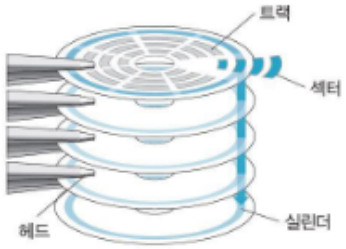
11. File System

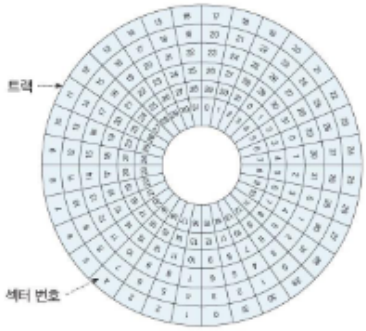
Disk System

데이터는 디스크에 저장됨

Disk Pack

> 데이터 영구 저장 장치 (비휘발성)



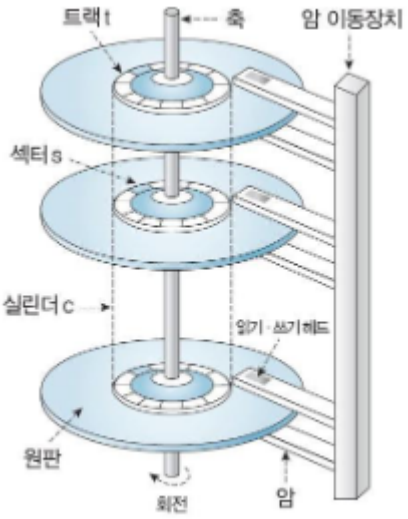


- Sector
  - 데이터 저장/판독의 물리적 단위
- Track
  - Platter 한 면에서 중심으로 같은 거리에 있는 sector들의 집합
- Cylinder
  - 같은 반지름을 갖는 track의 집합
- Platter
  - 양면에 자성 물질을 입힌 원형 금속판
  - 데이터의 기록/판독이 가능한 기록 매체
- Surface
  - Platter의 윗면과 아랫면

Disk Drive

> Disk pack에 데이터를 기록하거나 판독할 수 있도록 구성된 장치
>
> - Platter에 저장된 데이터를 읽어들임

- Head



- 디스크 표면에 데이터를 기록/판독

- Arm
  - Head를 고정/지탱
  - 모든 면에 있는 정보를 읽어야해서 앞/뒷면 둘 다 있음
- Positioner (boom)
  - Arm을 지탱
  - Head를 원하는 track으로 이동
- Spindle
  - Disk pack을 고정 (회전축)
  - 분당 회전 수 (RPM, Revolutions Per Minute)

Disk Address

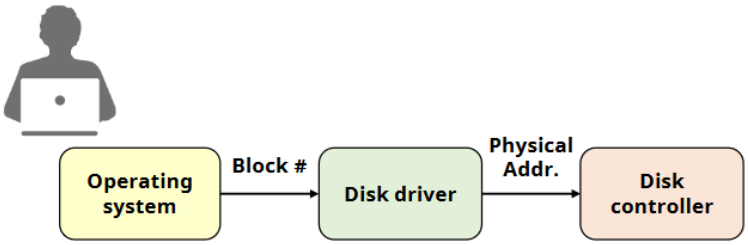
- Physical Disk Address
  - Disk Drive에 원하는 Sector를 찾아가기 위해 필요한 정보
  - Sector (물리적 데이터 전송 단위)를 지정

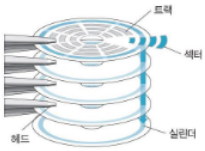
(a)	Cylinder Number	Surface Number	Sector Number
(b)	Surface Number	Cylinder Number	Sector Number

- Logical disk address: relative address
  - OS에서 생각하는 논리적인 Disk 구조
  - Disk system의 데이터 전체를 block들의 나열로 취급
    - Block에 번호 부여
    - 임의의 block에 접근 가능
  - Block 번호 (논리 주소/상대 주소) ➡️ physical address 모듈 필요 (disk driver)

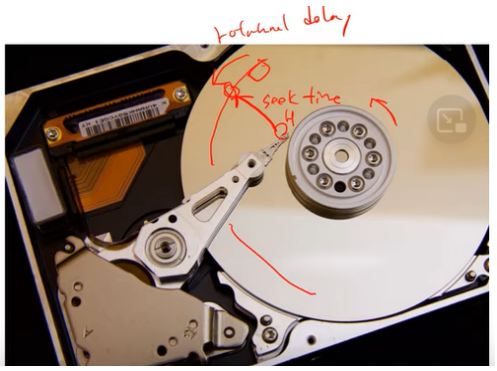
B_0	B_1	B_2	...	B_{n-2}	B_{n-1}

- Disk Driver가 논리 주소를 물리 주소로 바꿔주는 역할을 함 (HW 벤더가 제공)





Data Access In Disk System



1. Seek Time (탐색 시간)
    - 디스크 head를 필요한 cylinder로 이동하는 시간
2. Rotational Delay
    - 1) Seek Time 이후로부터, 필요한 sector가 head 위치로 도착하는 시간
3. Data Transmission Time
    - 2) Rotational Delay 이후로부터, 해당 sector를 읽어서 전송(or 기록)하는 시간

- Data Access Time = Seek Time + Rotational Delay + Data Transmission Time

File System

> 사용자들이 사용하는 파일들이 관리하는 운영체제의 한 부분

File System의 구성

- File
    - 연관된 정보의 집합
- Directory Structure

    - 시스템 내 파일들의 정보를 구성 및 제공
- Partition
    - Directory들의 집합을 논리적/물리적으로 구분
    - ex) C 드라이브, D 드라이브

File Concept

> 보조 기억 장치(Disk)에 저장된 연관된 정보들의 집합
>
> - 보조 기억 장치 할당의 최소 단위
> - (물리적 정의) Sequence of bytes

[내용에 따른 분류]

- Program files
    - Source program, object program, executable files
- Data File

[형태에 따른 분류]

- Text (ASCII) file
- Binary file

File Attributes (속성)

- Name
- Identifier
- Type
- Location
- Size
- Protection
    - Access Control Information
- User identification (owner)
- Timer, date

- creation, late reference, last modification

File Operations

➡️ OS는 file operation들에 대한 system call을 제공해야 함

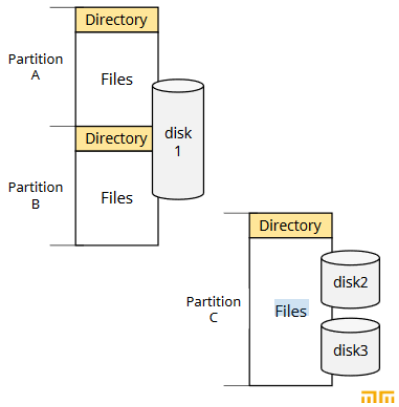
- Create
- Write
- Read
- Reposition
- Delete
- etc

File Access Methods

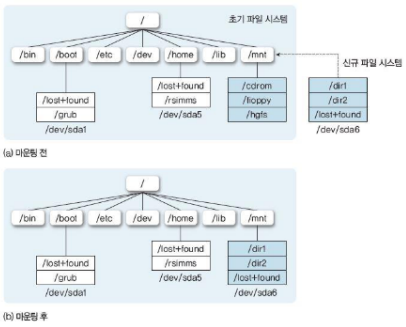
- Sequential Access (순차 접근)
  - File을 record(or bytes) 단위로 순서대로 접근
    - ex. `fgetc()`
- Directed Access (직접 접근)
  - 원하는 block(파일)을 직접 접근
    - ex. `lseek()`, `seek()`
- Indexed Access
  - Index를 참조하여, 원하는 block를 찾은 후 데이터에 접근
    - ex. `a[index[2]]`

File System Organization

- Partitions (minidisks, volumes)
  - Virtual disk: 논리적 Disk
- Directory (폴더)
  - File들을 분류, 보관하기 위한 개념
  - Operations on directory
    - search, create, delete, rename file



- list a directory (`ls`, `dir`)
- traverse the file system
- OS가 system call을 통해 제공해줘야 함



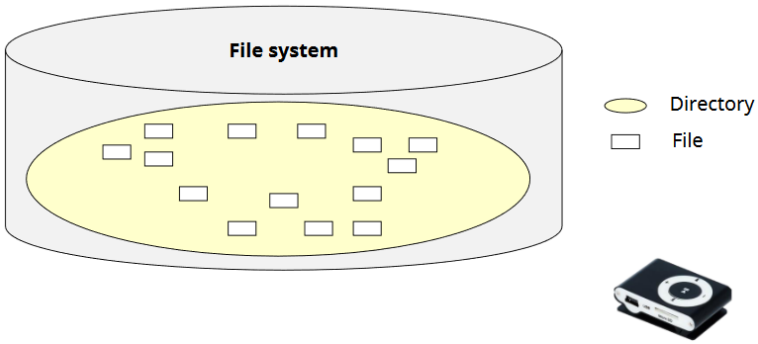
- Mounting
  - 현재 file system에 다른 file system에 붙이는 것
- Mount Point
  - Mounting 하는 지점 `/mnt`

Directory Structure

- Logical directory structure
  - Flat (single-level) directory structure
  - 2-level directory structure
  - Hierarchical (tree-structure) directory structure
  - Acyclic graph directory structure
  - General graph directory structure

Flat Directory Structure

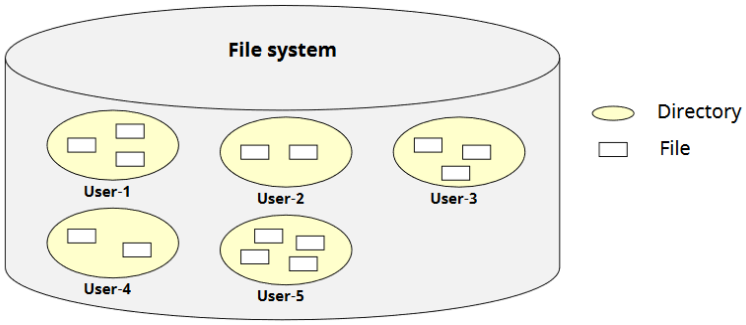
> File system 내에 하나의 directory만 존재
>
> - Single-level directory structure (계층 X, 큰 폴더 하나에 모든 파일 저장)



- Issues
  - File naming: 이름 충돌 발생 가능
  - File protection: 다중 사용자 환경에서 같은 파일 이름 설정해서 덮어씌워진다든지
  - File management
    - 다중 사용자 환경에서 문제가 더욱 커짐

2-level Directory Structure

> 사용자마다 하나의 directory 배정

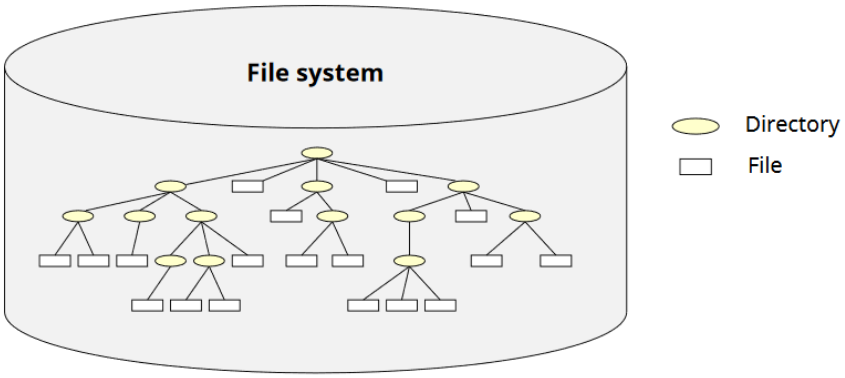


- 구조
  - MFD (Master File Directory)
  - UFD (User File Directory): MFD 아래 UFD 존재
- Problems
  - Sub-directory 생성 불가능
    - File naming issue: 내부적으로 sub directory 불가능, 파일 이름 충돌
  - 사용자 간 파일 공유 불가

    - 공유하려면 전체에 접근할 수 있도록 권한 줘야하는데, 보안 문제 발생

Hierarchical Directory Structure

> Tree 형태의 계층적 directory 사용 가능
>
> - 폴더 안에서 새로운 폴더 생성 가능

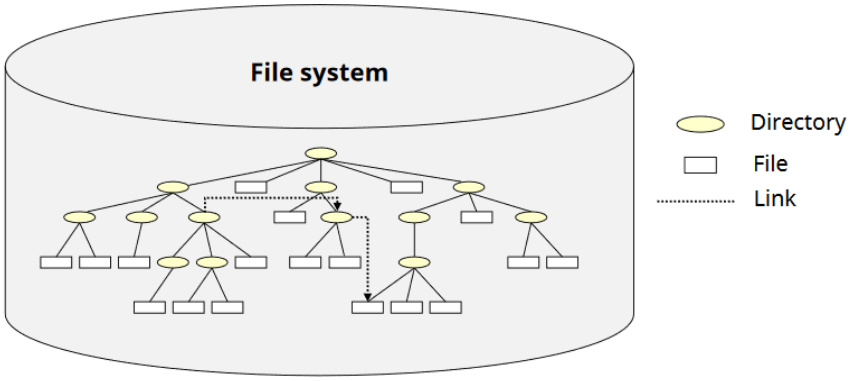


- 사용자가 하부 directory 생성/관리 가능
    - System call이 제공되어야 함 (OS 제공)
- Terminologies
    - Home directory: 제일 root directory
        - ex. `ls ~`: 나한테 할당된 home directory
    - Current directory: 현재 탐색 중인 directory
    - Absolute pathname (절대 경로): home directory부터 목표까지 전체 경로
        - ex. `/A/B/abc/`
    - Relative pathname (상대 경로): current directory부터 목표까지 전체 경로
        - ex. 현재 A일 때: `./B/abc`, 현재 C일 때: `../B/abc`
- 대부분의 OS가 사용

Acyclic Graph Directory Structure

> Hierarchical directory structure 확장
>
> - Directory 안에 shared directory, shared file를 담을 수 있음

> - 원형 그래프 발생 불가



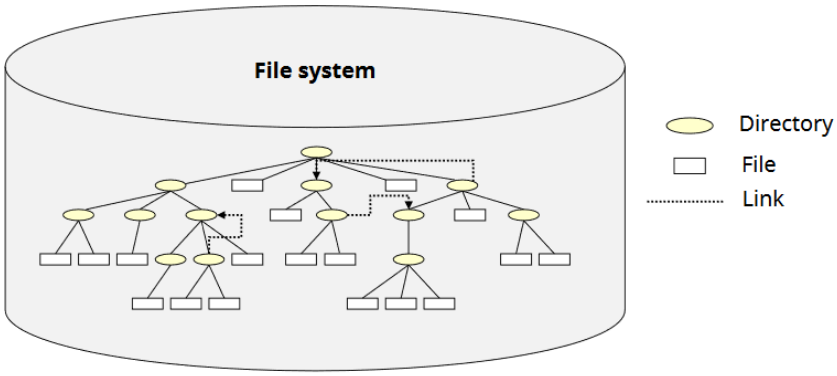
- Link의 개념 사용
  - ex. 바로가기 등. 폴더 내 다른 폴더로 갈 수 있는 link를 가질 수 있음
    - link 타고 이동한다고 해도, cycle이 발생하지 않음
  - ex. Unix system의 symbolic link

```
$ mkdir -p /tmp/one/two
$ echo "test_a" >/tmp/one/two/a
$ echo "test_b" >/tmp/one/two/b
$ cd /tmp/one/two
$ ls -l
-rw-r--r-- 1 user group 7 Jan 01 10:01 a
-rw-r--r-- 1 user group 7 Jan 01 10:01 b

$ cd /tmp
$ ln -s /tmp/one/two three
$ ls -l three
lrwxrwxrwx 1 user group 12 Jul 22 10:02 /tmp/three -> /tmp/one/two
$ ls -l three/
-rw-r--r-- 1 user group 7 Jan 01 10:01 a
-rw-r--r-- 1 user group 7 Jan 01 10:01 b
```

General Graph Directory Structure

> Acyclic Graph Directory Structure의 일반화
>
> - Cycle을 허용



- Problems
  - File 탐색 시, infinite loop를 고려해야 함

File Protection

> File에 대한 부적절한 접근 방지
>
> - 다중 사용자 시스템에서 더욱 필요

- 접근 제어가 필요한 연산들
  - Read(R), Write(W), Execute(X), Append(A)

File Protection Mechanism

파일 보호 기법은 system size 및 응용 분야에 따라 다를 수 있음

1. Password 기법
   - 각 file들에 PW 부여
   - 비현실적
     - 사용자들이 파일 각각에 대한 PW를 기억해야 함
     - 접근 권한별로 서로 다른 PW를 부여해야 함
2. Access Matrix 기법

Access Matrix

> 범위(domain, 사용자/사용자 그룹)와 개체(object, 파일) 사이의 접근 권한을 명시 (2차원 테이블 사용)

[Terminologies]

- Object
  - 접근 대상(file, device 등 HW/SW objects)
- Domain (protection domain)
  - 접근 권한의 집합
  - 같은 권한을 가지는 그룹(사용자, 프로세스)
- Access right
  - <object-name, rights-set>

[예시]

Object / Domain	F1	F2	F3	F4	F5
D1	R	R		RW	
D2	RW			RA	
D3		R		RW	X
D4	RW		X		

Object: File, Domain: User/User Group

[Implementation]

- Global table
- Access list
- Capability list
- Lock-key mechanism

Global Table

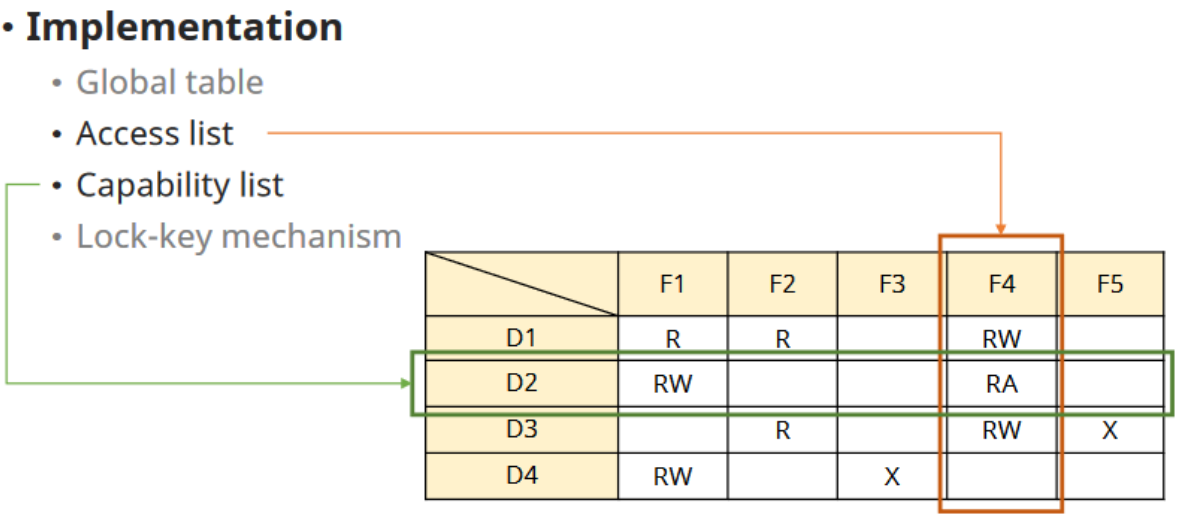
- 시스템 전체 file들에 대한 권한을 Table로 유지 (그대로 저장)
  - <domain-name, object-name, right-set>

Domain name	Object name	Right-set
D1	O1	R1
D2	O2	R2
D3	O3	R3
...	...	...

- 단점
  - Large table size
    - 공간 낭비: access matrix의 빈 영역 또한 다 저장하게 됨
    - 오버헤드 발생: 모든 내용을 다 저장해야 함

Access List & Capability List

- 빈 영역을 저장하지 않아도 되는 방법
  - Access List: Column 단위
  - Capability List: Row 권한



Access List

> Access matrix의 열(column)을 list로 표현
>
> - 각 object에 대한 접근 권한을 나열

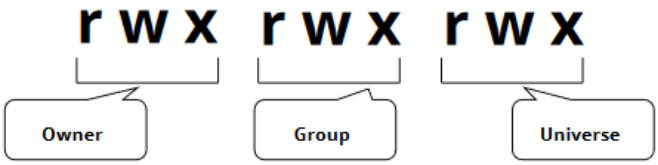
> - $A_{list}(F_k) = \{< D_1, R_1 >, < D_2, R_2 >, ..., < D_m, R_m >\}$

[구현]

- Object 생성 시, 각 domain에 대한 권한 부여
- Object 접근 시, 권한을 검사

[기타]

- 실제 OS에서 많이 사용됨
  - UNIX의 예: `ls -l`



- 공간 낭비 줄일 수 있음

Capability List

> Access matrix의 행(row)을 list로 표현
>
> - 각 domain에 대한 접근 권한 나열
> - $C_{list}(D_1) = \{< F_1, R_1 >, < F_2, R_2 >, ..., < F_p, R_p >\}$

- Capability를 가짐이 권한을 가짐을 의미
  - 프로세스가 권한을 제시, 시스템이 검증 승인
  - 파일에 접근할 때, 매번 Access List를 확인하는 게 아니라 Capability List를 사용하게 됨 (연산 줄어듦)
- 시스템이 capability list 자체를 보호해야 함
  - Kernel 안에 저장 (오버헤드 발생)

Lock-key Mechanism

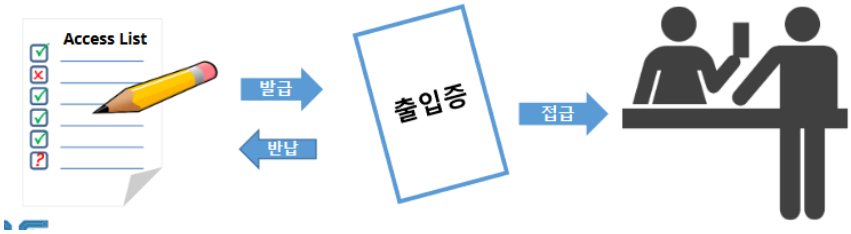
> Access list와 Capability list를 혼합한 개념

- Object는 Lock을, Domain은 Key를 가짐
  - Lock/key : unique bit patterns
- Domain 내 프로세스가 object에 접근 시
  - 자신의 key와 object의 lock짝이 맞아야 함
- 시스템은 key list를 관리해야 한다는 오버헤드 존재

Comparison of Implementations

- Global table
  - Simple, but can be large (테이블 사이즈)
- Access list
  - Object 별 권한 관리가 용이함
  - 모든 접근마다 권한을 검사해야 함
    - Object 많이 접근하는 경우 ➡️ 느림
- Capability list
  - List 내 object들(localized info.)에 대한 접근에 유리
  - Kernel에서 Capability list를 관리해야 함
  - Object별 권한 관리(권한 취소 등)가 어려움
    - 모든 Domain을 찾아가면서 권한 관리 작업을 해줘야 함

Comparison of Implementations



> 많은 OS가 access list와 capability list 개념을 함께 사용

- Object에 대한 첫 접근 ➡️ access list 탐색 (권한 있는지 확인)

    - 접근 허용 시, capability 생성 후 해당 프로세스에게 전달
    - 이후 접근 시에는 권한 검사 불필요
- 마지막 접근 후 ➡️ capability 삭제

File System Implementation

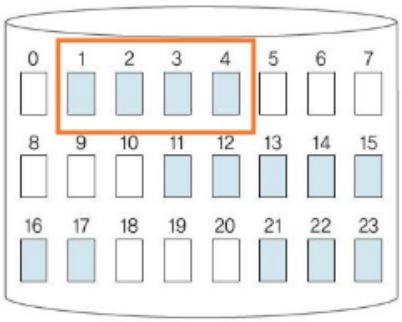
- Allocation Methods
    - File 저장을 위한 디스크 공간 할당 방법
- Free Space Management
    - 디스크의 빈 공간 관리

Allocation Methods

- Continuous allocation
- Discontinuous allocation
    - Linked allocation
    - Indexed allocation

Continuous Allocation

> 한 File을 디스크의 연속된 block에 저장



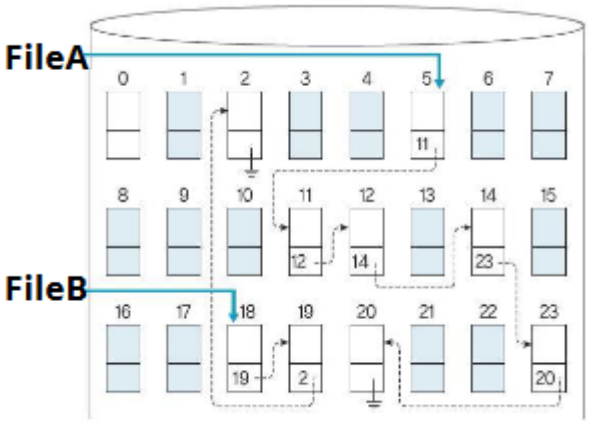
- **장점**
    - 효율적인 file 접근(순차, 직접 접근 모두 효율적)
- **문제점:** 공간 할당

- 새로운 file을 위한 공간 확보가 어려움
- External fragmentation
  - 위 상황에서 block 10개짜리 파일 B를 넣으려고 한다면?
- File 공간 크기 결정이 어려움
  - 처음 생성 시보다 파일이 커지는 경우 고려해야 함

Discontinuous Allocation

Linked Allocation

> File이 저장된 block들을 linked list로 연결
>
> - 비연속 할당 가능



- Directory는 각 file에 대한 첫번째 block에 대한 포인터를 가짐
- **장점**
  - Simple: 단순한 공간 할당
  - No external fragmentation
- **단점**
  - 직접 접근에 비효율적
    - linked list를 따라 계속 이동해야 함
  - 포인터 저장을 위한 공간 필요
  - 신뢰성 문제

- 사용자가 포인터를 실수로 건드리는 문제 (link를 끊었다면?) 등

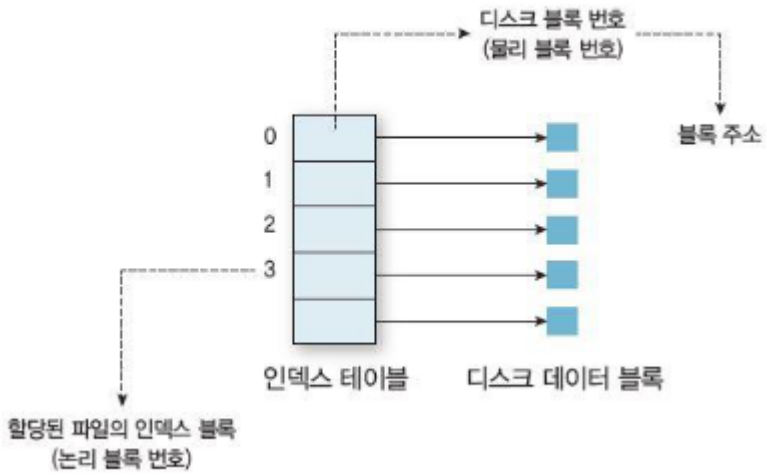
Linked Allocation: variation ➡️ FAT

[그림출처] 운영체제, 한빛미디어

- **File Allocation Table (FAT)**
  - 각 block의 시작 부분에 다음 블록의 번호(link)를 기록하는 방법
- MS-DOS, Windows 등에 사용됨

Indexed Allocation

> File이 저장된 block의 정보(pointer)를 index block에 모아둠



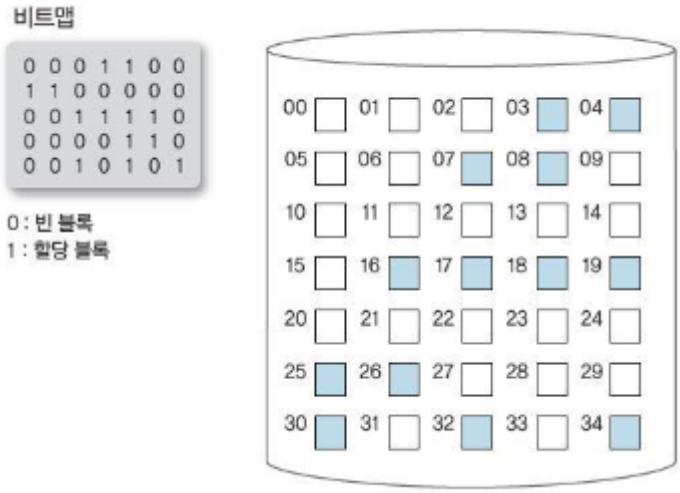
- 직접 접근에 비효율적 (linked allocation 대비)
  - 순차 접근에는 비효율적 (인덱스에 다시 접근한 후 이동해야 함)
- File 당 index block을 유지
  - Space overhead: 추가 공간 필요

    - Index block 크기에 따라 파일의 최대 크기가 제한됨
- Unix 등에 사용됨

Free Space Management

Bit Vector

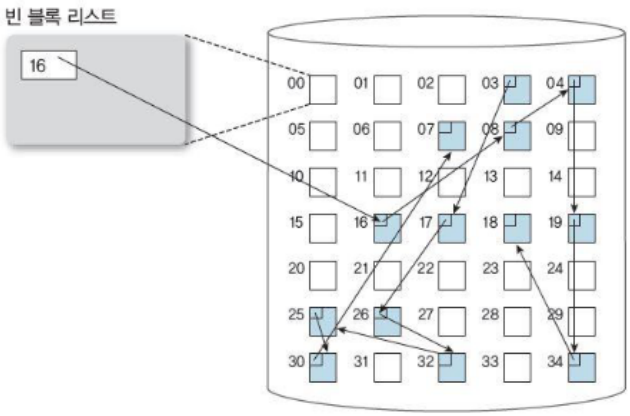
> 시스템 내 모든 block들에 대한 사용 여부를 1 bit flag로 표시



- Simple & efficient
- 비트맵(Bit vector) 전체를 메모리에 보관해야함
    - 대형 시스템에 부적합

Linked List

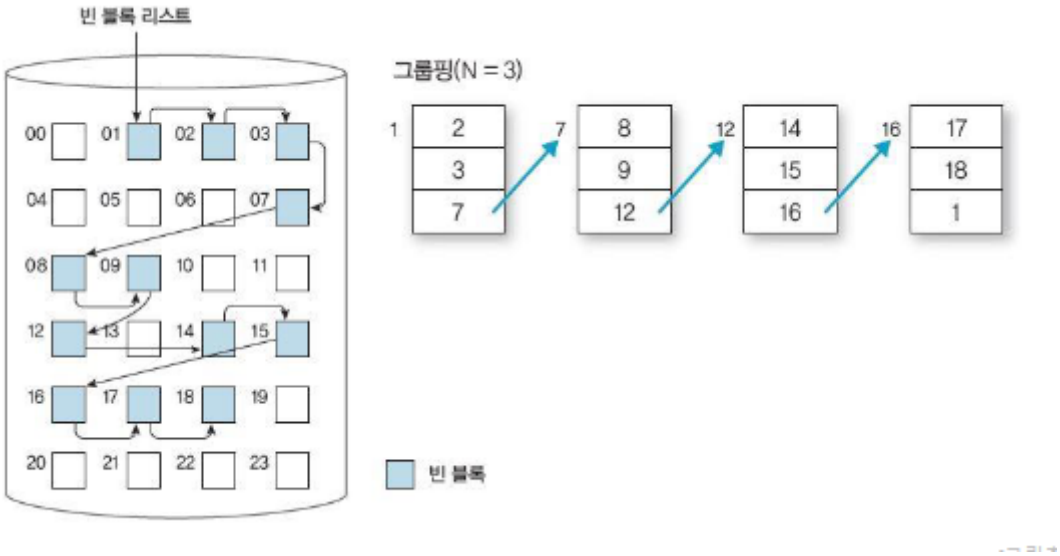
> 빈 block을 linked list로 연결



- 공간, 탐색 시간 둘 다 비효율적

Grouping

> n개의 빈 block을 그룹으로 묶고, 그룹 단위로 linked list로 연결



- 필요한 링크의 수가 Linked List 대비 $1/N$으로 줄어듦
- 연속된 빈 block을 쉽게 찾을 수 있음

Counting

> 연속된 빈 block들 중 첫 번째 block의 주소와 연속된 block의 수를 table로 유지

- Continuous allocation 시스템에 유리한 기법
  - ex. 빈 공간 테이블: 0번 index에서 6칸 비어있음, 8번 index에서 3칸 비어있음 등을 표시