑤ 파일이 저장된 디스크 주소(13바이트)
⑥ 접근 모드 및 시간

6.4.3 Ext2 파일 시스템

(5) i-노드 블록(계속)

- 여러 i-노드들은 디스크에 선형 배열(linear array)로 저장
- i-노드 정보 동기화
  - disk i-node vs. in-core i-node
  - 커널 내부의 내용을 디스크에 반영 필요
- 블록 할당



- 직접 참조(0~11)
- 간접 참조(12)
- 이중 간접 참조(13)
- 삼중 간접 참조(14)
  : 세 번의 디스크 블록 검색 필요

- i-노드의 크기를 작게 유지하며 매우 큰 크기의 파일 지원

[그림 6.22] 데이터 블록 할당

•블록크기: 1K, 블록주소: 4B → 블록 당 256개 주소 지정 가능
: (12 + 256 + 256*256 + 256*256*256)*1KB → 약 16GB 파일 지원

6.4.3 Ext2 파일 시스템

(6) 디렉토리/데이터 블록

- i-노드의 파일 종류를 통해 디렉토리와 파일을 구분

변위	i-node 번호 (2바이트)	파일명 (14바이트)
0	1	.
16	1	..
32	12	etc
48	13	bin
64	14	dev

• 루트 디렉토리
(i-노드 → 1: 루트 디렉토리 파일)

[그림 6.23] 디렉토리 구조

• /etc/passwd 파일 접근 과정

① 루트 디렉토리에서 "etc" 파일명에 해당하는 i-노드 번호(12) 획득
② i-노드 번호(12)에서 "etc" 디렉토리가 저장된 디스크 주소 획득
③ 디스크 주소에서 "passwd" 파일명에 해당하는 i-노드 번호 획득
④ 해당 i-노드로부터 "passwd" 파일이 저장된 디스크 주소 획득