6. Operation System (File System)

[운영체제 OS] 파일 시스템 File System — MYVELOP 마이벨롭 (tistory.com)

https://eunajung01.tistory.com/91

---

1. 파일과 파일 시스템

1. 파일

⇒ 파일(File)은 저장 단위이며, 관련된 정보 자료들의 집합에 이름을 붙인 것이다.

- 보조 저장 장치에 저장되어 있는 관련 정보의 집합
- 1개 파일은 디스크 내 여러 개의 섹터로 구성 → 파일 = 섹터 단위로 저장되는 정보의 집합체
- 메모리와 디스크 간의 입출력 전송은 **블록** 단위로 수행되며, 각 블록은 하나 이상의 섹터

⇒ 일반적으로 **레코드(Record) 혹은 블록(Block) 단위**로 비휘발성 보조기억장치에 저장

⇒ 운영체제는 다양한 저장 장치를 file이라는 동일한 논리적 단위로 처리 (mapping)

- 저장 장치의 물리적 특성을 추상화 → 컴퓨터 시스템의 편리한 사용 지향

⇒ OS는 create, read, write, reposition (lseek), delete, open, close 등의 시스템 콜 연산 제공

※ 파일 속성(File attribute), 파일의 메타데이터(metadata), FCB

⇒ 파일 자체의 내용이 아니라 파일을 관리하기 위한 각종 정보

⇒ 파일 타입 : 일반 파일 (기본적인 파일들) / 특수 파일 (디렉토리, 장치 파일, 링크, 소켓 등)

⇒ 파일 이름, 유형, 저장된 위치, 파일 사이즈, 접근 권한 (읽기 / 쓰기 / 실행), 소유자, 시간 (생성 / 변경 / 사용) 등의 파일에 대한 전반적인 정보

⇒ 이러한 속성들은 파일의 FCB, file control block에 저장된다.

⇒ 가상 메모리 시스템은 **파일 추상화**를 기반으로 동작, **추상화의 실현을 위하여 파일 시스템이 존재**

⇒ exe 파일을 구성하는 블록들이 프로세스 page로 맵핑 → 하나의 프로세스로 나타나도록 추상화

⇒ 모든 입출력 장치들은 **장치 파일**이라는 개념으로 일관성 있게 추상화

- 하드웨어 장치 역시 동일한 연산으로 조작 가능하다.

※ File 접근 제어 (파일 공유 수준)

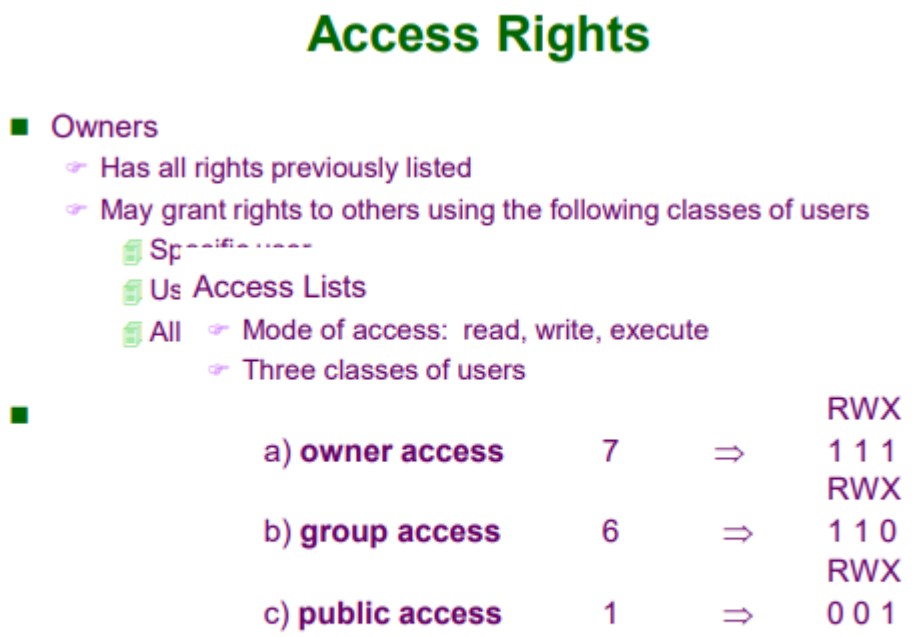
1. 파일 공유에 대한 대처, 동시 엑세스 문제
    - Lock 메커니즘 : 누군가 파일을 사용하는 동안 Lock을 걸어, 다른 사람의 접근을 막는다.
        - 파일 전체, 또는 파일의 일부인 Record 등의 부분에만 걸 수도 있다.
    - 또는, OS는 아무것도 안할 수 있음 (관리자 재량에 맡김)

2. 파일에 대한 접근 권한 수준
    a. **None** : lowerst level, 접근 권한이 전혀 없음
    b. **Knowledge** : 읽을 수 있는 권한, 직접 열지는 못하나 디렉토리 수준에서 확인 가능 (Read)
    c. **Execution** : 프로그램을 load하고 실행시킬 수 있지만, Copy는 못한다. (Execution, Read)
    d. **Reading** : 실행 및 Copying 포함한 Reading 작업을 허용한다. (Execution, Read)
    e. **Appending** : 기존 파일 끝에 새로운 파일을 추가함으로 덧붙이기가 가능하다. (Write)
    f. **Updating** : 파일 데이터의 삭제, 갱신, 추가가 가능하다. (Write)
    g. **Changing protection** : 다른 사용자들에게 권한을 부여할 수 있다. (Write)

h. **Deletion** : 사용자는 파일 자체를 삭제 가능하다. (Write)

⇒ 단, 실제로는 1번~8번을 간략화, 아래 약식방법을 사용!

3. **Access List, Grouping (Owner / Group / Public)**

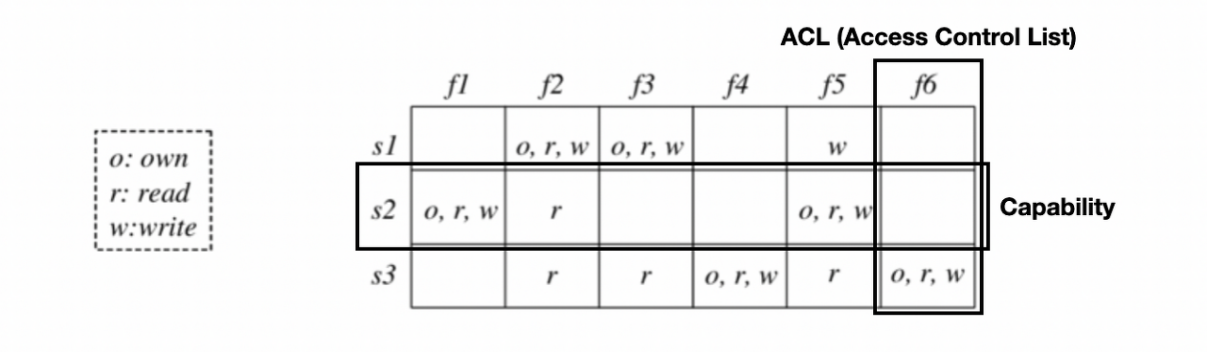


⇒ Owner : 가장 권한이 많은 사용자, 파일 소유자 (Read & Write & Execution)

⇒ Group : Owner 보다는 낮지만, 중간 등급의 권한을 가진 사람 (Read & Write)

⇒ Public : 다른 보통 사람들 (Read)

3. **Access Control Matrix (Read / Write / Execution)**

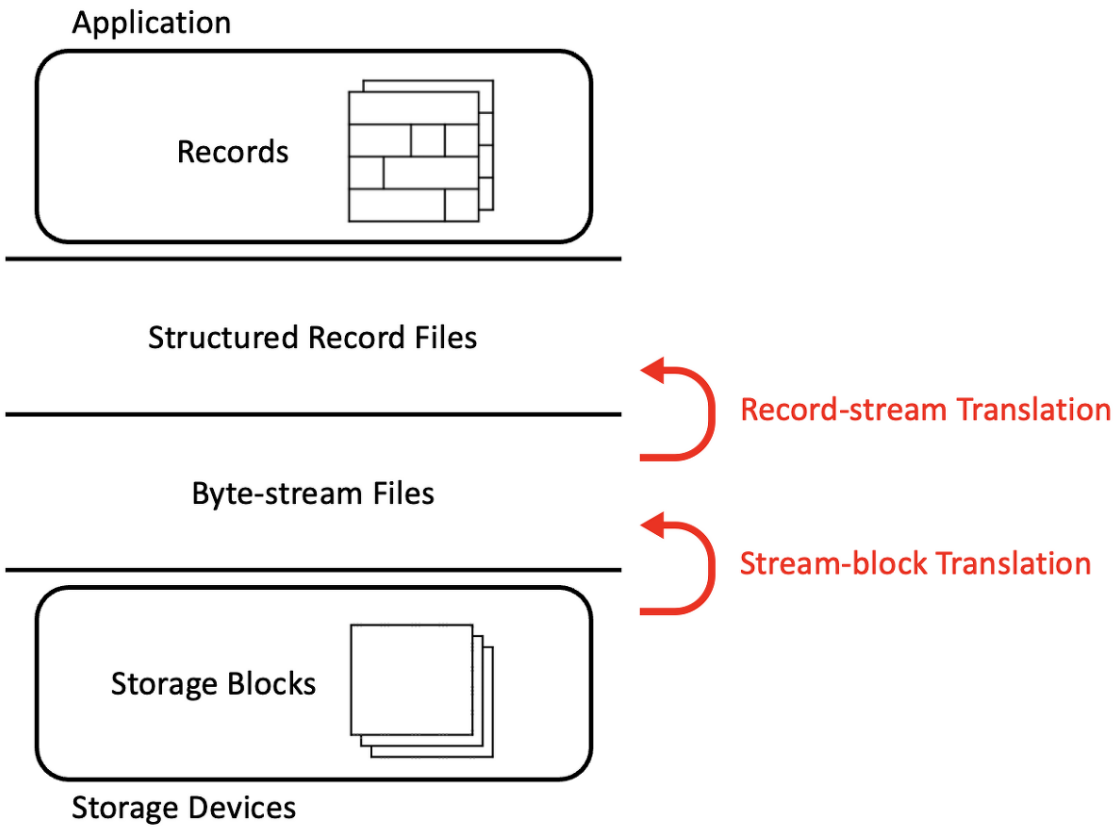


⇒ 사용자와 파일 이름을 매칭 →사용자가 특정 파일에 어떤 접근 권한을 가지는지 표시하는 방법

a. **Access Control List** : 파일 별로 누구에게 어떠한 접근 권한이 있는지, 파일 주체의 linked-list를 이용해 표시
b. **Capability** : 사용자 주체의 linked-list를 만들어 자신이 접근 권한을 가진 파일 및 해당 권한 표시

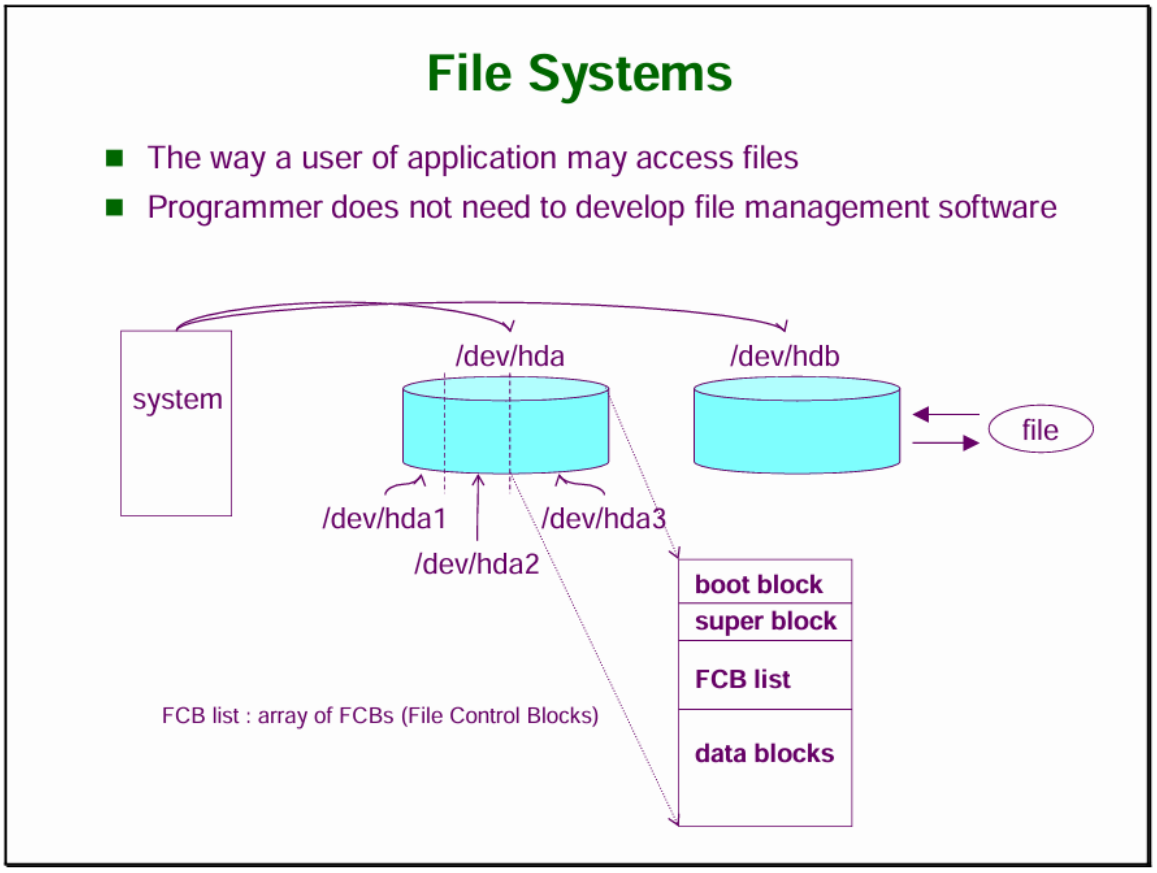
4. **Password**
    - 파일이나 디렉토리마다 password를 두는 방법
    - 모든 접근 권한에 대해 하나의 password: all-or-nothing
    - 접근 권한별 password: 암기 문제, 관리 문제

2. 파일 시스템



⇒ 운영체제와 모든 데이터, 프로그램의 저장과 접근을 위한 기법을 제공

- 디스크를 보다 효율적이고 편리하게 사용
- 저장 장치는 선형적인 주소를 가진 바이트의 블록만을 저장
- 파일 시스템은 **저장 장치와 응용 프로그램 간의 자료구조 차이점을 해결**



⇒ 시스템 내의 모든 파일에 관한 정보를 제공하는 계층적 디렉터리 구조

- 거대한 자료 구조, 디스크와 같은 보조 기억장치에 생성됨
- boot block + super block + FCB list + 데이터 블록 등
- 데이터 스트럭쳐 + 처리 알고리즘까지 포괄하는 개념

⇒ 파일 및 파일의 메타데이터, 디렉터리 정보 등을 관리

⇒ 파일의 저장 방법을 결정하고 파일을 보호

- 저장 장치는 **파일을 블록들로 나누어 섹터에 저장**

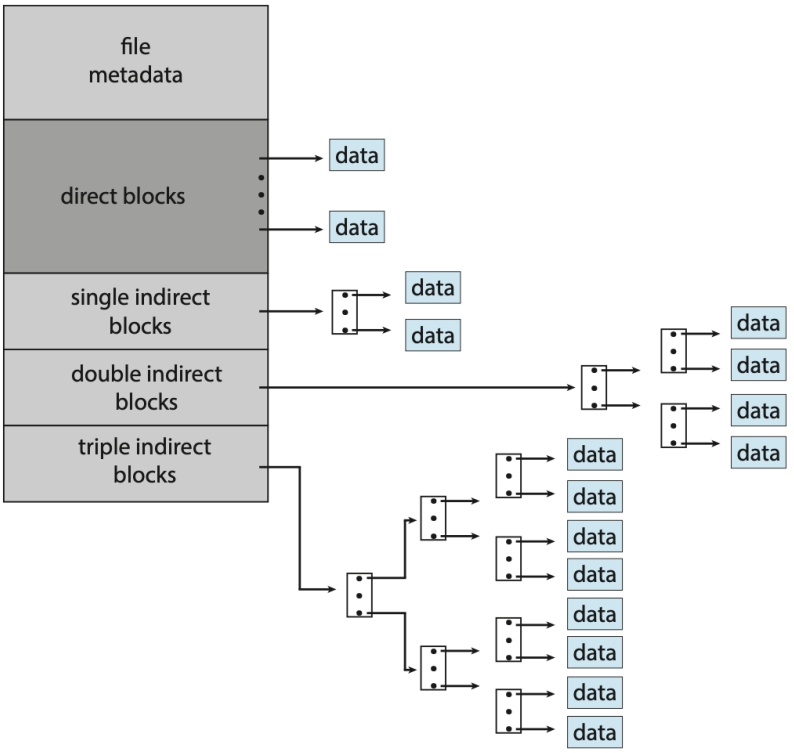
1. **Boot block**
   - 부팅 작업의 시작이 되는 부트 코드가 들어있는 블록, 디스크 가장 앞쪽에 위치
   - 부트 미디어로 쓰이는 저장 장치의 특정 블록에 저장 (SSD, 하드디스크)
   - 부팅 → BIOS가 하드웨어 체크 → 해당 블럭 내 부트코드 실행 → 코드는 OS 가져와 M에 로드

- 부팅(booting) : 컴퓨터 전원을 넣은 후 사람이 컴퓨터를 사용할 수 있도록 하는 준비 과정
- OS 등의 SW가 하드디스크에 저장, 이를 읽어 메인 메모리로 옮기고 실행시키는 작업을 의미

2. **Super block**
    - 중요하고 강력한 권한을 가진 정보 (파일 시스템 블럭 개수, 데이터 블럭 및 FCB 리스트 관련 정보 등)을 관리하는 블럭
    - 파일시스템에 대한 제어 정보를 가진 헤더에 해당
    - 수퍼블록 뒤 → FCB 리스트와 데이터 블록들의 집합, 해당 부분이 실제 파일을 구성하는 정보들로 모여 있는 부분
    - 슈퍼블록은 파일들 전체에 대한 정보, 파일 시스템에 대한 관리 정보를 가지고 있는 것

3. **FCB List** : 파일과 관련된 속성들의 집합인 FCB를 List 형태로 설정한 자료구조
    - 파일 이름 제외, 모든 메타데이터 저장 및 관리
    - Directory가 각 파일 이름만 관리, 나머지는 FCB List로써 처리함
        - direct blocks : 해당 파일에 대한 data block 위치
        - single & double & third indirect : 파일 크기가 너무 커서, 모든 direct 블럭에 접근 불가!
            - 블럭들을 분할하여, 중간에 거쳐갈 수 있도록 하는 자료구조를 담는 영역

※ (Direct & Single / Double / Triple indirect blocks)



⇒ 처음 Direct Block이 비어있다면, 해당 부분으로 직접 파일을 연결시킨다. (파일들)

⇒ direct block이 다 찼다면, Indexed Block 기법을 활용해 single indirect block과 매핑

- super block에서 빈 데이터 블록 하나를 할당받아, 블럭 관리하도록 지시 (1개)

⇒ 해당 부분도 다 차게 된다면, 2번 참조하는 double indriect blocks으로 처리

- index block 들의 주소를 관리하는 하나의 블럭으로 관리 (여러 개)

⇒ 마지막 계층으로, 그러한 블럭들의 집합으로 관리하는 triple indirect blocks (매우 많음)

⇒ 단, 이렇게 계층적으로 구성하면 접근 시간이 매우 오래 걸림!

※ FCB, i-node

- 각 파일에 대한 Attribute (소유자, 타입, 수정 날짜, Data 블록 주소, 크기, Link Count 등)
- Link Count : 디렉토리에서 해당 FCB로 참조하는 갯수, dangling pointer 해결에 사용

- 파일 내용 주소 : 데이터 블록들의 블록 번호

4. **Data Block Area**
    - 데이터 블록에는 파일의 내용이 들어간다. 파일 크기가 크면 여러 데이터 블록들이 할당!
    - 파일이 주어졌을 때 → 데이터 블록들이 어떻게 구성하는지에 대한 기법? = File Allocation

3. 디렉토리와 파티션

※ Directory, Folder : 파일 메타데이터 중 일부를 보관하고 있는 특수 파일, 폴더라고도 함

⇒ 일반 파일인지, 디렉토리인지 구별하는 특수한 비트가 별도로 존재한다.

⇒ 자신에게 속하는 모든 파일과 부 디렉터리의 목록을 가진 자료구조

- 파일 이름을 해당 디렉토리 항목으로 변환해주는 symbol 테이블

⇒ 그 디텍토리에 속한 파일 이름 및 파일 속성(attribute)을 가지고 있다.

- 실제 디렉토리 안 파일 내용이 수정되면, 해당 파일의 FCB를 통해 접근해 갱신시킨다.
- FCB 접근을 통해, 같은 속성 중복 저장을 피하면서 폴더가 해당 파일을 갱신 가능케 한다.

⇒ (search for, create, delete, list, rename) a file & traverse the file system 등의 연산 수행

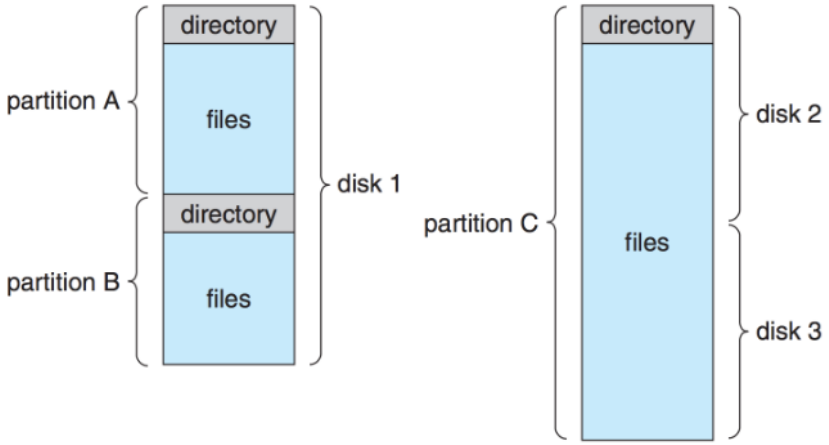


Figure 11.7 A typical file-system organization.

- 각 파티션에는 파티션 내의 파일에 대한 정보가 들어 있다. 이 정보는 장치 디렉터리 또는 볼륨에 저장된다.
- 각 파티션에는 해당 파티션에 저장된 파일에 대한 정보가 들어 있는 디렉터리 구조가 포함되어 있다.
- 디렉토리는 파일 이름을 디렉토리 및 디스크로 변환하는 테이블로 볼 수 있다.

※ Directory 구현 방법

1. **선형 리스트**
    - File name & File FCB로 구성된 List
    - 구현이 간단하나, 디렉토리 내 파일을 찾기위한 선형 탐색의 비효율성!

2. **해시 테이블**
    - 선형 리스트 + 해시
    - 해시 테이블 : file name → Hash → 선형 리스트 index
    - search 시간복잡도 : 상수 시간
    - 단, Collision 문제 발생 가능

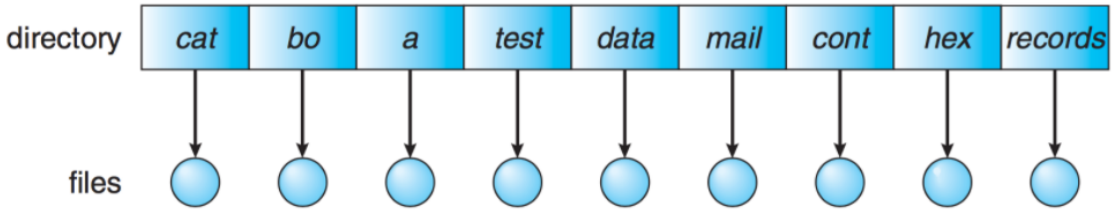
⇒ [file name & meta data, fcb] 배열의 각 entry는 고정된 크기, 만약 name이 길어진다면?

- 각 entry 마지막 부분에 이름의 뒷 부분이 위치한 곳의 포인터 설정

- 이곳에 범위를 벗어난 나머지 이름을 이어서 기록하도록 설정 가능

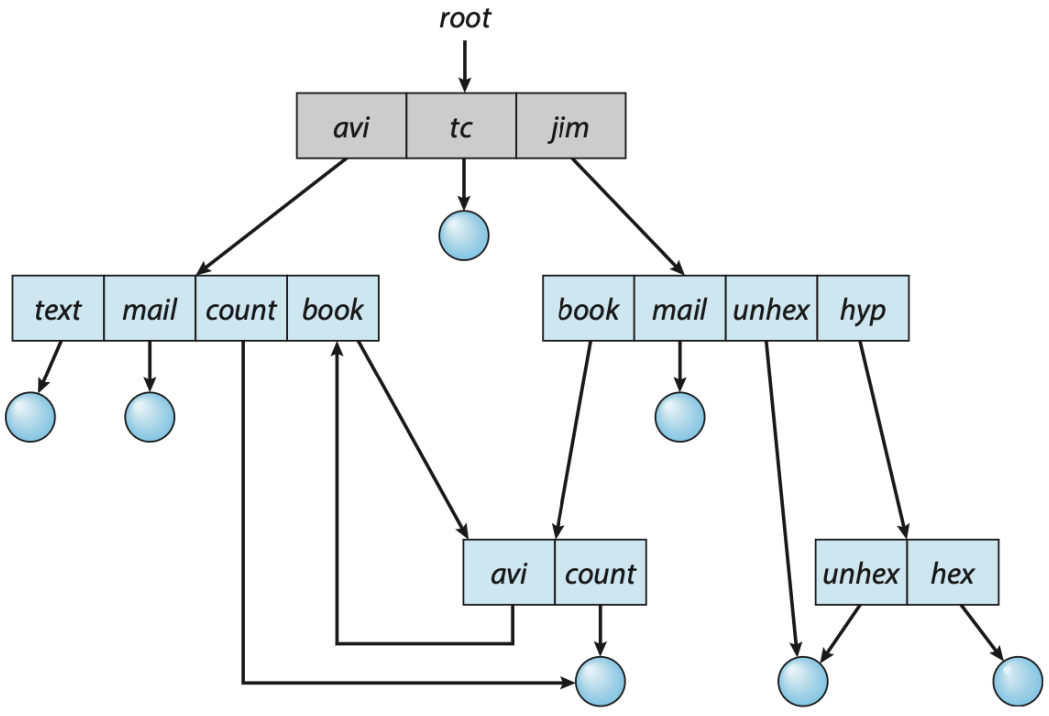
< Directory 구조 >

1. **1단계 디렉토리** (single-level directory)



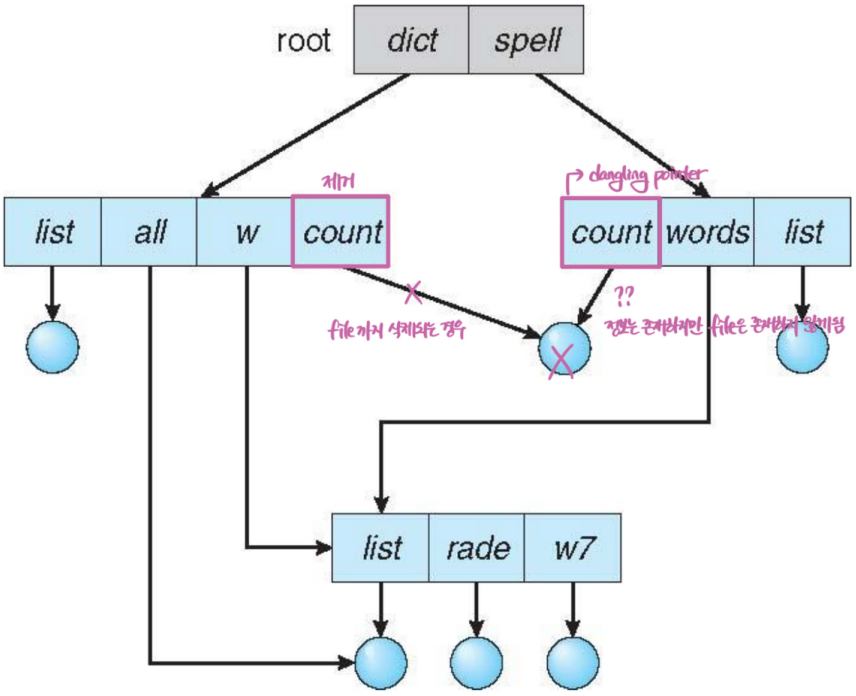
⇒ 가장 간단한 구조, 모든 파일이 한 디렉터리 내에 있기 때문에 **모두 유일한 파일명을 가져야 한다**는 제약 조건이 존재

2. **일반 그래프 디렉터리 (General Graph Directory)**



⇒ 임의 파일이나 디렉터리가 서로 다른 두 개 이상의 디렉터리를 부모로 가질 수 있음

⇒ 사이클을 형성 가능, 만약 사이클이 없다면 비순환 그래프가 되어서 트리 구조의 일반화된 유형

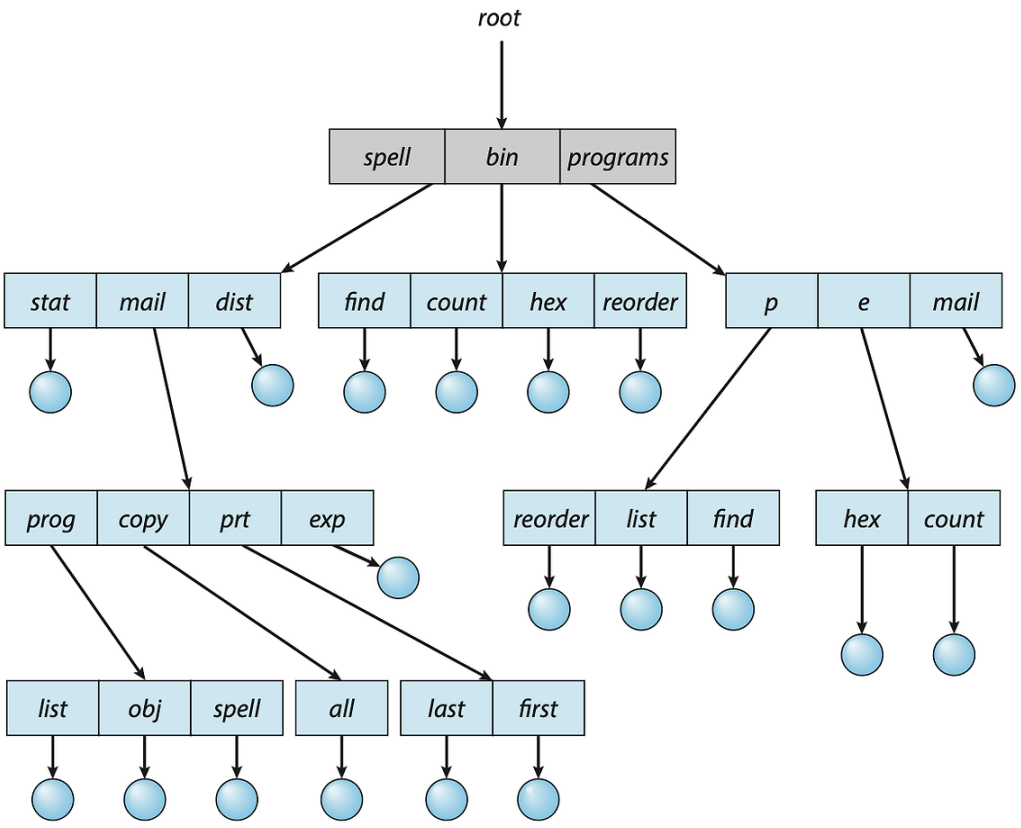


※ 하나의 파일이 여러 개의 완전한 경로 이름을 갖고 있으므로, 서로 다른 경로 이름이 같은 파일을 가리키는 경우도 존재!

⇒ 파일의 삭제 과정에서 **고아가 된 포인터(dangling pointer)** 문제가 발생할 수 있다.

- 모든 파일 시스템을 검색 → 이미 삭제된 개체 연결링크 제거 (완탐, 매우 비효율)
- Backpointers (파일과 관련된 참조 리스트 관리, 삭제 될 때 관련 모든 참조 삭제, 비효율적!)
- 사용자에게 처리를 맡긴다. (Windows)
- Entry-hold-count 솔루션 (UNIX)
  - 디렉토리에 파일이나 바로가기가 만들어질 때마다 (디렉토리 엔토리가 생길 때마다)
  - 이 파일에 해당하는 엔트리-홀드-카운터 필드 값 1씩 증가
  - 파일을 지울 때는 엔트리-홀드-카운터를 1만큼 뺌
    - 엔트리-홀드-카운터 값이 0이면 파일을 시스템 상에서 삭제
    - 1 이상이면 파일을 지우지 않음

3. 트리 구조 디렉터리 (Tree-Structured Directory)



⇒ 가장 일반적인 디렉터리 구조로, 계층 구조 디렉터리의 가장 간단한 예

⇒ 계층 구조 디렉터리 : 하나의 루트(root) 디렉터리가 다른 디렉터리나 파일을 포인팅

- 한 디렉터리는 부 디렉터리를 포인트 하면서 계층이 내려가기를 반복
- 요즘 트리에서 확장되어 (비순환) 그래프 모양으로 변화 중 (Link 간선을 기반)

⇒ 모든 파일은 고유한 경로(절대 경로 / 상대 경로)를 가짐

- 효율적인 탐색과 그룹화가 가능!

⇒ 최상위에 하나의 루트(root) 디렉터리가 존재

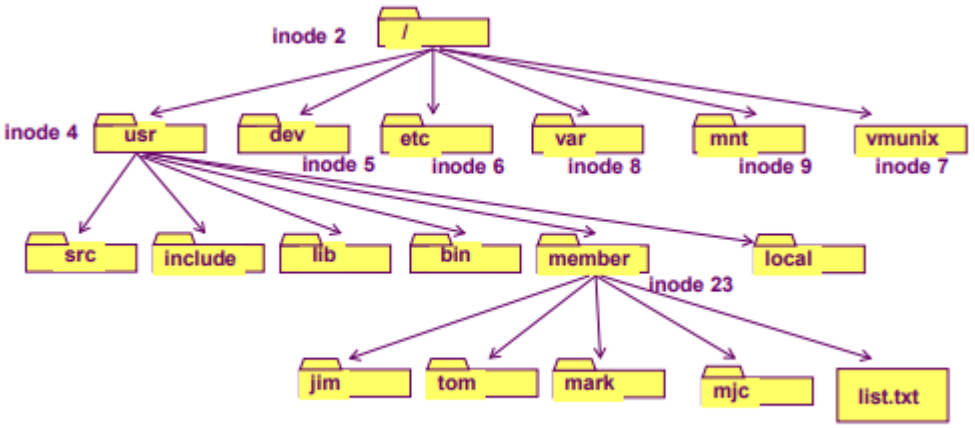
- 모든 디렉터리나 파일은 단지 한 개의 상위 디렉터리에 의해서 포인트
- 루트 디렉터리는 부모 디렉터리가 없다.

※ **Current Directory** : 계층구조 상에서 현재 위치한 디렉토리, 그래서 작업 중인 위치

⇒ A/BC/D.../E.exe 의 Path name 대신, 그냥 E.exe 등으로 파일 경로를 단순화!

- **절대 경로명(absolute path name)** : 루트에서 지정된 파일까지의 경로가 명시된 것
- **상대 경로명(relative path name)** : 현재 디렉터리를 기준으로 목적하는 파일까지의 경로를 지정하는 것

❖ Directory 내부에는 디렉토리가 담은 파일의 이름과 그 파일의 FCB가 존재



⇒ FCB 식별자 : 파일 시스템에서 해당 파일의 FCB를 찾아내는 주소 정보

⇒ 커널이 FCB를 만들어서, 보조기억장치에 있는 파일 시스템 내 FCB 리스트에 저장

- 파일의 내용은 파일 시스템의 데이터 블록 영역에 저장
- 주소, 길이, 소유자, 파일 타입 등의 속성은 FCB 리스트에 FCB로 묶어 저장
- 이를 통해, 파일 속성 및 각종 데이터로 각 파일 판별 가능
- 단지 FCB 번호만 파일 시스템에 저장, FCB 번호가 식별자이며 주소이다.

⇒ 반면, 커널이 파일을 찾고자 Directory를 읽을 땐 다음과 같다.

1. 현재 위치한 디렉토리를 기준, 이에 대한 FCB를 커널을 통해 읽어온다.
2. FCB 내 기입된 파일 데이터 블록을 읽어, 다음 위치 (폴더 or 파일)와 매칭 비교
3. 맞다면, 다음 위치에 대해 위 과정을 반복적으로 수행한다.

⇒ Open 작업은 파일에 대한 정보를 메인메모리로 읽어오는 작업을 의미!

※ 파티션 (Partition)

⇒ 연속된 저장 공간을 하나 이상의 연속되고 독립적인 영역으로 나누어 사용할 수 있도록 정의한 Logical Disk 단위

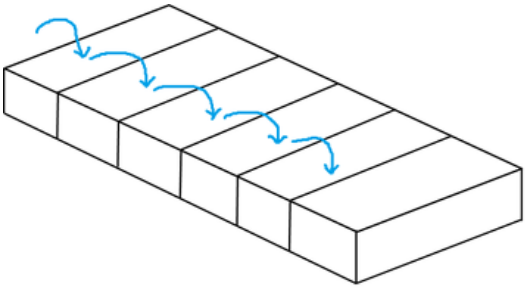
⇒ 하나의 물리적 디스크 안에 여러 파티션을 두는 게 일반적 → local disk C & local disk D

⇒ 물리 디스크를 파티션으로 구성 후, 각 파티션에 file system 설치 or swap 등 다른 용도로 사용

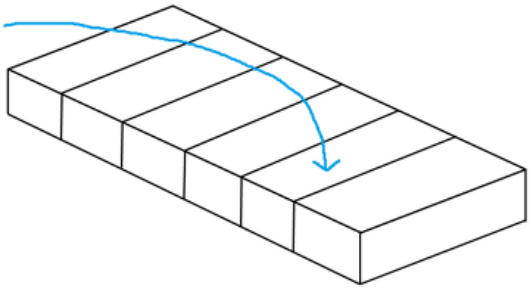
---

2. File Allocation & Access Method

※ Access 기법



sequential access



random access

1. **순차 접근 (Sequential Access)**

   ⇒ 가장 단순한 방법으로 파일의 정보가 레코드 순서대로 처리된다.

   ⇒ 현재 위치에서 읽거나 쓰면 offset이 자동으로 증가하고, 뒤로 돌아가기 위해선 되감기가 필요

2. **직접 접근 (Direct Access, Random Access)**

   ⇒ 파일을 구성하는 레코드를 임의의 순서로 접근

   ⇒ 데이터를 기억 장치로부터 호출할 때, 데이터가 저장된 위치나 접근 시간에 구애 받지 않음

3. **색인 접근 (Indexed Access)**

   ⇒ 파일에서 레코드를 찾기 위해 색인을 먼저 찾고 대응되는 포인터를 얻음

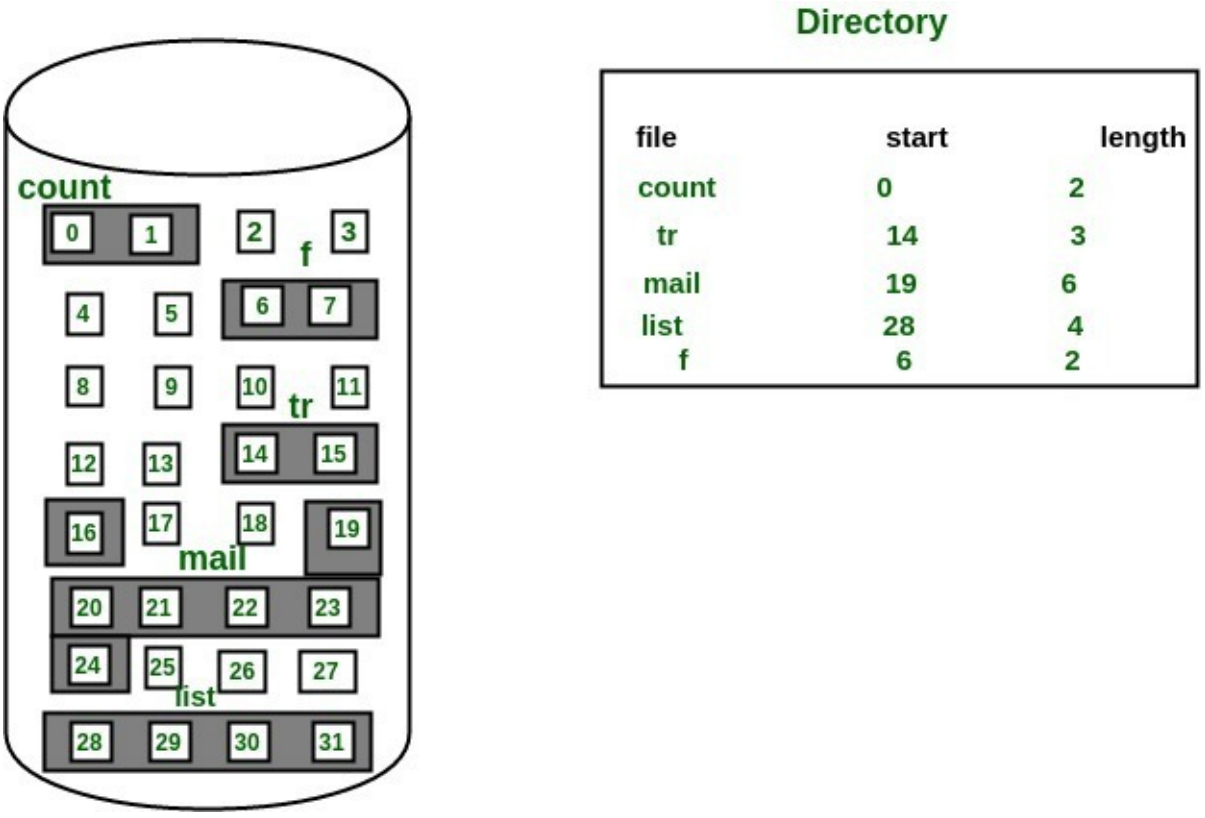
⇒ 이를 통해 파일에 직접 접근하여 원하는 데이터를 얻는다. (크기가 큰 파일에서 유용)

※ File Allocation의 정의

⇒ 데이터 블록에는 파일의 내용 삽입, 파일 크기가 크면 여러 개의 데이터 블록들이 할당

⇒ 디스크의 직접 접근 특성이 파일의 구현에 융통성을 허용

1. 연속 할당 (Contiguous Allocation)



⇒ **각 파일이 디스크 내에서 선형적으로 연속된 블록을 차지**하도록 할당해주는 방법 (연속 저장)

⇒ 그러나, 상기된 단점으로 인해 현재 사용하지 않음

※ 장점

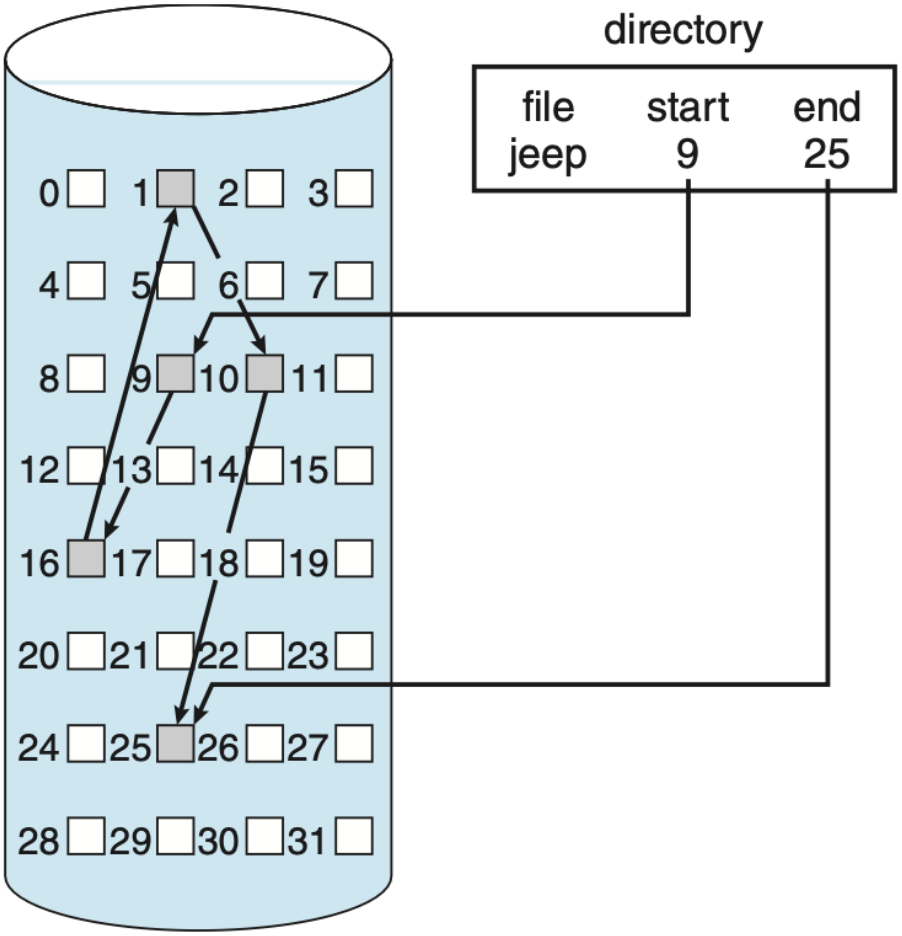
- FAST I/O, Disk 내 파일을 읽는 디스크 헤더 이동 시간 seek time이 적게 걸린다.
    - 한 번의 읽기 작업으로 많은 바이트를 읽을 수 있음

    - 이미 실행 중인 프로세스의 swapping & realtime file 용으로 사용하기 좋음
- 파일 직접 접근 가능 ⇒ (start 번호와 length 만으로 현재 위치에서 계산으로 바로 jump!)
- Directory 구현 단순화 ⇒ 시작 블록의 주소와 블록 개수만 유지

※ 단점

- File Grow 어려움, 새로운 파일의 생성을 위해서는 필요 공간의 크기를 미리 명시
    - 디스크 특정 공간을 점유한 파일의 사이즈가 동적으로 증가 시, 더 이상 증가 할 공간이 없다면 통째로 다른 빈 곳으로 이동
    - 이는 많은 디스크 연산을 요구, 시간이 기하급수적으로 오래 걸리게 된다.
- 단편화 문제 (연속 할당 기법의 문제점 공유)
    - 외부 단편 : 파일의 생성 & 삭제 반복 → 가용 공간의 단편화 (압축 기법 요구됨)
    - 내부 단편 : 파일 확장 대비, 미리 여분 공간 확보 → 기억 공간의 낭비

2. 링크 연결 할당 (Chained Allocation, Linked Allocation)



⇒ 파일의 저장을 위한 블록들이 디스크 임의의 곳에 흩어져 있어도 접근이 가능한 동적 할당 기법

⇒ 파일은 디스크 블록의 연결 리스트 형태로 저장, 각 블록은 다음 블록을 가리키는 포인터를 포함

⇒ 디렉터리는 파일의 첫 번째와 마지막 블록에 대한 포인터를 가지고 있다.

⇒ 안정성 및 시간 등의 문제로 잘 안 쓰임

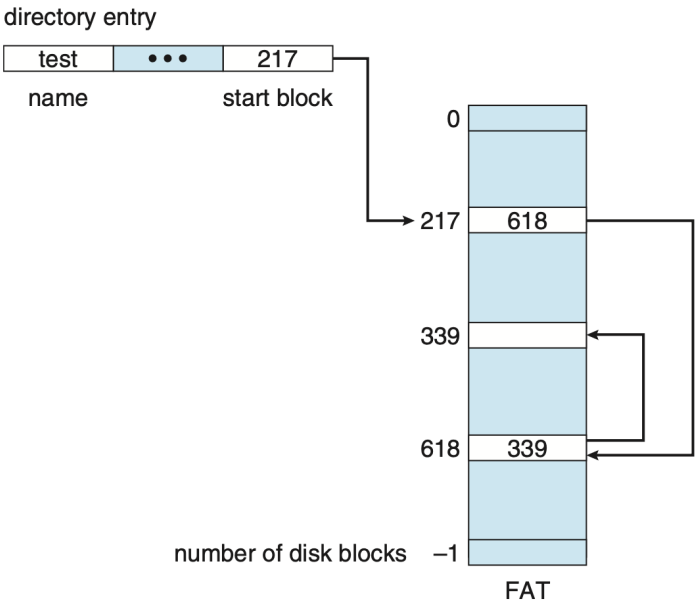
※ 장점

- 파일 크기 변화에 유연하게 대응 가능 (압축 X, 파일 사이즈 & 외부 단편 문제 해결)

※ 단점

- bad sector error : 체인 중간의 블록이 깨지면, 나머지 뒤의 블록을 모두 잃게 되는 문제
- 파일 블록들을 모두 직접 접근 하는 것이 불가능, 언제나 연결 리스트의 순차 탐색 수행
- 포인터를 위한 공간이 블록의 일부가 되어 공간 효율성을 떨어트림
    - FAT (File allocation table) 잘 활용 시, 포인터를 별도 위치에 보관해 문제점 해결

※ FAT (File allocation table)



⇒ 각 파티션의 시작 부분이 FAT로 사용, FAT 테이블은 각 디스크 블록마다 한 개의 항목

- 이 항목은 디스크 번호를 인덱스로 찾음

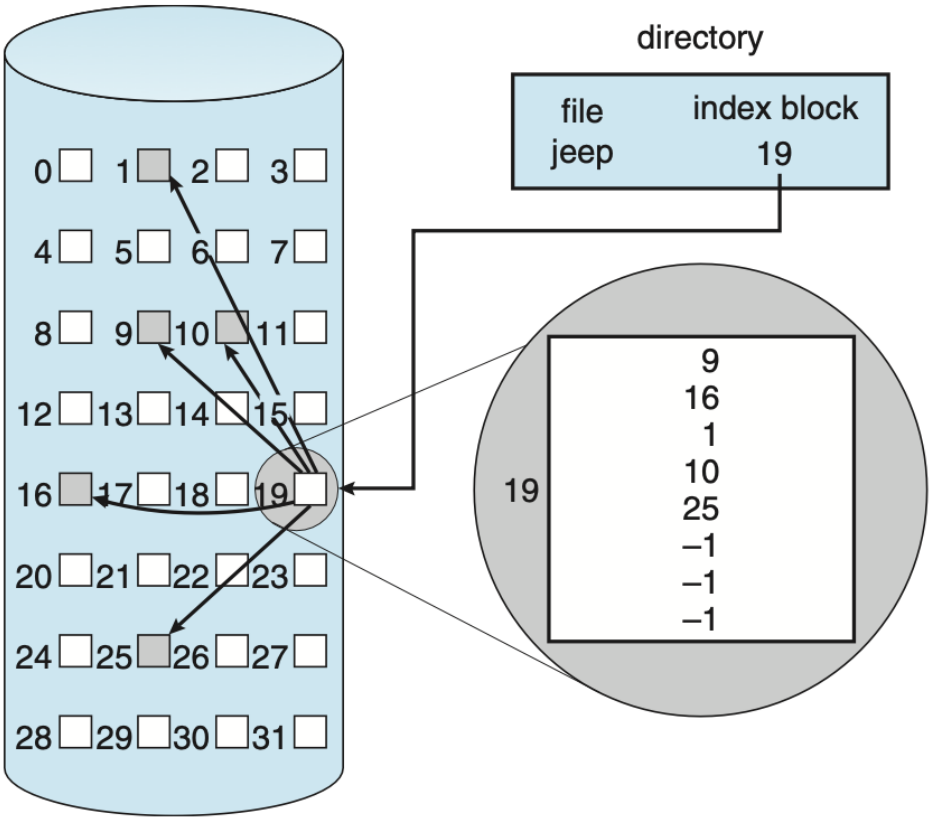
⇒ 디렉터리 항목은 각 파일의 첫 번째 블록 번호를 가리키고, FAT 항목이 다음 블록의 블록 번호를 가리키는 것으로 사슬 역할을 수행

⇒ 이러한 사슬은 마지막 블록까지 계속 되며, 마지막 블록의 테이블 항목은 파일의 끝을 나타내는 특수한 값을 갖고 있다.

⇒ 파일에 새로운 블록을 할당할 경우 → 값이 0인 테이블 항목 찾기

- 이전 파일의 끝 값을 새로운 블록의 주소로 대체

3. 색인 할당 (Indexed Allocation)



⇒ 파일의 데이터 블록 번호(주소)들을 인덱스 블록이라고 불리는 별도 블록 내에 저장

- 디스크 내 특정 하나의 블록을 선택하여, 다른 블록 번호를 관리하도록 지정
- 일종의 페이지 테이블로 기능, Paging기법과 유사

⇒ 파일 시스템은 이런 인덱스 블록들의 번호를 기록하는 방법

⇒ 디렉터리는 그 디렉터리에 속한 각 항목의 색인 블록에 대한 포인터를 소유

※ 장점

- 탐색이 색인 블록 자체에서 일어나기 때문에 속도가 빠르다.
  - **색인 블록을 주 기억 장치에 유지 →** 탐색 시간을 더 줄일 수 있다.
- 모든 단편화 없이 **직접 접근**을 제공

※ 단점

- 파일을 중간에 삽입하고자 할 경우 색인 블록을 완전히 재구성
- 만약 인덱스 블록이 깨진다면, 해당 기법은 완전히 터지는 문제점

  (그래도, 체인 기법보단 안정성 좋음)
- **공간의 낭비**
  - 작은 파일에도 하나의 색인 블록이 할당
  - 향후 파일 삽입을 위해 색인 블록을 남김 → 이는 기억 장치의 낭비
- 하나의 색인 블록으로 모든 블록을 색인 할 수 없을 정도의 큰 파일은 저장할 수 없음

※ Indexed Allocation + Contiguous Allocation

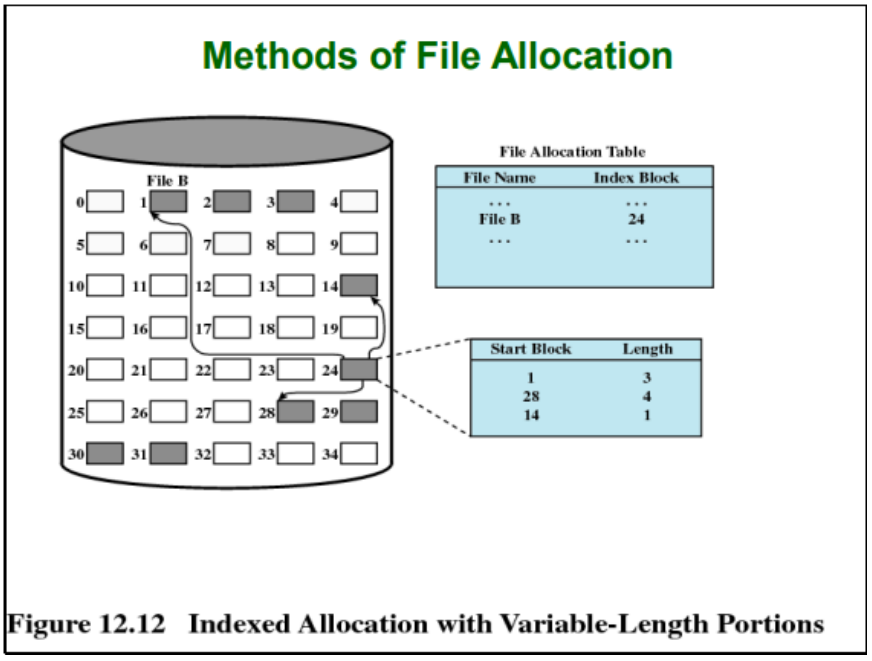


Figure 12.12 Indexed Allocation with Variable-Length Portions

⇒ 인덱스와 mapping된 파일은 정해진 크기 만큼을 연속한 블록으로 가지고 있음

---

3. Free Space Management

⇒ 반대로, 빈 디스크 블록을 관리하는 기법, 현재 시점에 어느 블록이 비어 있는지 찾는 문제!

⇒ 디스크의 공간은 제한 → 삭제된 파일이 차지하던 공간 → 새로운 파일들을 위해 재 사용

⇒ 때문에, 파일 시스템은 빈 공간을 리스트로 유지하고 관리해야만 한다.

1. Counting

⇒ 연속 할당 기법을 빈 데이터 블록들에 대해 적용한 것, 원리는 같으나 빈 데이터 블록 처리 활용

⇒ **연속된 자유 블록의 첫 번째 블록 주소와 연속된 블록의 개수(count)만 기록**한 리스트를 관리

- 이를 통해 자유 공간 리스트의 크기는 작아진다.

2. Linked List

⇒ Chain 할당 기법을 빈 데이터 블록들에 대해 적용한 것, 빈 데이터 블록에 대한 연결 리스트 관리

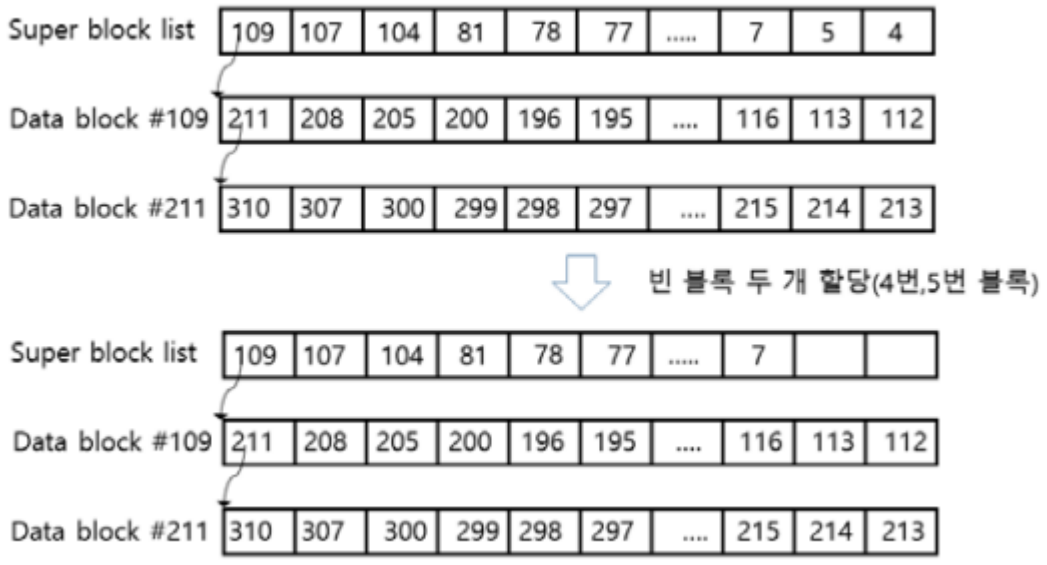
⇒ 단순히 빈 데이터 블록만 찾는 순서대로, 파일을 삽입해주기만 하면 된다.

- 단편화 문제는 해결

⇒ 특정한 공간에 접근하려면 list의 헤더부터 탐색 해야 되기 때문에 비효율적

⇒ bad sector로 비롯되는 안정성 문제는 여전히 존재

3. Grouping



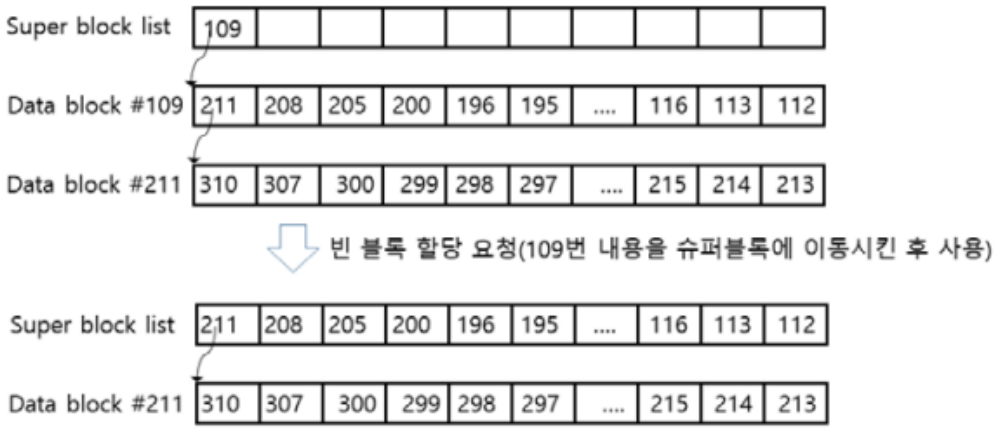
⇒ 빈 데이터 블록들에 대해 Indexed 할당 기법을 적용한 것, 빈 블록을 관리하는 인덱스 블록 존재

⇒ 해당 인덱스 블록을 관리하는 블록은 Super Block, 빈 블록의 번호를 차례대로 해당 블록에 기록

- 해당 블록이 꽉 차게 되면, 가장 위 번호의 데이터 블록을 임시 인덱스 블록으로 활용
- 해당 블록도 꽉 차면, 다시 위 번호 블록 가져와 체인으로 연결

⇒ 이처럼, 슈퍼 블록 하나 → (임시적) 일반 데이터 블록의 Linked List 으로써 관리하게 된다.

⇒ 빈 블록이 필요할 때 마다, 해당 List의 오른편에서 왼편쪽 방향으로 번호를 빼서 할당



- 109번 인덱스 블록을 빈 데이터 블록으로 할당하기 전, 109번에 있는 빈 데이터 블록 집합을 Super block list로 올림

4. Bit Vector (bit map)

⇒ 각 블록을 1비트로 표현하여 Free-block에는 0, 사용중인 block에는 1을 부여하여 free - space를 관리하는 방식 (빈 블록이 1이냐 0이냐는 관리 방식의 차이, 유닉스는 반대)

- Ex) { 0, 1, 4, 5, 8 } 번 블록이 Free Block (bit = 1) 인 길이 10의 bit map (유닉스 시스템)

```
0011001101
```

⇒ 디스크 내에서 직관적으로 관리되어서 free-space에 찾기 쉬움

- 첫번째 free-block과 n개의 연속된 free-space를 탐색하는데 유리

⇒ 비트 벡터가 길어질수록, 부가적인 공간을 필요로 한다는 특징

- 디스크 용량이 커질수록, 비트맵 크기 자체도 커지게 된다.

5. 공간 맵 (Space Map)

⇒ (Oracle의 ZFS에서 사용하는 방식)

⇒ 해당 파일 시스템은 대규모의 파일, 디렉터리, 심지어 파일 시스템도 저장할 수 있도록 설계

- 큰 규모에서는 메타데이터 입출력이 성능에 커다란 영향을 미침

⇒ ZFS는 자유 공간을 관리할 때 자료 구조의 크기를 제어

- 이 자료 구조를 관리하기 위해 필요한 입출력을 최소화 요구 → 여러 기법을 조합하여 사용

1. 대용량의 공간을 관리 가능한 크기의 덩어리로 나누기 위해 **메타 슬랩(metaslabs)을** 생성
    a. 한 볼륨은 수백 개의 메타 슬랩을 포함하며, 각 메타 슬랩은 연관된 영역의 **공간 맵을** 가진다.
2. 공간 맵은 할당과 반환의 모든 블록 활동을 **계수** 형식으로 시간 순서로 기록

---

4. 가상 파일 시스템, VFS (공통 인터페이스)

⇒ 특정 파일을 서로 다른 파일 시스템에서도 접근 가능하도록 구현한 응용 프로그램

⇒ 다종다양한 file system에 대해 동일한 시스템 콜 인터페이스(API)를 통해 접근 시켜주는 OS Layer

- 커널은 VFS 형태의 시스템 콜을 호출, 이를 통해 파일 시스템 종류와 무관하게 동일 기능 제공

※ 네트워크 파일 시스템 (Network File System)

⇒ NFS는 네트워크 상에서 파일시스템을 공유하도록 설계된 파일 시스템

⇒ 다른서버의 파티션을 마치 내 로컬영역인 것처럼 네트워크를 이용하여 사용할 수 있다.

⇒ 해당 시스템을 통해 분산 시스템에서 네트워크를 통해 파일이 공유될 수 있다.
