12. File system

`File` : 데이터나 program을 담기 위한 그릇

- File의 첫 부분은 offset = 0
- File 자체가 **하나의 연속적인 logical address space**이다.

`File system` : 그 그릇을 관리하는 시스템

File structure

1. `None` : 특정한 구조 없이, Byte, word의 연속

2. `Simple record structure` : 한 Line이 하나의 의미 단위를 갖는 구조
   - Fixed length / Variable length 로 나뉜다.

3. `Complex record` : 복잡한 구조를 가짐
   - **Formatted document:** hwp, word 등 그 자체로 내부적인 형식을 갖는 파일
     - Application이 이 형식을 정해준다.
   - **Relocatable load file:** OS가 정한 자체적인 형식을 갖는 복잡한 파일

파일 형태는 **Application이 정할 수도, OS가 정할 수도 있다.**

- OS의 실행 파일은 OS가 형식을 정한다.
  - window → .exe
  - Linux → .ELF

File Attributes (File Metadate)

File을 관리하기 위해서는 아래와 같은 정보가 필요하다.

- **Name** – only information kept in human-readable form
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time**, **date**, **and user identification** – data for protection, security, and usage monitoring

이 정보들은 directory structure에 저장되어져 있다.

- File 접근 전에 폴더에 먼저 접근해야 하는 이유이다.

File Operations

create

write

read

repositioning within file – file seek

delete

truncate

open(F_i) – Copies file metadata from disk to memory
(Search the directory structure to do this)

close (F_i)

- `File seek`: file 내에서 **접근하고자 하는 offset (위치)**을 바꿈
- `open()`: Memory로 파일의 메타데이터를 복사
  - File을 다루기 위해서는 `fopen()`이 반드시 먼저 호출되어야 한다.

Directory

Directory가 존재하는 이유??

File을 harddisk 아무 곳에나 흩어놓는다면, 어디에 어느 파일이 있는지 관리하기가 어렵다.

- 비슷한 종류의 파일을 모아 Grouping하는 것이 불가능하다.

이에 directory를 사용하여 **아래 두 가지**를 제공한다.

- User에게는 File을 체계적으로 관리하기 위한 구조를 제공한다.
- File system에게는 `Naming interface` 를 편리하게 해준다.
  - Harddisk에 있는 모든 파일의 이름을 모두 다르게 하는 것은 쉽지 않기에 Directory name까지 추가하여 File을 구별할 수 있게 한다.

    - **다른 Diretory라면 같은 이름의 파일이 있을 수 있다.**
- 결론적으로, 파일에 대한 특정 데이터나 아이템이 어디 있는지에 대한 **정보를 정확한 Disk 상 위치를 모르더라도 어느 directory의 어느 file**이라고 말할 수 있게 된다.

최근의 OS는 `Multi level directory` 를 사용한다.

- **file name이 root directory부터 시작해서 leaf까지로 표현**되는 것

System은 `Current directory` 를 관리할 수도 있다.

- 이를 통해 **root부터 시작하는 절대 주소 대신, 상대 주소를 사용할 수 있다.**

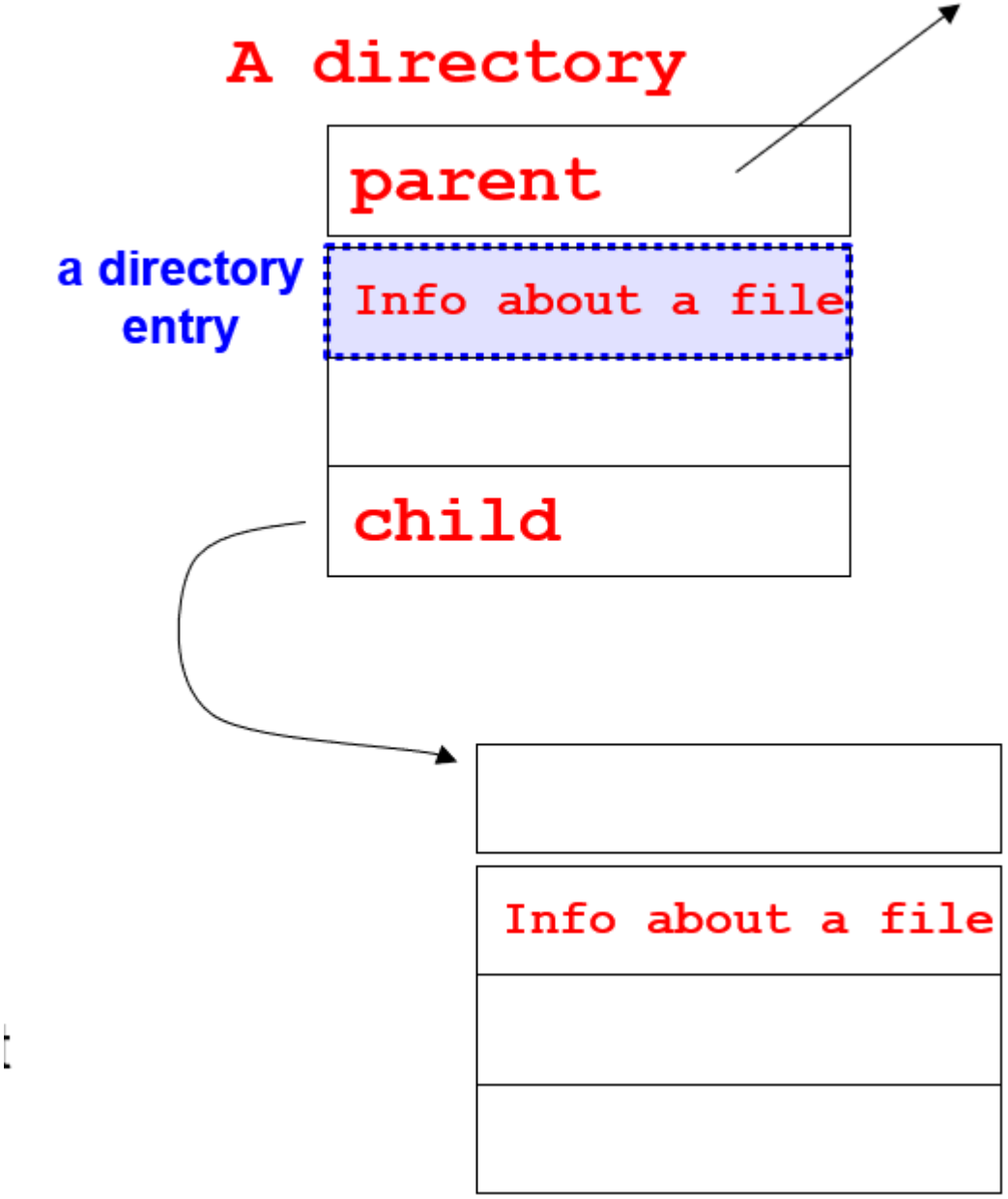
Directory는 파일에 대한 Logical information을 관리해야 한다.

- File name, type, location ..
- 이 정보들은 disk에 저장되어 있다.

그럼 이 Logical information (meta data)를 어디서 관리할까?

→ **Directory가 directiry entry에 저장하여 관리**

- File에 접근하면 그 데이터가 Disk의 어디에 저장되어져 있는 지 (metadata에 포함된 정보)를 알아야 한다.
- 이때, Meta data를 directoru에서 저장하도록 하면, **File 접근 시에 File이 저장된 directory만 알련 된다.**
- **Directory에서 여러 파일의 metadata를 저장하게 하고, 원하는 파일의 meta data를 찾아 읽을 수 있도록 한다.**
- **상위 Directory는 하위 Directory가 알 수 있는 모든 위치 정보**를 알 수 있다.
    - 이 때문에 Root부터 파일 탐색하는 것이 가능하다.



- Directory는 다음과 같이 구성된다.
  - Parent, child directory를 가리키는 `pointer`가 존재한다.
  - 한 Directory는 자신의 하위 directory가 갖는 file의 metadata 역시 전부 확인할 수 있다.
- `Directory entry` : OS가 metadata를 저장하기 위해 정한 자료구조
  - OS가 정했기 때문에 임의로 Metadata 추가가 불가능 하다.
  - 한 directory entry는 파일 개수만큼 존재해야 한다.

일반적으로 `UNIX 계열의 OS`는 **directory 역시 하나의 file**처럼 취급한다.

- file인데 그 file의 컨텐츠가 그 directory 내에 존재하는 다른 모든 file의 정보인 file
- Directory를 찾는 방식도 file을 찾을 때의 방식과 동일하다.

하지만 문제점이 있다.

1. fscanf()로 어떤 파일의 데이터를 읽을 때, 이를 위해 파일의 metadata가 저장된 directory로 이동해야 한다.
2. 해당 directory에서 directory entry로 data의 위치 정보를 알아야 한다.
3. 이 경우, 만약 fscanf()가 loop처럼 돌아 같은 파일의 다른 데이터 접근해야 하는 경우에도 같은 작업을 반복해야 한다.
    - 즉, 같은 파일임에도 불구하고, 접근 시마다 I/O가 발생한다.
    - `fopen` 을 통해 해결할 수 있다.
        - **open 하는 파일의 meta data를 메모리에 올려서 또 접근했을 때, I/O가 발생하지 않도록 한다.**

하지만, 이처럼 file의 metadata를 메모리에 올려 사용하게 되면, Disk에 있는 metadata와 일관성을 유지해야 한다.

- Memory에 있는 metadata가 업데이트 되었을 때, harddisk의 원본과 일치하도록 보장해야 한다.
- Memory에 있는 metadata 업데이트마다 disk도 업데이트하면 I/O가 많이 발생한다.
- Memory에서 metadata가 쫓겨날 때 업데이트 하는 방식을 사용할 수 있다.
    - File system은 이때의 시간차를 고려해야 한다.

추가적으로 보완할 수 있는 부분이 있다.

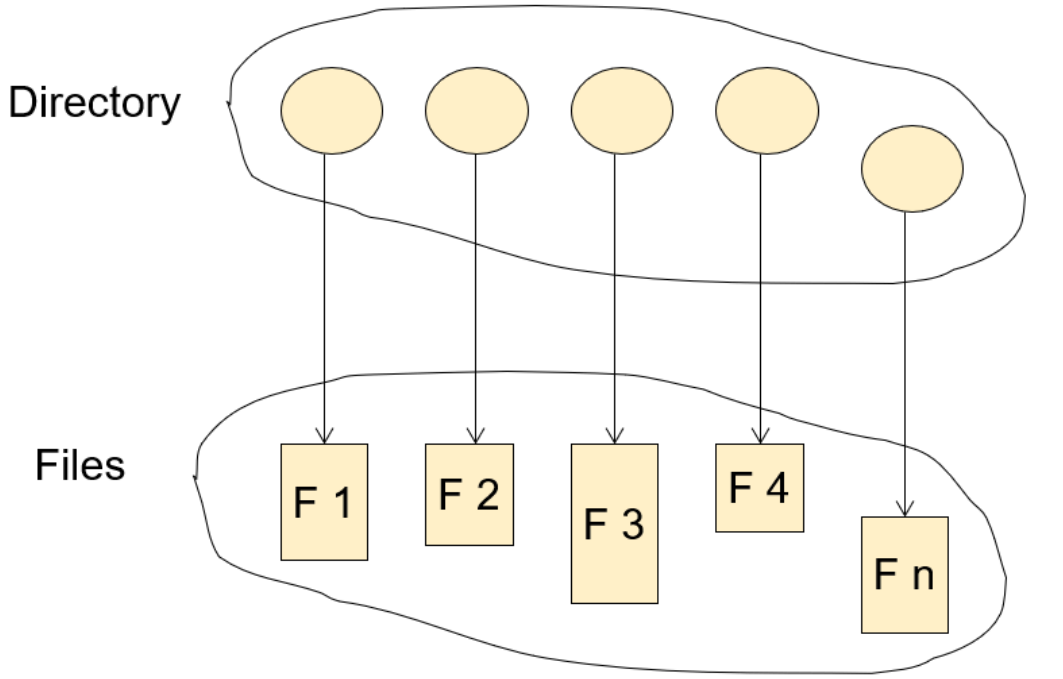
1. fopen()으로 파일의 메타데이터를 메모리에 올려 사용하다가, 파일 사용이 끝나 close를 호출해 metadata를 메모리에서 내보낼 수 있다.
2. 하지만 이 경우, 다시 해당 파일을 접근해야 할 때, I/O가 발생한다.

3. Linux는 이를 방지하기 위해 `Directory cache` 를 사용한다.

Directory cache를 사용하게 되면 Harddisk에 있는 meta data와 consistent를 유지해야 한다.

Directory sturcture (일반적인 경우)

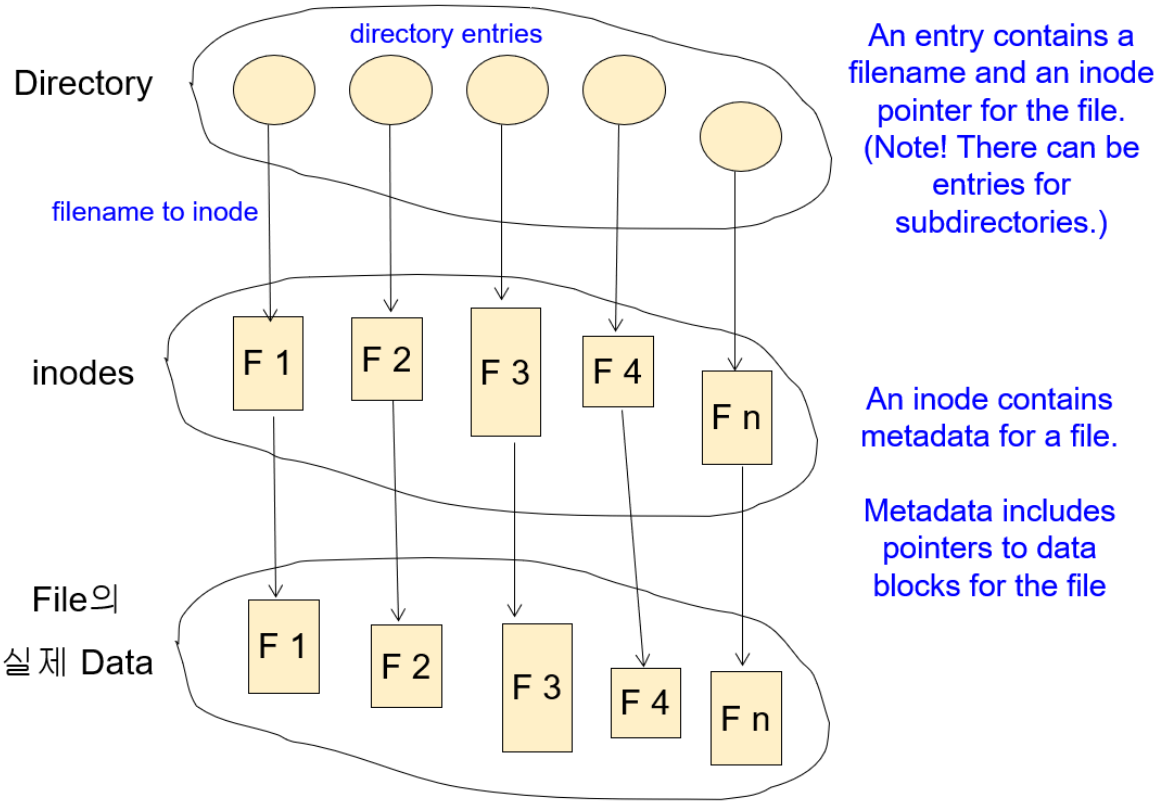
Directory는 파일에 대한 정보를 담고 있는 node의 집합이다.



Both the directory structure and the files reside on disk

- node가 directory enrty이다.
- 구현 방식은 OS마다 다르다.
- Directory 입장에서는 S**ub directory도 하나의 file처럼 인식하므로 sub directory를 위한 directory enrty가 필요**하다.
- 하나의 directory 내에 많은 directory entry가 존재하게 된다.
  - **우리는 이를 Grouping 하여 연속적으로 관리할 필요가 있다.**

Directory structure (UNIX)



- 우리가 Directory에서 파일을 찾기 위해서 사용해야 하는 metadata는 `name` 뿐이다.
- 이 경우, **Directory entry는 filename 만을 관리하게 되어 entry size가 작아진다.**
  - **작은 Directory entry를 연속적으로 모아 관리하므로 search하는 시간도 절약**된다.
- `name` 을 제외한 다른 metadata는 `inode` 에 저장하도록 한다.
- `inode` 는 다른 metadata와 실제 file block을 가르킨다.

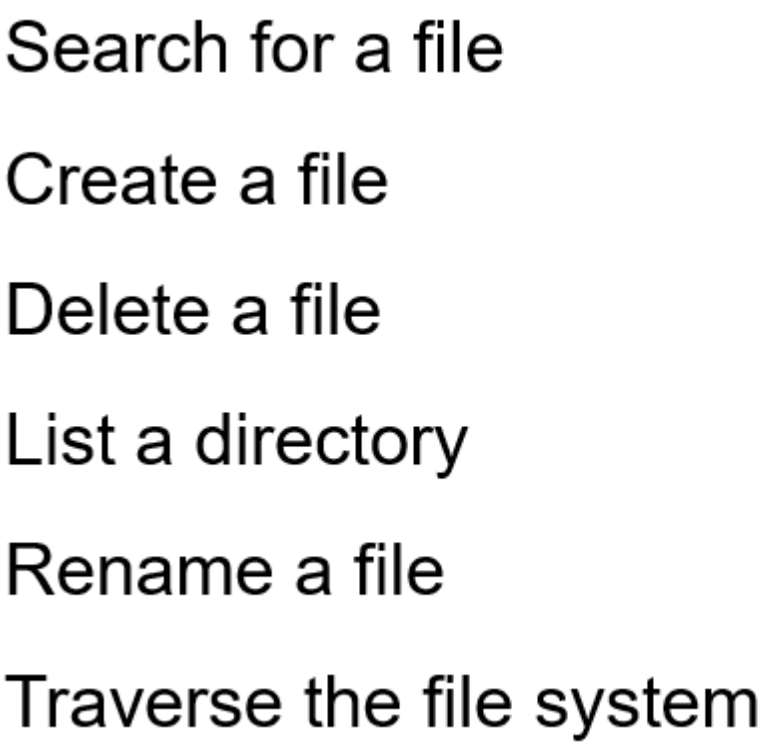
Directory Implement

1. **Linear list of file names with metadata | pointers to inodes (UNIX)**
   - 구현이 쉽다

- Linear search (O(n)) = Time consuming (Overhead)

2. **Hash Table**
    - Linear array 구조는 유지하되, 별도의 Hash table을 추가
    - Hash table은 filename → file을 가리키는 pointer
    - Search time = O(1)
    - `collision` : 자료구조 때 배운 방법으로 해결

Directory에 사용할 수 있는 명령

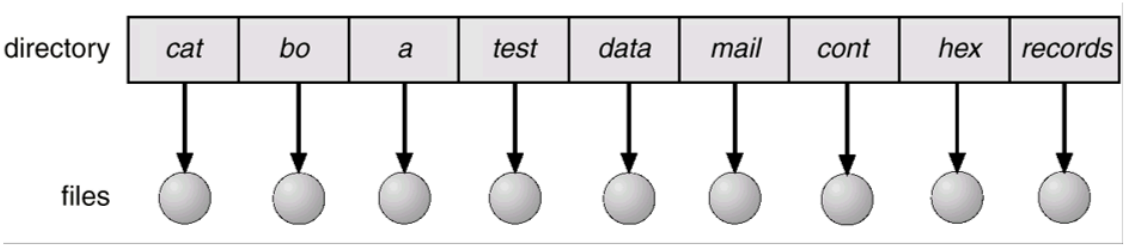


- Searching은 빨라야 한다.
- File 생성 = 새로운 Directory entry
- File 삭제 = Directory entry 삭제
- List a directory: directory 속 모든 파일 확인 (ls)

Single level directory

- **Directory가 없고, Root 아래에 모든 파일이 존재**
- 같은 이름의 파일 존재 불가

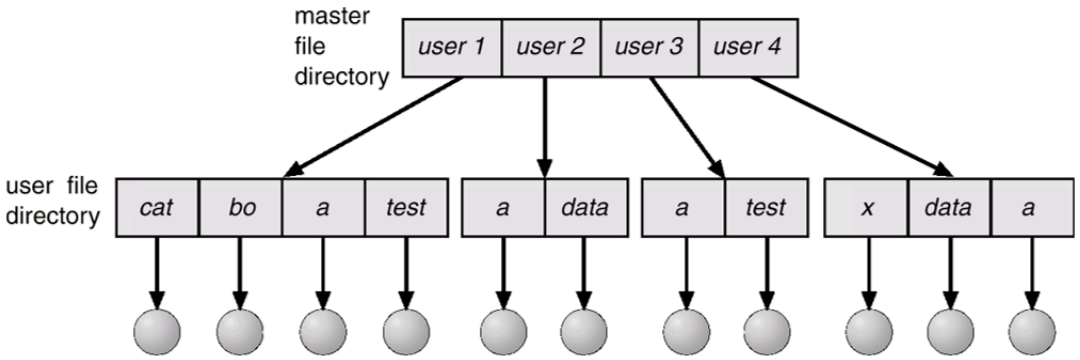
- A single directory for all users



- 같은 이름은 충돌하고, UNIX의 경우 255개의 이름 사용 가능
- Grouping probelm : 관련 있는 파일을 묶을 수 없다
- 실제로 사용하지 않음

Two-level directory

- Separate directory for each user



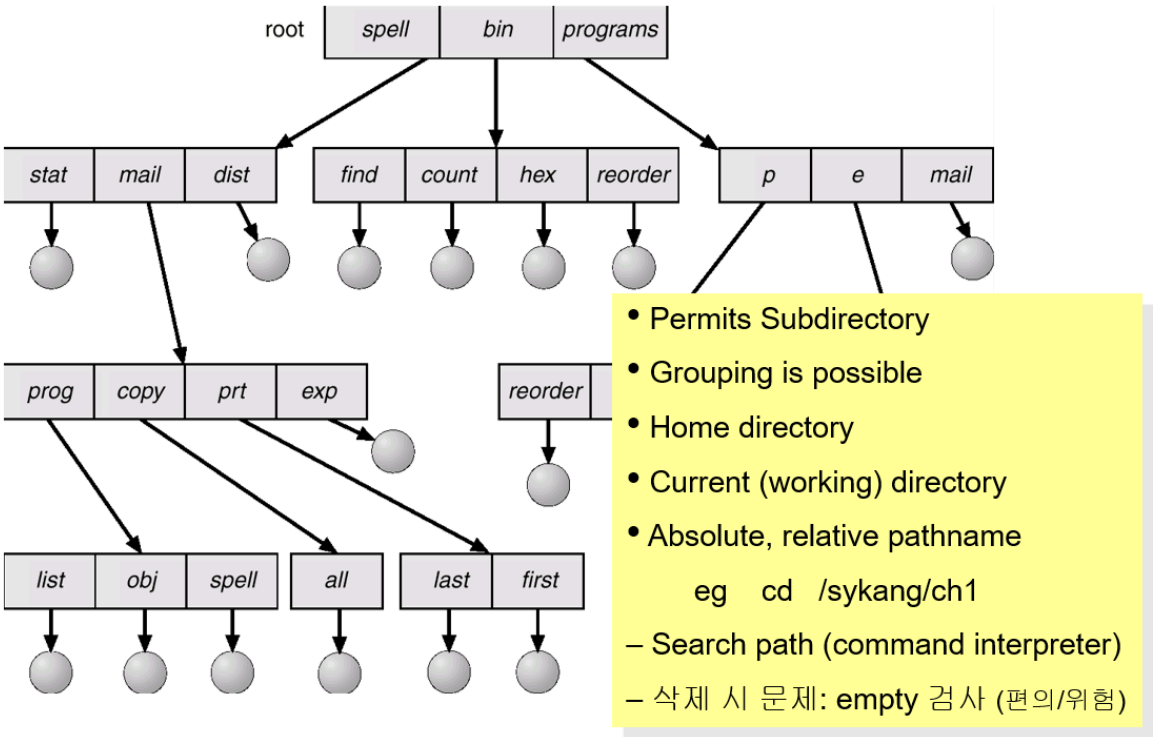
- 각 User 마다 각 directory를 갖는다.
- Pathname

- 다른 사용자끼리는 이름 충돌이 발생하지 않는다.
- 한 User 입장에서는 여전히 하나의 file이다.
    - Naming, Grouping 문제 발생

Level을 한정 짓지 말고, Tree 구조로 만들어보자

Tree-structered Directories

- 어느 Directory 던지 sub directory를 생성할 수 있게 하자.



문제점)

1. **Direcory의 수가 너무 많아진다.**
    - 특정 파일을 찾을 때, Search time이 너무 오래 걸린다.
    - Search path 를 이용하여 해결할 수 있다.

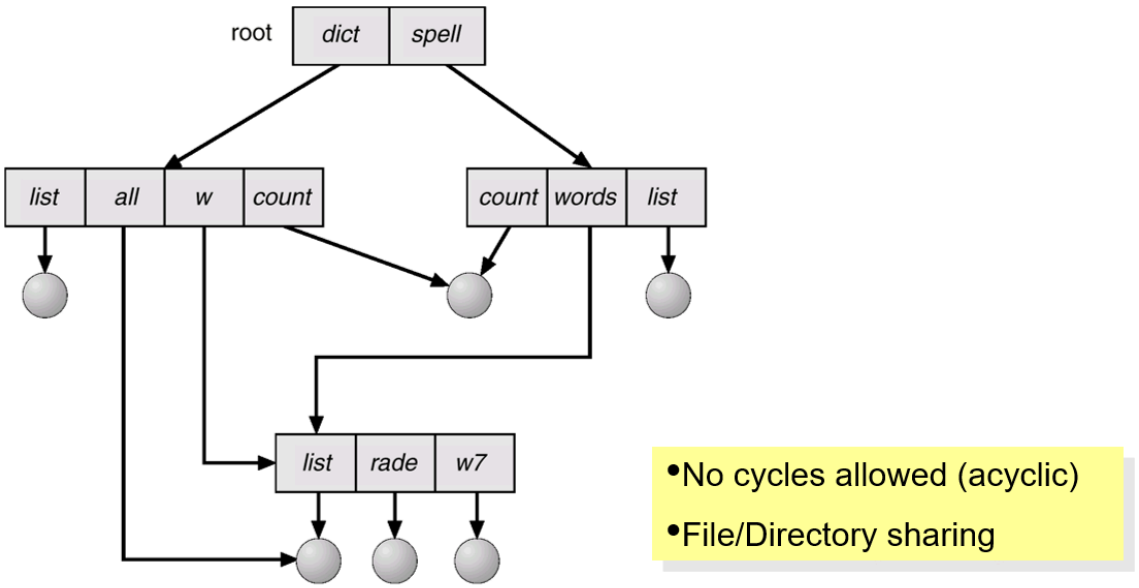
    - 어떤 file name이 주어졌을 때, 알아서 그 경로를 찾아서 실행하도록 위치를지 정해 놓은 것

2. **Sub directory를 삭제할 때, Sub directory 밑에 있는 file과 그 Sub directory의 Sub directory도 전부 삭제해야 하는가?**
    - 정책적인 문제
    - Directory 삭제 시, Directory 내부에 File이 존재하지 않는 경우에만 삭제할 수 있도록 한다. (안정성)
        - 하지만 이 경우, 특정 directory를 삭제하고 싶은 경우에 밑에서 부터 삭제해야 한다. (편리성이 낮다)
        - 편리성과 안정성이 경쟁

하지만, Tree는 Parent node가 반드시 하나여야 하므로 다른 directory간의 파일 공유가 불가능하다.

Acyclic-Graph Directories

Graph 를 사용하면 **다른 directory에서 directory나 file을 공유**할 수 있다.

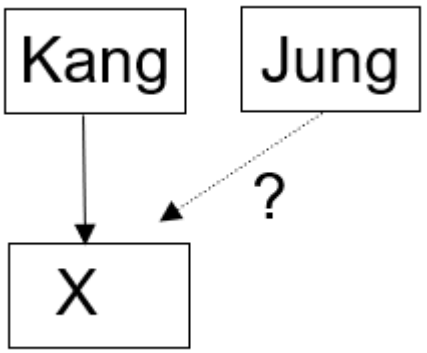


- 중간에 공유되는 것을 확인할 수 있다.
- **공유되는 상황에서는 Copy 대신에 pointer로 가리키도록 한다.**

- **Cycle이 발생할 수 없다**

하지만 이 방법은 아래와 같은 문제점이 있다.

1. Traverse 할 때, 공유되고 있는 같은 node (file)을 2번 이상 반복 접근하게 된다.
2. 삭제 시, 문제가 발생한다.



1. **Kang이 x라는 directory를 만듦**
    - Kang에 x를 가리키는 directory entry가 생성
2. **Jung이 x를 공유**
    - UNIX 계열이라 가정, link 한다.
3. **Kang이 x를 삭제**
4. **Jung은 x에 접근**

위 경우에 `link` 를 생성하는 두 가지 방법이 있다.

1. `Symbolic link`
    - 위 경우에, Jung에는 Jung directory 밑에 X directory의 pathname(root부터)을 Jung 밑에 등록한다.
    - Jung은 새로운 directory entry를 생성하는 대신, Path name을 등록한다.
    - 하지만, **위 경우에서** `Dangling reference` **link**가 된다.
        - 아무 정보도 없는 공간을 가리키는 포인터

2. `Hard link`
    - Kang이 가지고 있는 directory entry를 그대로 Jung에 복사
    - 이 경우에는 Kang이 삭제해도 Jung은 접근할 수 있다.
    - 하지만 이런 경우에는 다음과 같은 문제점이 발생한다.
        1. 그럼 언제 삭제해야하나?

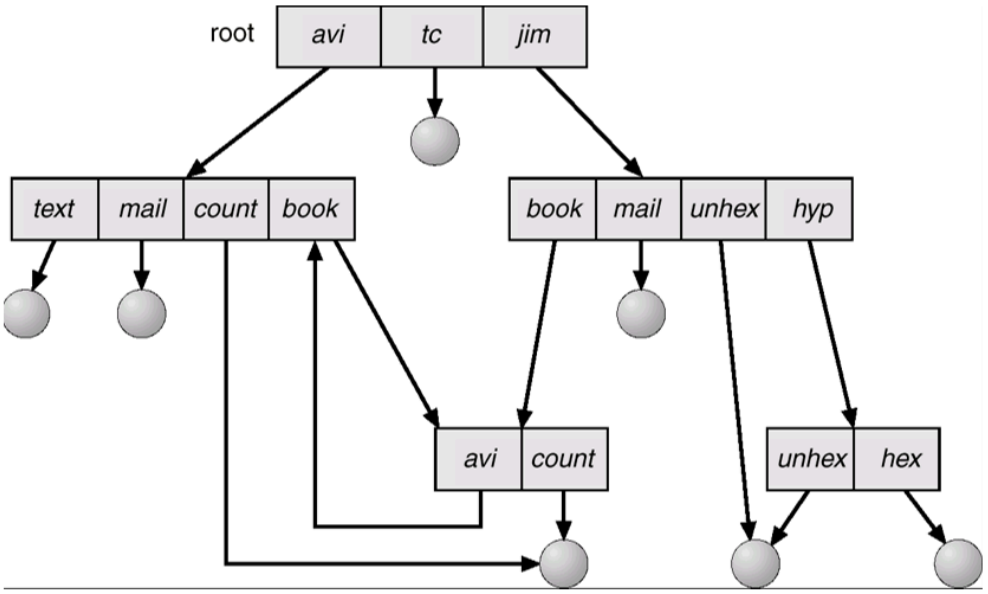
a. Ref count 같은 것을 따로 관리하며, 아무 것도 해당 file을 사용하지 않는 경우에만 삭제할 수 있도록 해야 한다.

b. Kang이 Update한 값은 Kang의 directory entry에만 적용된다.

- Consistency problem
- OS가 따로 해결해야 한다.

General Graph Directory

- Cycle 을 허용한다.
  - 하위 Directory가 상위 directory를 자신의 하위 directory로 가지고 있을 수 있도록 한다.
- General 하지만, 복잡하다.
- 이 구조를 적용하는지 여부는 OS가 결정한다.

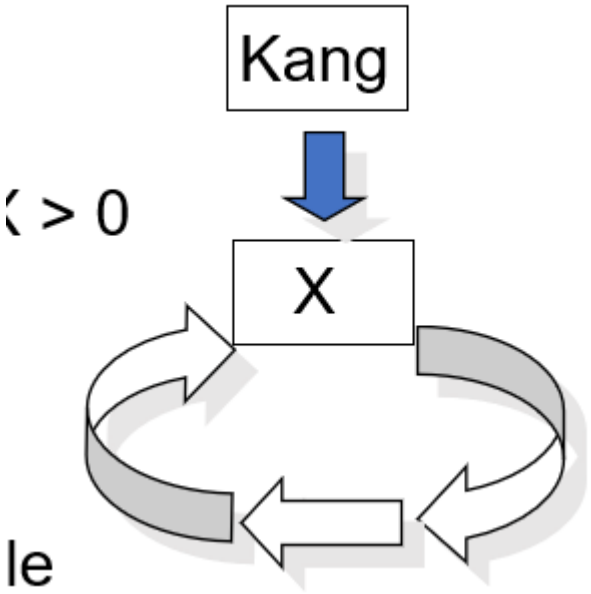


Traverse와 delete에 복잡한 알고리즘이 필요하다.

- 전체적으로 모든 연산이 복잡해진다.

`Traverse` 의 경우에는 Cycle에 의해 Infinite loop가 발생할 수 있다.

`Delete` 의 경우에는 심한 문제가 발생한다.



이 경우를 생각해보자. `Hard link` 를 사용한다고 가정한다.

위 상태에서 Kand이 X에 대한 directory entry를 삭제해도, X의 ref count = 1 이기 때문에 삭제되지 않는다.

하지만 우리는 Root부터 파일에 접근해야 하는데, Kang → X로의 경로가 끊겨 X에 대한 접근이 불가능해지게 되었다.

- **이는 심각한 메모리 낭비이다.**

우리는 이러한 접근 불가능한 데이터를 지워야 한다.

- `Garbage collecting` 은 이런 접근 불가능한 데이터를 지운다
- 하지만 시간이 너무 **오래 걸린다는 단점**이 있다.

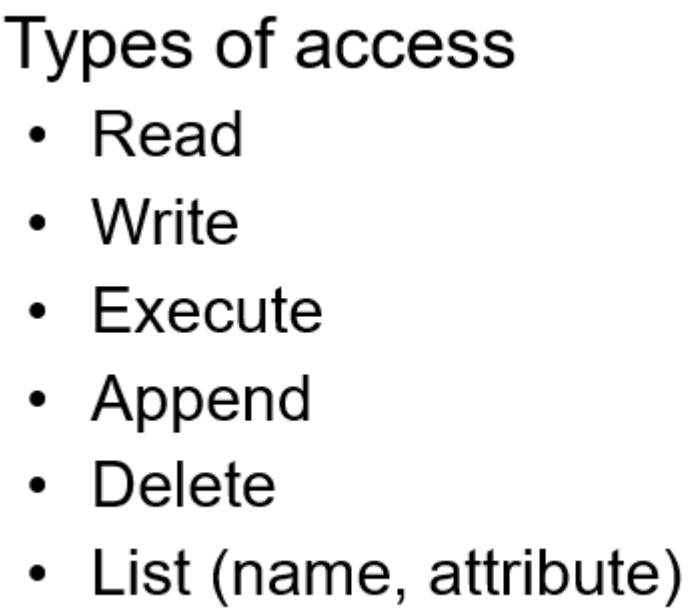
따라서 Cycle을 허용하는 경우 Operation이 너무 복잡해진다.

- 따라서 보통 `Acyclic` 하게 만든다.
- `Acyclic` 을 위해선 사용자가 Link 시마다, Cycle detection을 해야한다.

- Cycle이 생성될 시, OS가 Link를 실패하게 만들도록 지원한다.

Protection

- **어떤 file을 누가 어떻게 접근할 수 있는지 권한을 설정한다.**
- 권한 설정은 `해당 파일의 소유자`가 한다.



- 이 권한 중 어떤 접근을 할 수 있는지 각 사용자마다 부여한다.
- 크게 **RWX 세 가지**로 나눈다.

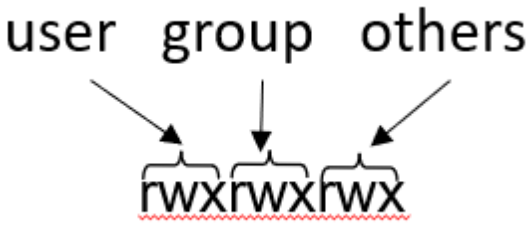
권한 부여는 어떻게 할까?

`Access control Matrix`

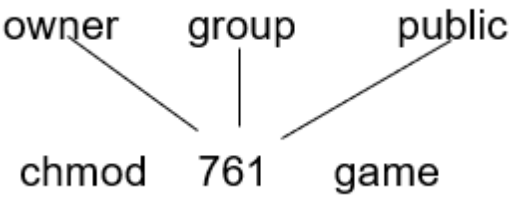
- Row: file 이름, Col: 사용자 이름
- 너무 크다

UNIX는 다른 방법을 이용한다.

사용자를 크게 **Owner / Group / Others 로 나누어서** 아래처럼 권한을 부여한다. Binray bit로 표현한다.



			RWX
a) owner access	7	⇒	1 1 1
			RWX
b) groups access	6	⇒	1 1 0
			RWX
c) public access	1	⇒	0 0 1



- chmod 명령어로 권한을 변경할 수 있다
- 761 → 10진수로 game이라는 파일에 대한 각 권한을 부여한다.

chgrp G game: game이라는 파일이나 directory가속한 group을 G로 변경한다.

File System Stucture

`File system` 은 보통 Disk에 저장된다.

- **File과 directory를 관리하기 위한 File system이 접근할 수 있는 형태로 모든 정보는 disk에 저장**된다.
- File system은 일종의 `OS kernel code` 이다.
- **Layer 구조**로 되어있어 file 한 번 접근시 이 계층 구조를 거쳐서 마지막에 DIsk로 이동한다.
  - 이 과정에서 굉장히 많은 function call이 발생한다.
  - Layer 하나씩 이동할 때마다 function call 발생한다.
  - File system은 굉장히 무겁다.

`Open()` System call

- FIle system은 너무 무겁다
- File system을 거치지 않기 위해 open() 사용한다.
- open()은 접근하고자 하는 file의 meta data를 memory에 올린다.
- file system의 계층 구조를 pass하고 데이터가 있는 곳으로 바로 접근하겠다는 의미이다.

`Open file table` : Meta data가 메모리에 올라올 때, 메모리에 저장하는 공간

`File descriptor` : Open file table의 index로 사용

- file pointer라고 생각해도 된다.
- open()시 file decriptor 값이 받아와짐

Directory search가 너무 오래 걸리고, Disk에는 너무 많은 파일이 존재하기에 이런 구조가 필요하다.

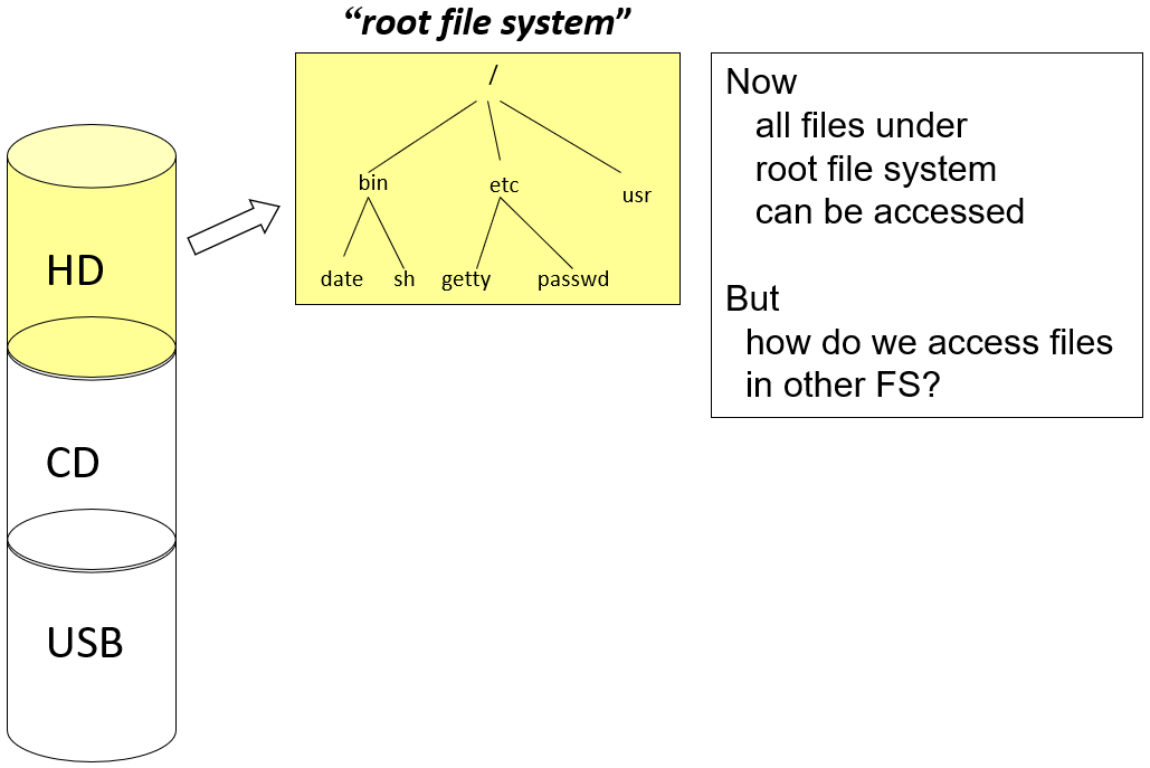
Mounting file system

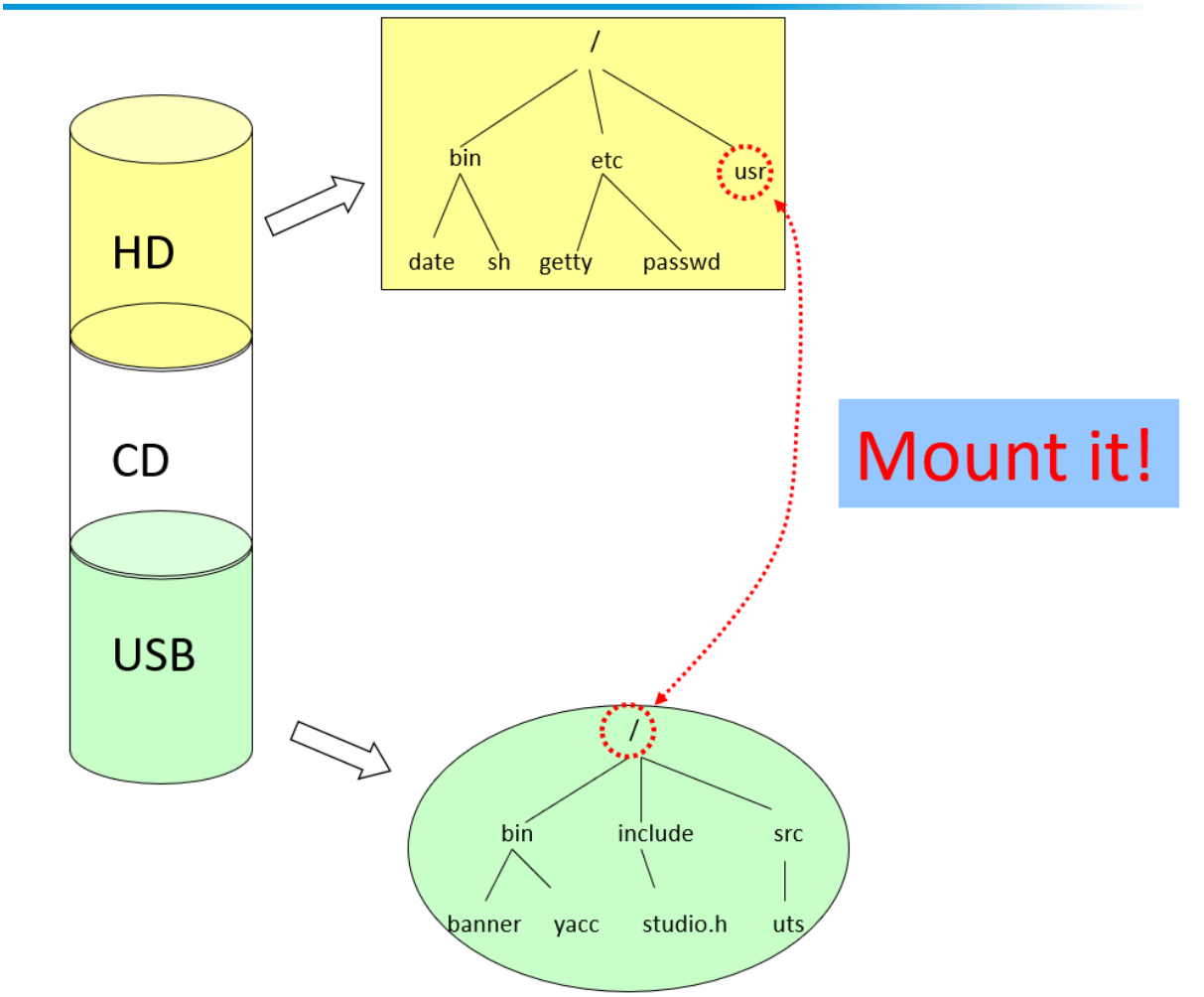
기존에는 하나의 시스템에 여러 저장 장치가 존재할 때, 각 저장 장치는 자체적으로 root에서 시작하는 directory를 가졌다.

- USB에 있는 root / / 를 접근하고자 하면, 별도의 USB driver가 생기고 그 dirve로 이동해 root를 찾는다.

각 drive가 독립적으로 사용되다가 다른 drive에 있는 데이터를 얻으려면, 우선 그 dirve로 이동해야 했다.

다른 방법은 없을까? → Mounting!





Drive를 변경할 필요가 없고, 모든 File system은 drive의 종류와 관계 없이 HD의 root에서 시작해 접근이 가능하도록 한다.

- 하나의 거대한 file system에서 모든 `secondary device` 를 관리
- Device와 무관하게 단일한 view를 제공한다.

`Mount` : 다른 device의 root directory를 primary storage가 갖는 file system이 관리하는 dirctory의 하위directory로 붙인다.

Allocation of File data in disk

- File data를 HD에 저장하기 위해 HD 저장공간을 어떻게 관리할까??
- File data를 Disk에 저장하는 것은 `Block` 단위이다.
- 결국 `Block` 을 어떻게 할당할 지가 문제이다.

`Block`

- 4KB = 512 byte sector 8개
- 이건 default 값이고, Format할 때 유저가 사이즈 지정이 가능하다.

Second storage는 block 단위로 동작하는 이유

- 느리기 때문에, Block 단위로 동작하도록 하여 속도를 높인다.

Allocation method

1. Contigous Allocation

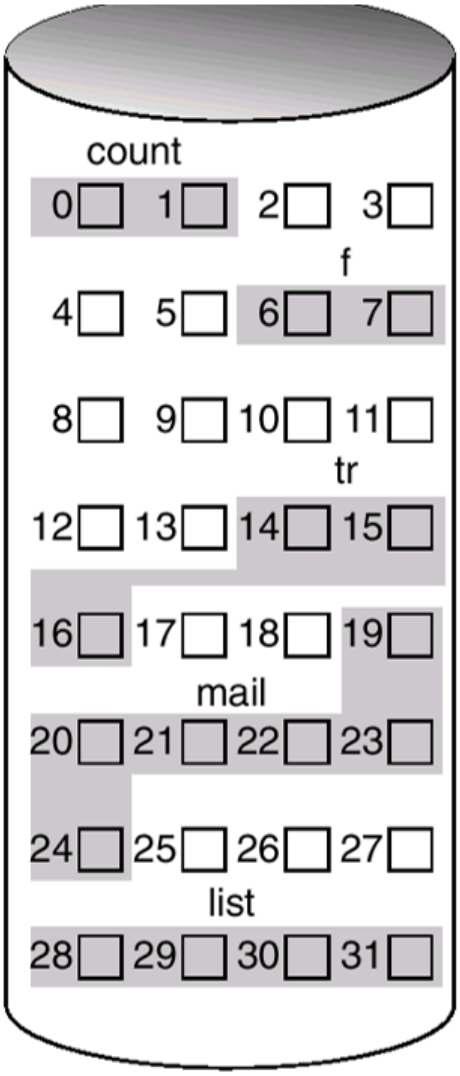
장점)

1. 첫 번째 Block의 시작 위치만 알면 전체 Data 접근이 가능
    - 첫 번째 데이터는 첫 번째 block의 첫 byte이다.
    - 4KB인 경우에는 다음 Block으로 넘어간다.
2. 구현이 간단하다.
3. `Fast I/O`
    - Disk에서 메모리를 읽을 때, 이 방법은 연속적으로 저장되어 있기 때문에 처음 파일 위치로 이동하기 위해 head를 딱 한번만 움직여도 되도록 한다.
    - 실시간 경우에 유용하다.

단점)

1. `External fragmentation` + `Dynamic storage allocation`이 발생 → 메모리 공간 낭비

2. File 크기를 늘리는데 문제가 있다.
    - File을 처음 생성할 때의 얼마나 큰 Hole을 배정해야 하는 지에 대해 문제가 생긴다.
    - 나중에 Grow를 가능하게 하려면 크게 할당 vs Internal fragmentation 방지
    - 이를 방지하는 방법으론 일단 딱 맞게 할당 후, 나중에 더 큰 위치로 옮기는 방법이 있다.
        - 그러나 I/O 발생



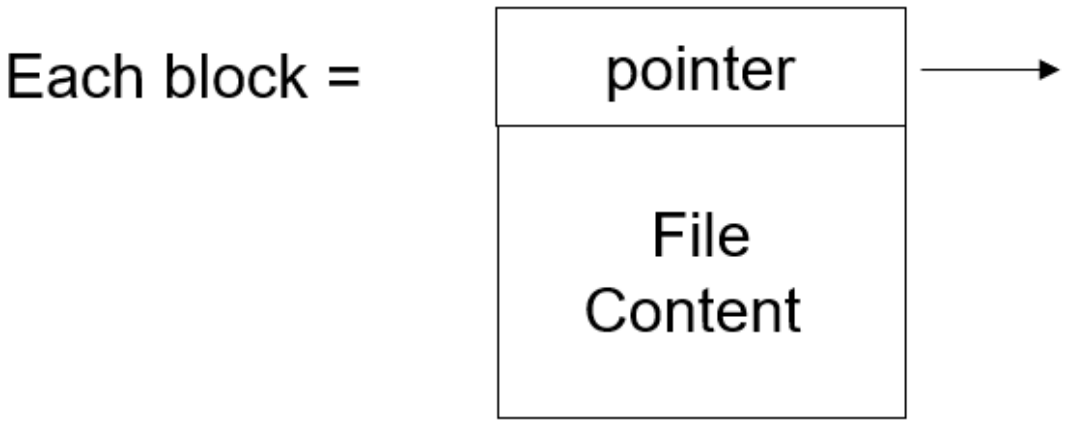
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- `start` 와 `length` 만을 관리하면 된다.

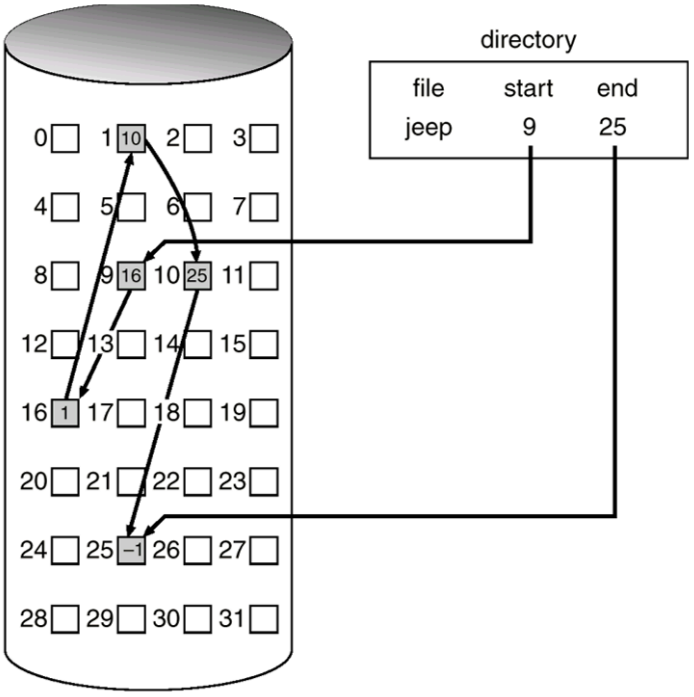
2. Linked Allocation

contigous의 파일 크기 증가가 어려운 문제를 해결하자



- 각 file은 disk block의 `linked list`로 관리된다.
  - File의 Data를 저장하는 block을 linked list로 관리한다.
- block은 디스크에 어디에나 **흩어져서 존재**한다.

- Allocate as needed, link together
  - e.g., file starts at block 9



- 하나의 파일에 대해 `start` , `end` 만 관리하면 된다
- File이 커지면 end block에 Next block pointer만 적어주면 된다.
- 작아지면 기존 포인터를 변경

장점)

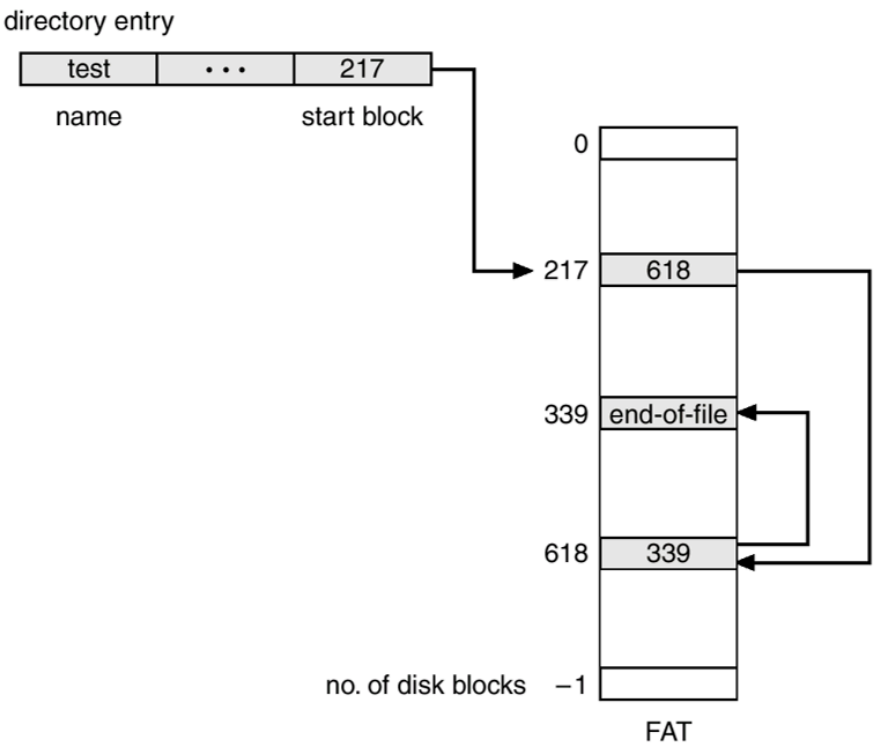
1. Start address만 관리하면 된다.
2. 메모리 공간을 자유롭게 관리하고, 낭비가 없다.

단점)

1. **random access가 불가능하다.**
   - 중간 block에 있는 경우에는 처음부터 시작해서 그 block까지 이동해야 한다.
   - 접근 시마다 I/O 발생하기 때문에 효율이 떨어진다.
2. 매 Block을 움직일 때 마다 Head가 움직여 **Disk I/O 효율이 떨어진다.**
3. `Reliablility` 문제
   - Bad sector로 인해 중간에 pointer가 유실되면 많은 부분을 읽는다.
4. 중간에 Block 내에 Pointer를 저장할 공간이 필요하다.

3. File - Allocation table (FAT)

공간 효율성 측면에서만 보면, Linked allocation 이 가장 좋지만, 단점이 명확하기 때문에 사용한다.



- Faster random access

- 특정 데이터를 찾기 위해 Block을 쫓아가는게 아니라, Memory 상에서 table을 돌며 찾을 수 있도록 한다.
- `Start block` 을 만들어 FAT table에서 첫 번째 로 확인할 위치를 가리킨다.

장점)

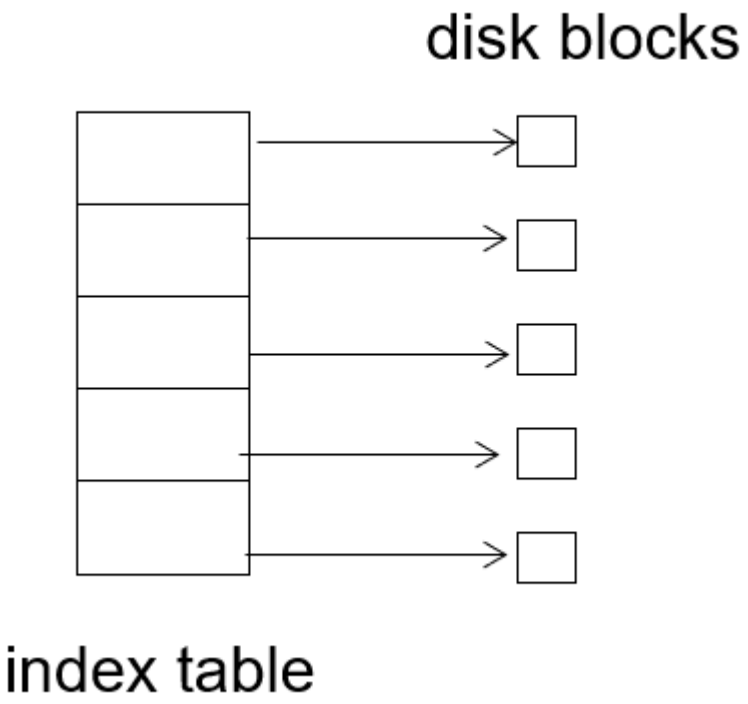
1. Random access 가능
2. DIsk I/O 감소

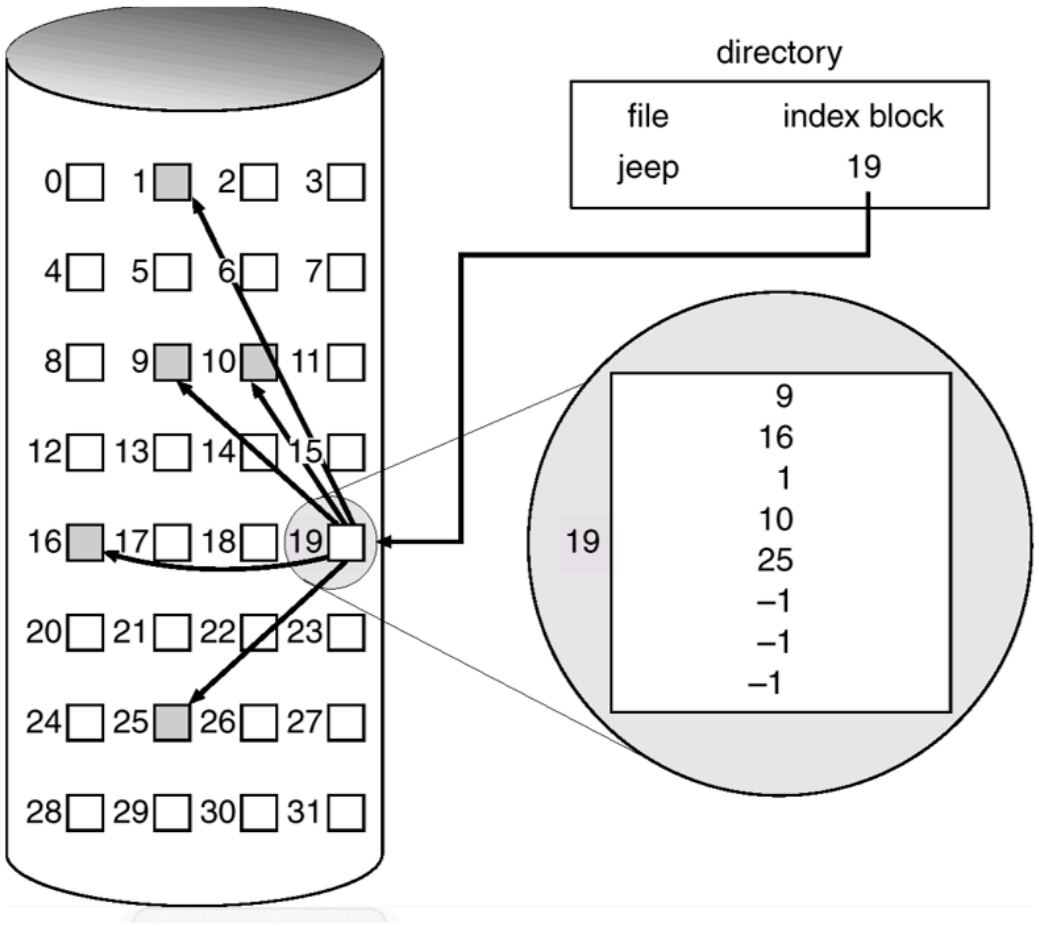
단점

1. FAT은 kernel이 유지하는 자료구조이다.
2. FAT가 저장되어 있는 Disk block이 깨지면, 모든 file system이 날라간다.
    - 이를 방지하기 위해 3개의 copy 본을 만들고, 3개 중 하나가 깨지면 나머지 2개를 이용해 새로운 하나를 다시 생성
    - 하나의 FAT이 바뀌면 다른 FAT에 반영해야 한다.

Indexed Allocation

- 각 file은 자신만의 `index block` 을 갖는다.
- `Index block`은 모든 disk block에 대한 pointer를 갖는 배열이다.
- Context 대신 위치 정보를 저장한다.
- 해당 부분만 읽으면 모든 파일의 위치 정보를 확인할 수 있다.
    - Random access가 가능하다.





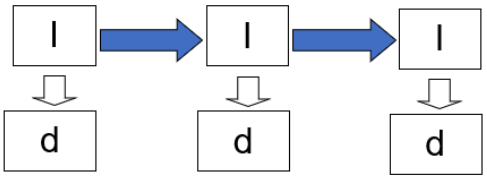
각 file마다 하나의 index block만을 사용하면 생기는 문제점

- block은 4KB이다.
- 즉, 4 Byte pointer 1024개를 가질 수 있다.
- 각 pointer에 의해 가리켜지는 block 역시 4KB 이다.
- **따라서 하나의 indexed block으로 얻을 수 있는 파일 저장 공간은 4KB *1024 = 4MB 이다.**

4MB 파일을 저장하는 것은 부족하다/

우리는 Index block을 늘려야 한다

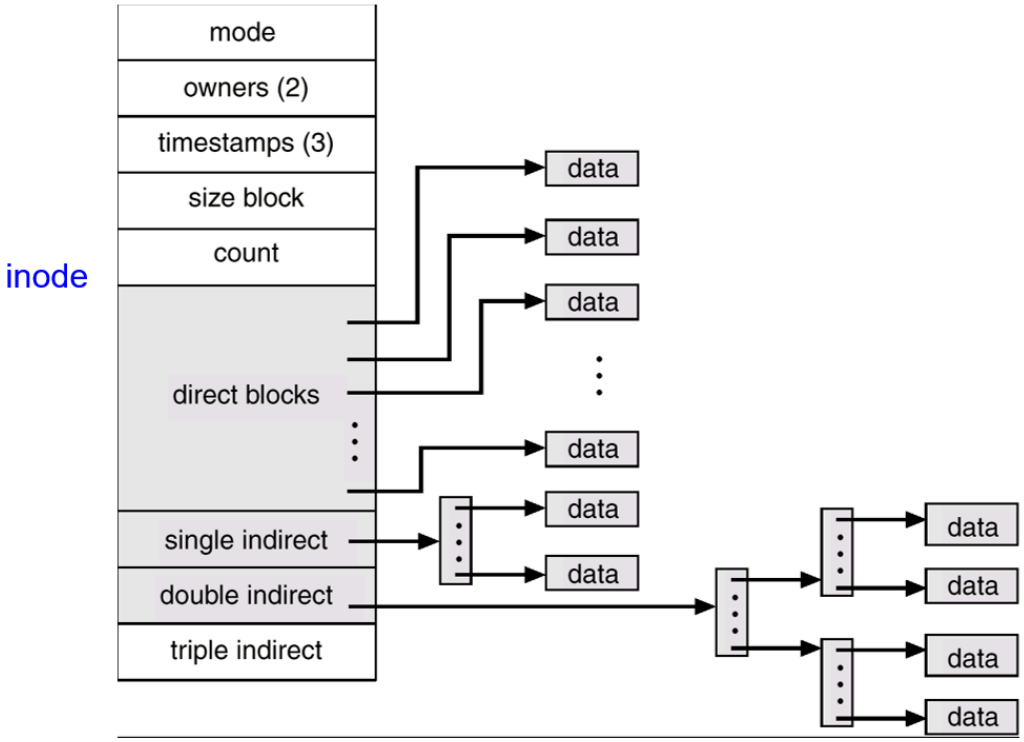
1. Linked scheme – Link blocks of index table (no limit on size)



- Index block을 linked list로 관리하는 방법이 있다.
- 그러나 이 역시 중간에 block이 깨지면 파일 정보가 유실된다는 단점이 있다.
- 이 경우에는 한개의 pointer는 다음 index block을 가리켜야 하므로, 1023개의 data 만을 저장할 수 있다.
- File size에 제한이 없다.

2. Two level index
    - index block을 2-level로 한다.
    - 1024 * 1024 개의 데이터 block을 가질 수 있다.
    - 총 2^20 개만 가질 수 있는데 이 또한 file size의 제한이 생긴다.

3. Combined

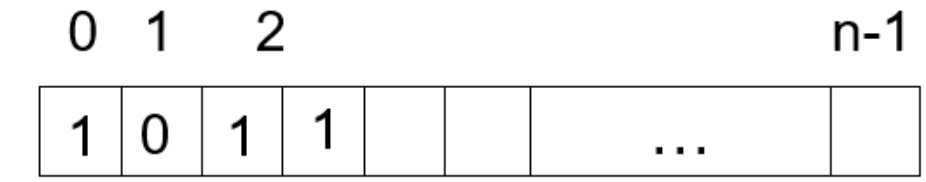


- inode는 UNIX 계열의 OS가 file meta data를 자료구조이다.
- direct block 이 data block을 직접 가리키는 pointer를 갖는다.
- 밑에는 single indirect / double / triple
- 2^11 + 2^ 20 + 2^30 개의 데이터 block을 갖는다.

Free-space management

- File data를 block에 할당하려면, file system이 어떤 공간을 비었는지 알아야 한다.

Bit vector



$$bit[i] = \begin{cases} 1 \Rightarrow block[i]\ free \\ 0 \Rightarrow block[i]\ occupied \end{cases}$$

이런 bit vector를 bit 단위로 접근한다고 생각해보자 → O(n) 오래 걸린다.

word 단위로 찾아보자 이를 위해선 추가적인 확인이 필요하다.

```
word 0: 00000000 00000000 00000000 00000000   (0-value word)
word 1: 00000000 00000000 00000000 00001000   (첫 1 등장 → offset = 3)
```

이 같은 경우에는 (word 당 bit 개수) * (0-value word) + (first 1-value offset) 으로 계산한다.

장점)

- Bit[i] = 1 인 block이 연속적으로 존재한다면, 연속적인 block이 존재함을 확인 가능
- 연속적인 할당에 유리

단점)

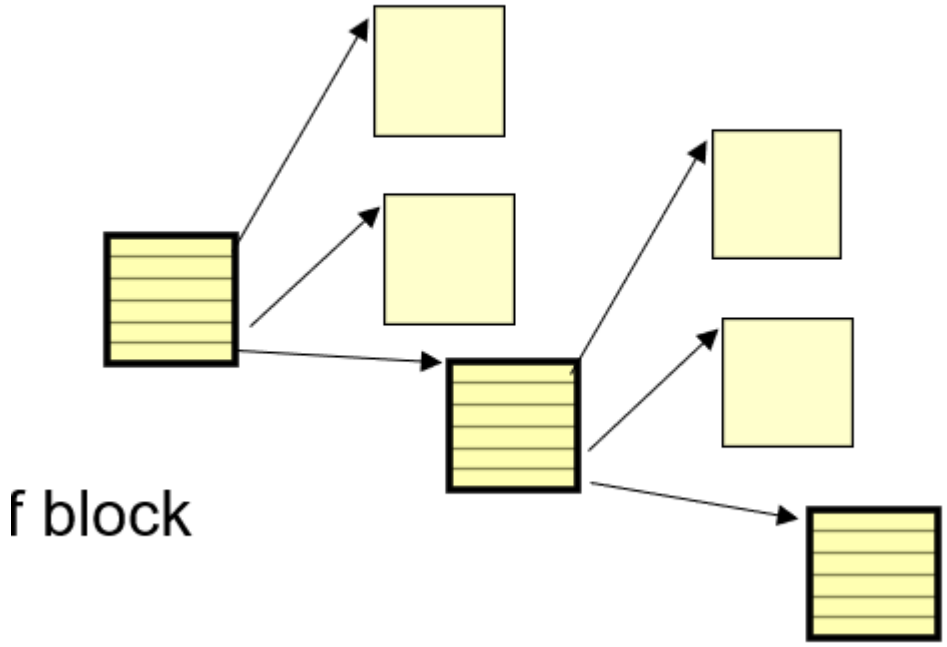
- bit map을 관리하기 위한 Space overhead

위 방법 외에 다른 방법을 살펴보자

1. Linked list
    - free block을 linked list로 관리한다,

- disk I/O 발생
- 빈 공간을 찾기 어렵다
- 공간 활용성이 좋다.

2. Grouping



- free block을 가리키는 pointer를 갖는 block이 따로 존재한다.
- n개의 pointer 중에서 (n-1) 개는 free block, 1개는 다른 pointer를 갖는 block을 가리킨다.

3. counting

- 첫 번째 비어있는 block 부터 몇 개가 그 block에 연속적으로 비어잇는 지를 숫자로 관리한다.
- 일반적으로 몇몇 연속된 block은 동시에 할당되고 동시에 free되는 된다.
- {첫번째 빈 block, 몇개가 연속적으로 비었는지}
- OS는 웬만하면 연속적으로 할당하는 것을 선호 한다.

Performance

Disk cache

- OS가 자체적으로 메모리 중 일부를 Disk cache로 사용

Free behind: 해당 공간에 있는 데이터를 다 사용했어도 따로 바뀌지 않고 그 공간을 나중에 써야할 때 비운다

Read-ahead: 따로 요청하지 않아도 데이터를 미리 가져온다.

Virtual disk

- 메모리의 일부를 디스크처럼 사용
- 전원이 날라가면 데이터가 꺼지기에 주기적인 업데이트가 필요하다.
- Memory에 laod, store 요청

Directory entry cahce

- Directory entry를 memory로

Recovery

File을 다룰 때, 성능을 위해 메모리에 의존하는 경우가 많다.

이때 메모리와 디스크 사이에 일치성 문제 발생

- 메모리만 업데이트 하는 경우

어떤 데이터가 어떻게 바뀌었는 지를 계속 확인해야 한다.

- log를 남기는 방식을 사용할 수 있다.
- Journaling File system : 저널을 읽고 data는 반영, metadate는 반영되지 않았다면 Data에 맞추어 metadata를 수정

메모리는 전원이 꺼지면 날라간다.

정상적인 종료가 되었다는 표시가 없으면 file system을 체크한다.

Formatting

Logical formatting

- Data를 초기화 하는 것이 아니라 자료구조 metadata를 초기화
- 실제 데이터는 남아있는데 날라간 것처럼 보임

Physical formatttng

- 데이터를 초기화