File System

파일 및 파일 시스템

파일은 논리적인 저장 단위로, 관련된 정보 자료들의 집합에 이름을 붙인 것이다. 컴퓨터 시스템의 편리한 사용을 위해 정보 저장의 일괄된 논리적 관점을 제공한다.

일반적으로 레코드 혹은 블록 단위로 비휘발성 보조기억장치에 저장된다.

파일 속성 또는 파일의 메타데이터는 파일을 관리하기 위한 각종 정보들이다. 파일 자체의 내용은 아니다.

파일 이름, 유형 저장된 위치, 파일 사이즈, 접근 권한, 소유자, 시간(생성/변경/사용) 등 파일에 대한 전반적인 정보를 말한다.

파일 시스템은 운영체제와 모든 데이터, 프로그램의 저장과 접근을 위한 기법을 제공한다.

시스템 내의 모든 파일에 관한 정보를 제공하는 계층적 디렉터리 구조이고, 파일 및 파일의 메타데이터, 디렉터리 정보 등을 관리한다.

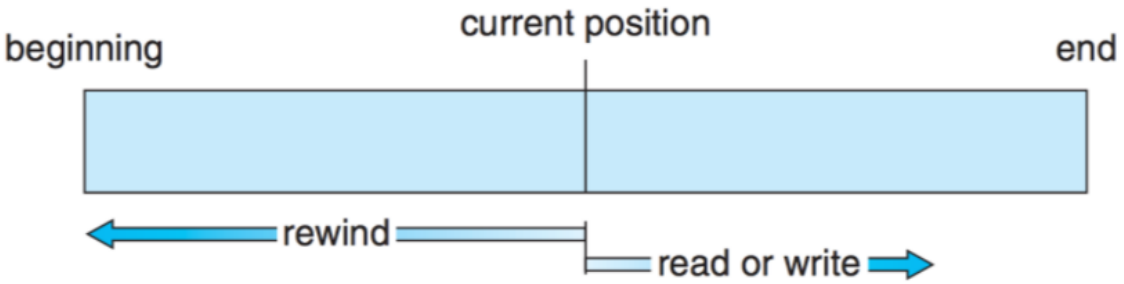
파티션은 연속된 저장 공간을 하나 이상의 연속되고 독립적인 영역으로 나누어 사용할 수 있도록 정의한 규약이다. 하나의 물리적 디스크 안에 여러 파티션을 두는 것이 일반적이지만 여러 물리적 디스크를 하나의 파티션으로 구성하기도 한다.

접근 방법

시스템이 제공하는 파일 정보의 접근 방식으로는 순차 접근(Sequential Access)와 직접 접근(Direct Access, Random Access), 색인 접근(Index Access)으로 나뉜다.

1. 순차 접근(Sequential Access)

   가장 단순한 방법으로 파일의 정보가 레코드 순서대로 처리된다. 카세트테이프를 사용하는 방식과 동일하다. 현재 위치에서 읽거나 쓰면 offset이 자동으로 증가하고, 뒤로 돌아가기 위해선 되감기가 필요하다.



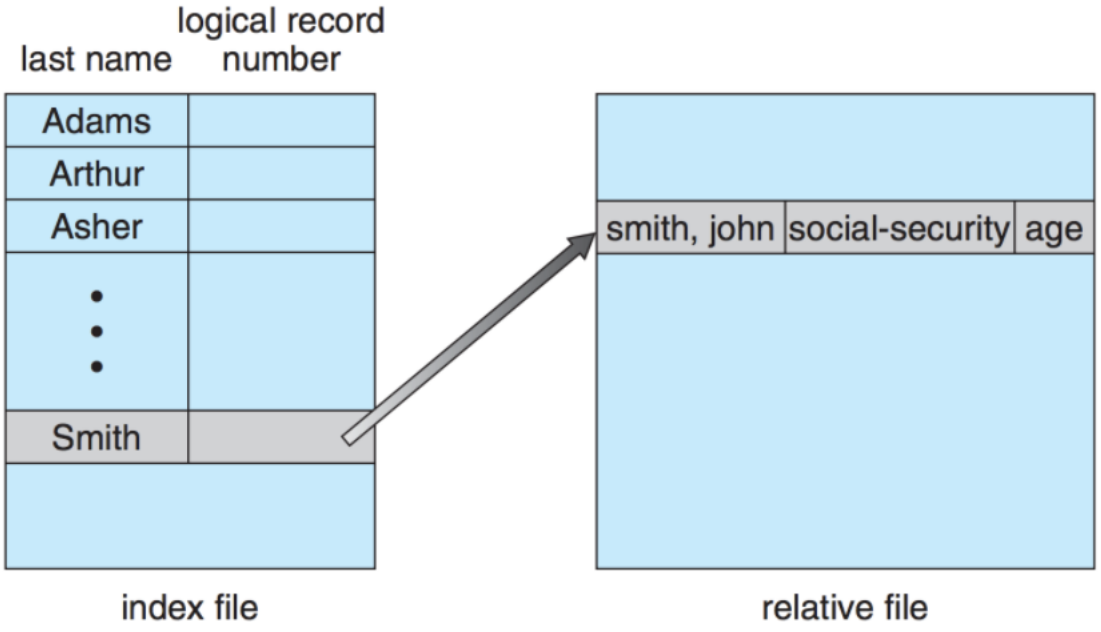
2. 직접 접근(Random Access)

   파일의 레코드를 임의의 순서로 접근할 수 있다. LP판을 사용하는 방식과 동일하다.

   읽기나 쓰기의 순서에 제약이 없으며 현재 위치를 유지할 수 있다면 이를 통해 순차 접근 기능도 구현할 수 있다.

3. 색인 접근(Index Access)

   파일에서 레코드를 찾기 위해 색인을 먼저 찾고 대응되는 포인터를 얻는다. 이를 통해 파일에 직접 접근하여 원하는 데이터를 얻을 수 있다. 따라서 크기가 큰 파일에서 유용하다.



디렉터리

디렉터리는 파일의 메타데이터 중 일부를 보관하고 있는 일종의 특별한 파일이다.

해당 디렉터리에 속한 파일 이름과 속성들을 포함하고 있고, 다음과 같은 기능들을 제공한다.

- 파일 찾기(Search)
- 파일 생성(Create)
- 파일 삭제(Delete)
- 디렉터리 나열(List)
- 파일 재명명(Rename)
- 파일 시스템 순회(Traverse)

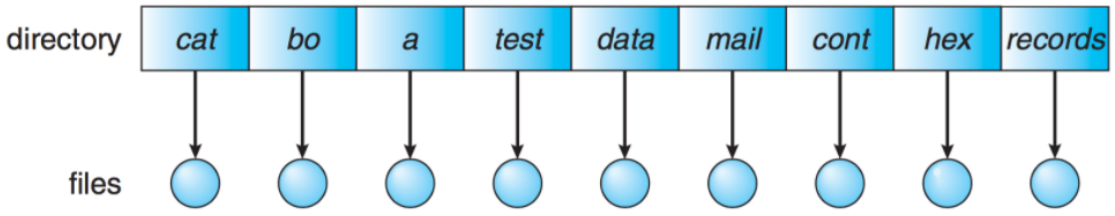
디렉터리는 어떻게 구성되어야 할까?

기본적으로 디렉터리의 파일을 빠르게 탐색할 수 있어야 한다. 또 적절한 이름으로 사용자들이 편리하게 사용할 수 있으면 좋을 것이다. 그리고 파일들을 적절한 분류로 그룹화해두면 사용하기 편리할 것이다. 이를 위해 디렉터리의 논리적 구조를 정의하는 여러 방법들이 있다.

1. **1단계 디렉터리**(Single-Level Directory)

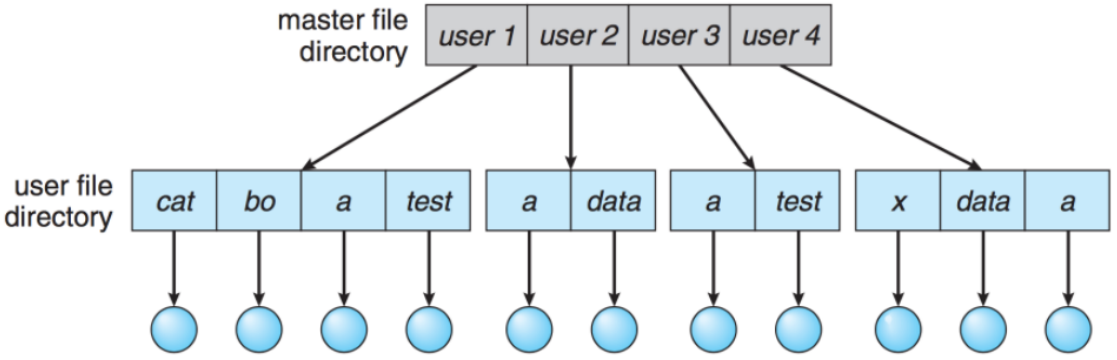
   1단계 디렉터리는 모든 파일들이 디렉터리 밑에 존재하는 형태이다. 파일들은 서로 유일한 이름을 가지고 서로 다른 사용자라도 같은 이름을 사용할 수 있다.

   지원하기도 쉽고 이해하기도 쉽지만, 파일이 많아지거나 다수의 사용자가 사용하는 시스템에서는 심각한 제약을 갖는다.



2. **2단계 디렉터리**(Two-Level Directory)

   2단계 디렉터리는 각 사용자별로 별도의 디렉터리를 갖는 형태이다.



UFD : 자신만의 사용자 파일 디렉터리

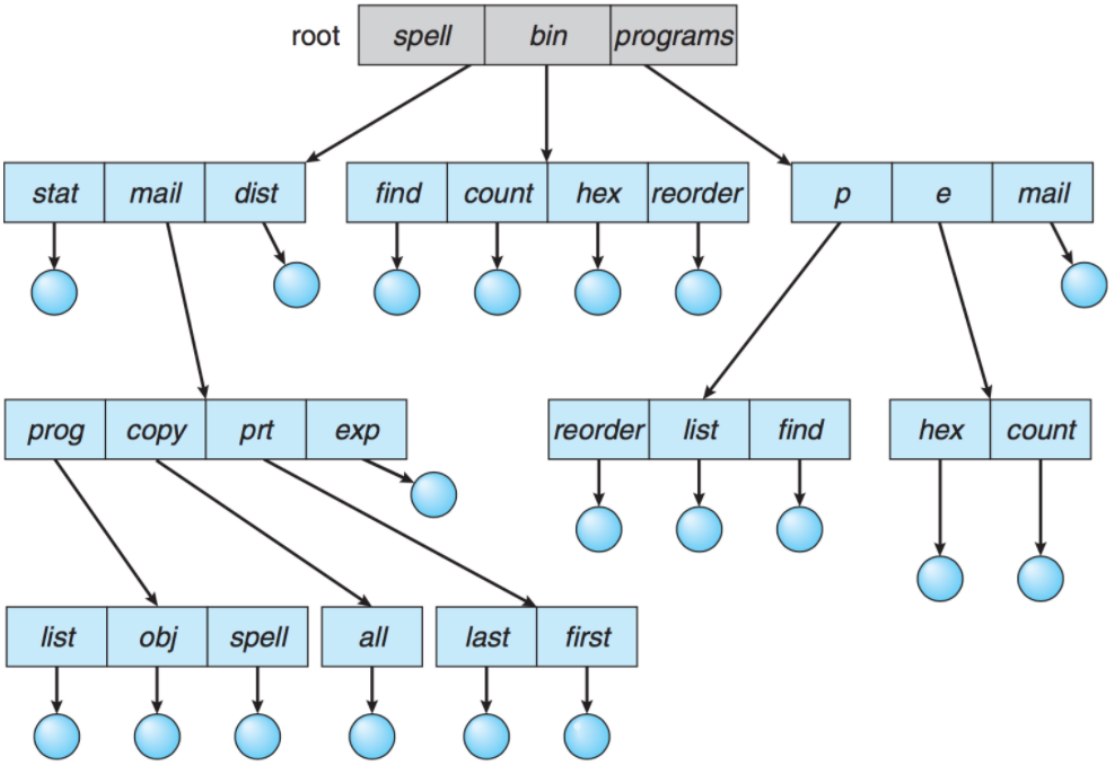
MFD : 사용자의 이름과 계정 번호로 색인되어 있는 디렉터리. 각 엔트리는 사용자의 UFD를 가리킨다.

서로 다른 사용자가 같은 이름의 파일을 가질 수 있고, 효율적인 탐색이 가능하다. 하지만 그룹화가 불가능하고, 다른 사용자의 파일에 접근해야 하는 경우에는 단점이 된다.

3. **트리 구조 디렉터리**(Tree-Structured Directory)

   사용자들이 자신의 서브 디렉터리(Sub-Directory)를 만들어서 파일을 구성할 수 있다. 하나의 루트 디렉터리를 가지며 모든 파일은 고유한 경로(절대 경로/ 상대 경로)를 가진다. 이를 통해 효율적인 탐색이 가능하고, 그룹화가 가능하다.

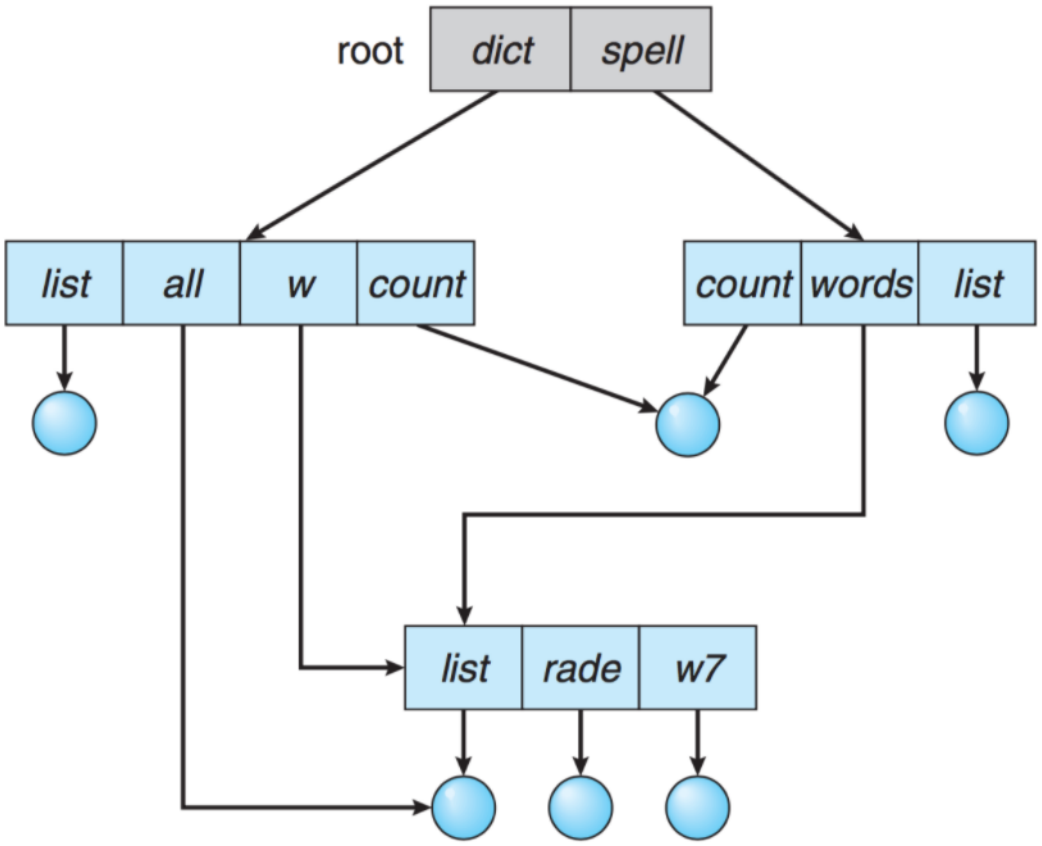
   디렉터리는 일종의 파일이므로 일반 파일인지 디렉터리인지 구분할 필요가 있다. 이를 bit를 사용하여 0이면 일반 파일, 1이면 디렉터리로 구분한다.



4. **비순환 그래프 디렉터리**(Acylic-Graph Directory)

   디렉터리들이 서브 디렉터리들과 파일을 공유할 수 있도록 한다. 트리 구조의 디렉터리를 일반화한 형태이다. 단순한 트리 구조보다는 더 복잡한 구조이기 때문에 몇몇 문제가 발생할 수 있다.

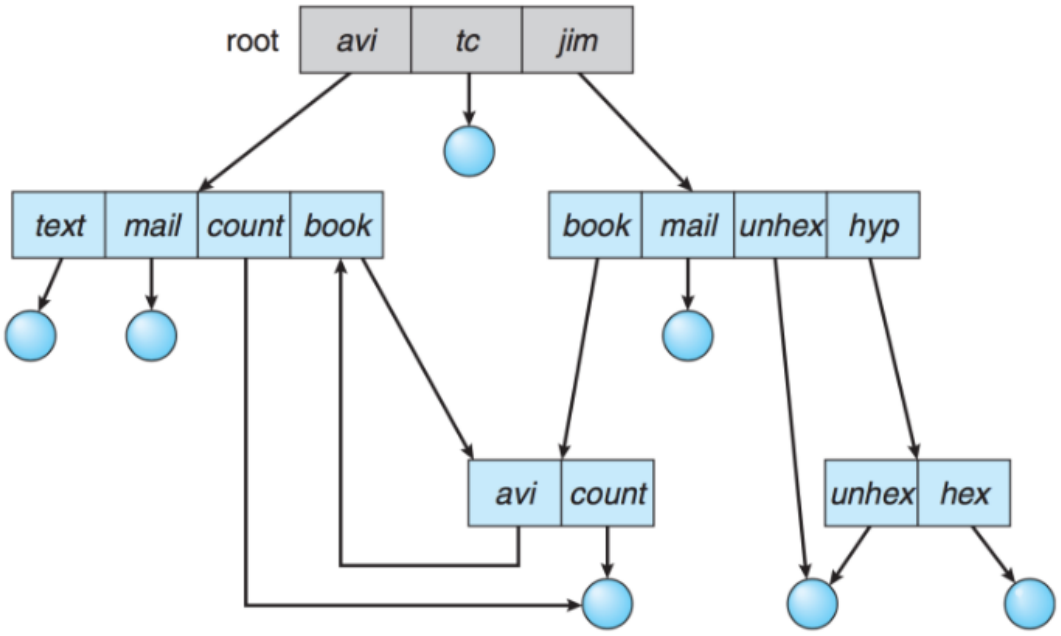
   파일을 무작정 삭제하게 되면 현재 파일을 가리키는 포인터는 대상이 사라지게 된다. 따라서 참조되는 파일에 참조 계수를 두어서, 참조 계수가 0이 되면 파일을 참조하는 링크가 존재하지 않는다는 의미이므로, 그때 파일을 삭제할 수 있도록 한다.



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

   순환을 허용하는 그래프 구조이다. 순환이 허용되면 무한 루프에 빠질 수 있다.

   따라서, 하위 디렉터리가 아닌 파일에 대한 링크만 허용하거나 garbage collection을 통해 전체 파일 시스템을 순회하고, 접근 가능한 모든 것을 표시한다.



Allocation of File Data in Disk

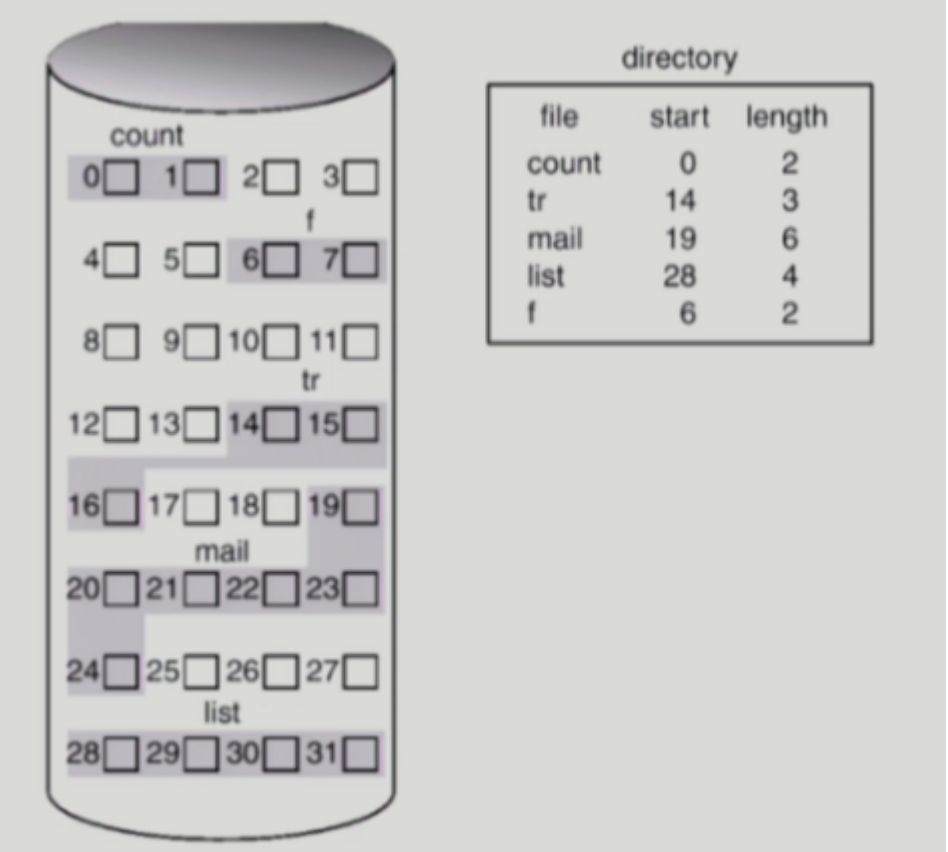
파일 데이터를 디스크에 할당하는 방법으로 다음 세 방식이 존재한다.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

1. **Contiguous Allocation**

   연속 할당은 말 그대로 파일을 디스크에 연속되게 저장하는 방식이다.

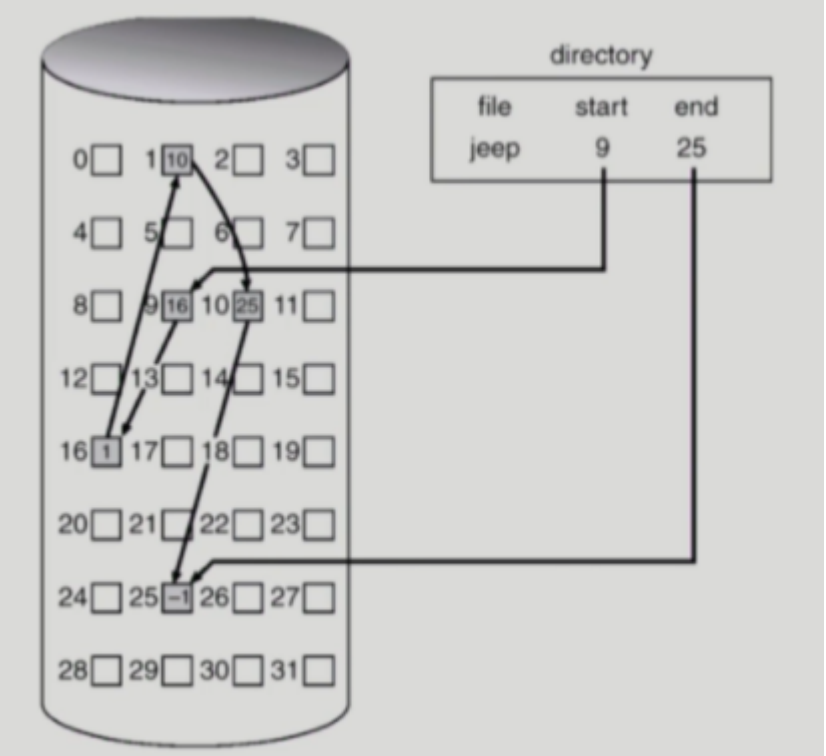
   디렉터리에는 파일이 시작 부분의 위치와 파일의 길이에 대한 정보를 저장하면 전체를 탐색할 수 있다.



이 방식은 연속적으로 저장되어 있으므로 한 번의 탐색으로 많은 양을 전송할 수 있다. 그리고 Random access가 가능하다. 하지만, 외부 단편화가 발생하며, 파일의 크기를 키우기가 어렵다. 파일의 커질 가능성을 고려해서 미리 큰 공간을 할당한다면 내부 단편화가 발생할 수도 있다.

2. **Linked Allocation**

   연결 할당(Linked Allocation)은, 연속적으로 할당하지 않고 빈 위치면 자유롭게 할당될 수 있다. 그리고 다름 읽어야 할 위치를 연결 리스트처럼 포인터로 가리킨다. 따라서 디렉터리에는 파일이 시작하는 위치와 끝나는 위치만 저장된다.



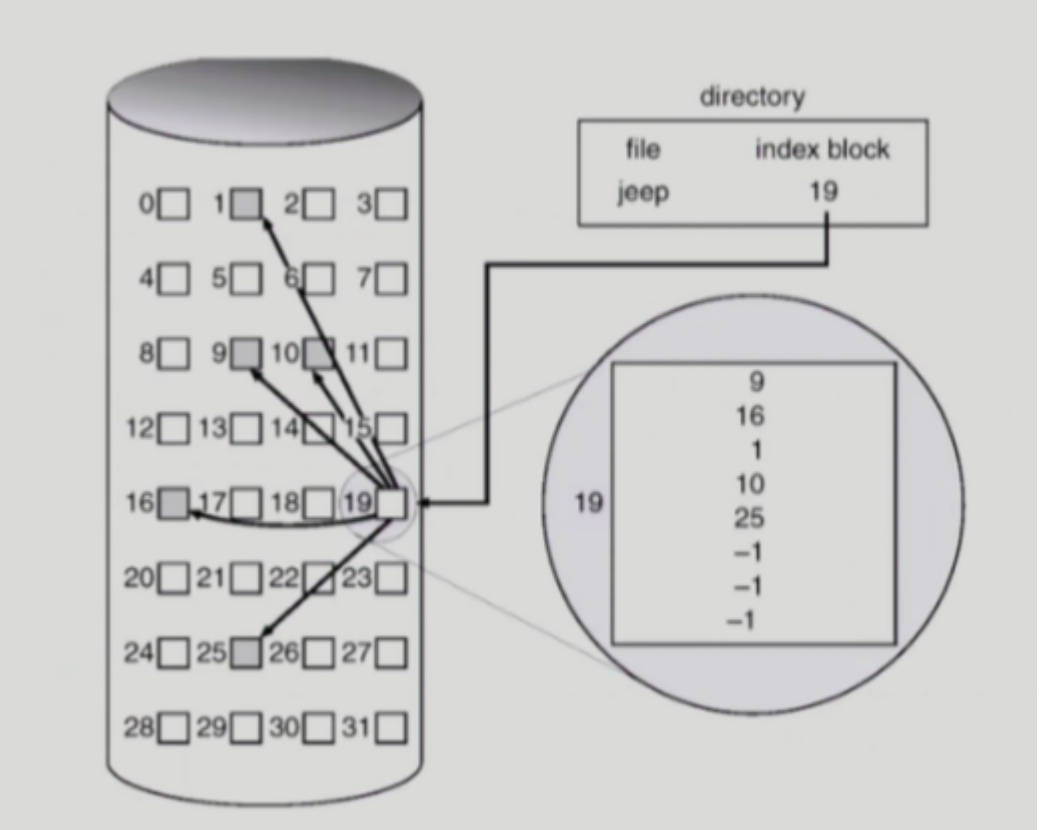
이 방식은 외부 단편화가 발생하지 않는다는 장점이 있다.

하지만 Random access가 불가능하고, 포인터를 위한 공간이 block의 일부가 되어 공간 효율성을 떨어뜨린다. 주로 한 Sector당 512 byte를 배정하는데, 4byte의 포인터르 위한 공간이 사용되면 512의 배수로 할당하기 위헤서는 포인터를 위해 한 Sector만큼 공간이 소모되는 비효율이 발생한다. 또, 만약 한 sector가 고장 나 포인터가 유실되면 많은 부분을 잃게 되는 신뢰성 문제도 있다.

이러한 단점을 보완하기 위해 FAT(File-allocation table)이라는 파일 시스템을 사용한다. 이는 포인터를 별도의 위치에 보관하여 신뢰성 문제와 공간 효율성 문제를 해결한다.

3. **Indexed Allocation**

색인 할당(Indexed Allocation)은 한 블록에 하나의 파일에 대한 데이터의 index들을 모두 저장하는 방식이다. 따라서 디렉터리에는 해당 블록의 위치만 남게 된다.



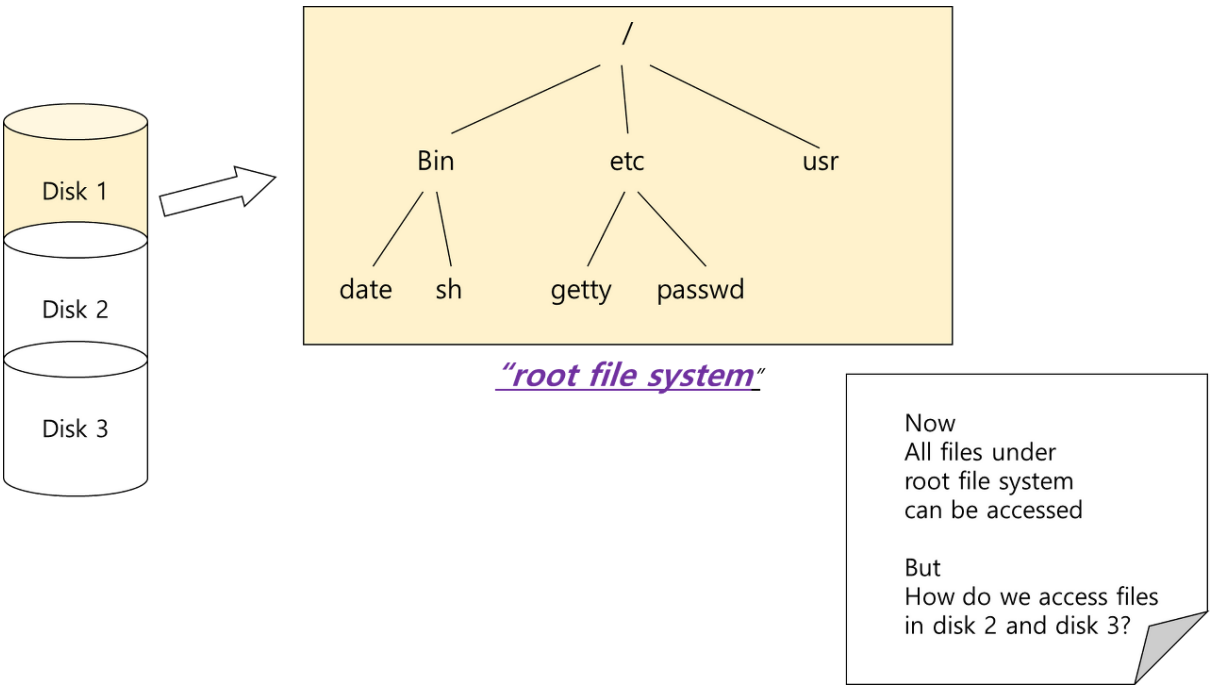
이 방식은 외부 단편화가 발생하지 않고 Random Access가 가능하다.

다만, 작은 파일인 경우 위치를 저장하는 블록의 공간 낭비가 생기고, 너무 큰 파일인 경우에는 하나의 블록으로 파일의 index들을 모두 저장하기에 부족하게 되는 단점이 있다.

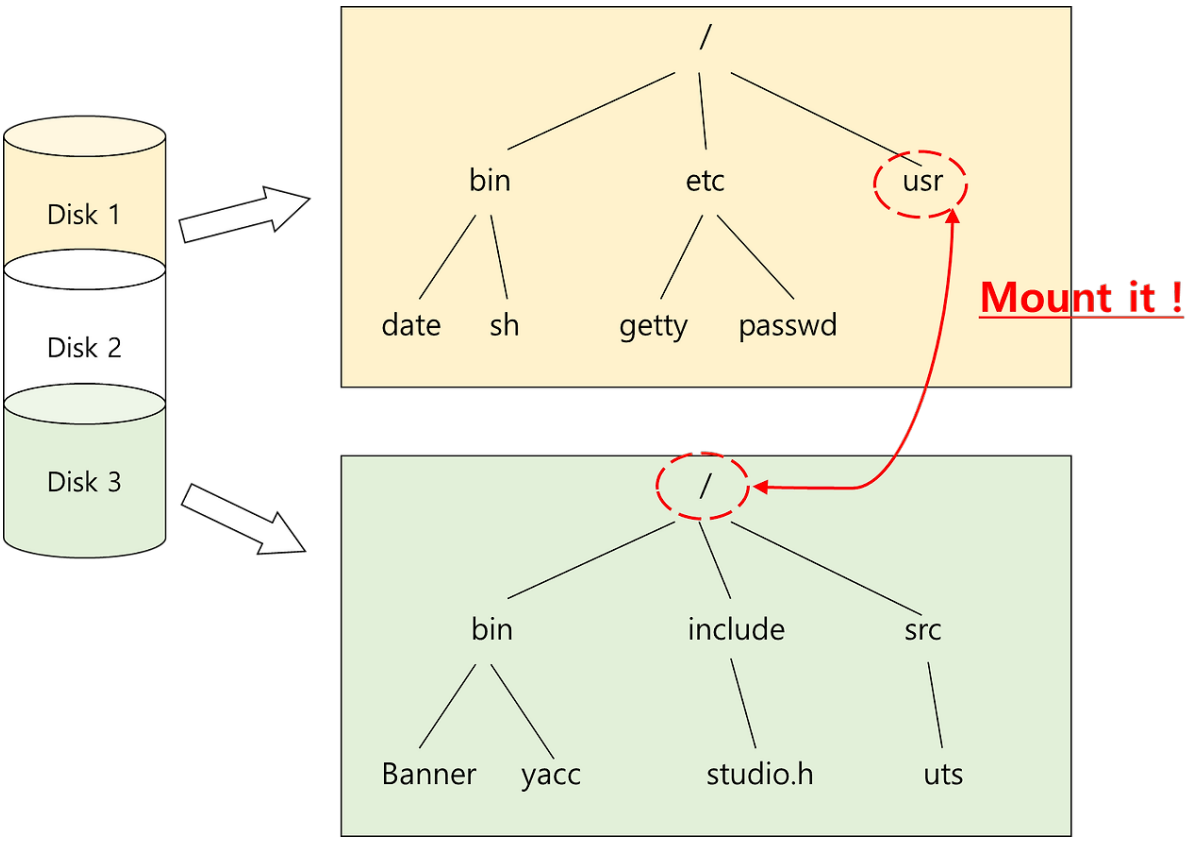
Mounting

하나의 물리적인 디스크를 여러 개의 논리적인 디스크로 나눌 수 있으며, 각각의 논리적인 디스크에서 파일 시스템을 설치해 사용할 수 있다.

어떤 특정 운영체제에 대해서 파일 시스템 하나가 접근 가능한데, 만약 다른 파티션에 설치되어 있는 파일 시스템에 접근하려고 하면 **Mounting**을 해야 한다.



Root File System의 특정 Directory에 또 다른 파티션에 있는 파일 시스템을 Mount해준다. 그러면 그 Mount 된 Directory에 접근하는 것은 또 다른 파일 시스템의 Root Directory에 접근하는 것이 된다.



시스템이 제공하는 파일 정보의 접근 방식

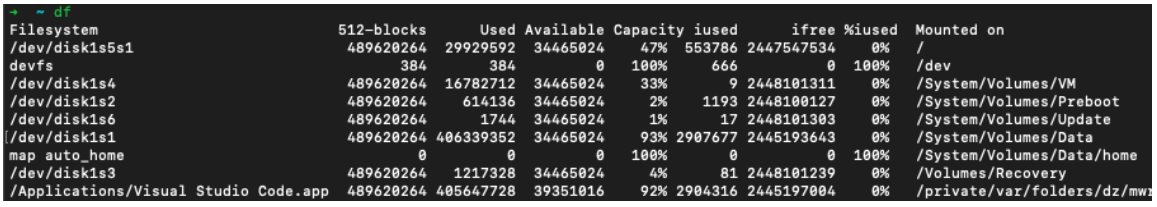
- **순차 접근**(Sequential Access) : 카세트테이프를 사용하는 방식처럼 접근, 읽거나 쓰면 offset은 자동으로 증가
- **직접 접근**(Direct Access, Random Access) : 레코드판과 같이 접근하도록 함. 파일을 구성하는 레코드를 임의의 순서로 접근할 수 있음

Inode

inode란 Linux/Unix의 파일 시스템에서 사용되는 자료구조를 말한다. inode 방식에는 크게 3가지 블록 타입이 있는데, super block, inode block, data block으로 나뉜다.

1. 슈퍼 블록

   슈퍼 블록은 파일 시스템 전체를 대표하는 블록으로, 파일 시스템 정보 및 파티션 정보 등을 포함하고 있다. 리눅스 터미널에서 df 명령어를 치면 슈퍼블록 정보를 볼 수 있다.



```
➜  ~ df
Filesystem                      512-blocks      Used Available Capacity iused      ifree %iused  Mounted on
/dev/disk1s5s1                   489620264  29929592  34465024    47%  553786 2447547534    0%  /
devfs                                  384       384         0   100%     666          0  100%  /dev
/dev/disk1s4                     489620264  16782712  34465024    33%       9 2448101311    0%  /System/Volumes/VM
/dev/disk1s2                     489620264    614136  34465024     2%    1193 2448100127    0%  /System/Volumes/Preboot
/dev/disk1s6                     489620264      1744  34465024     1%      17 2448101303    0%  /System/Volumes/Update
/dev/disk1s1                     489620264 406339352  34465024    93% 2907677 2445193643    0%  /System/Volumes/Data
map auto_home                            0         0         0   100%       0          0  100%  /System/Volumes/Data/home
/dev/disk1s3                     489620264   1217328  34465024     4%      81 2448101239    0%  /Volumes/Recovery
/Applications/Visual Studio Code.app 489620264 405647728  39351016    92% 2904316 2445197004    0%  /private/var/folders/dz/mw
```

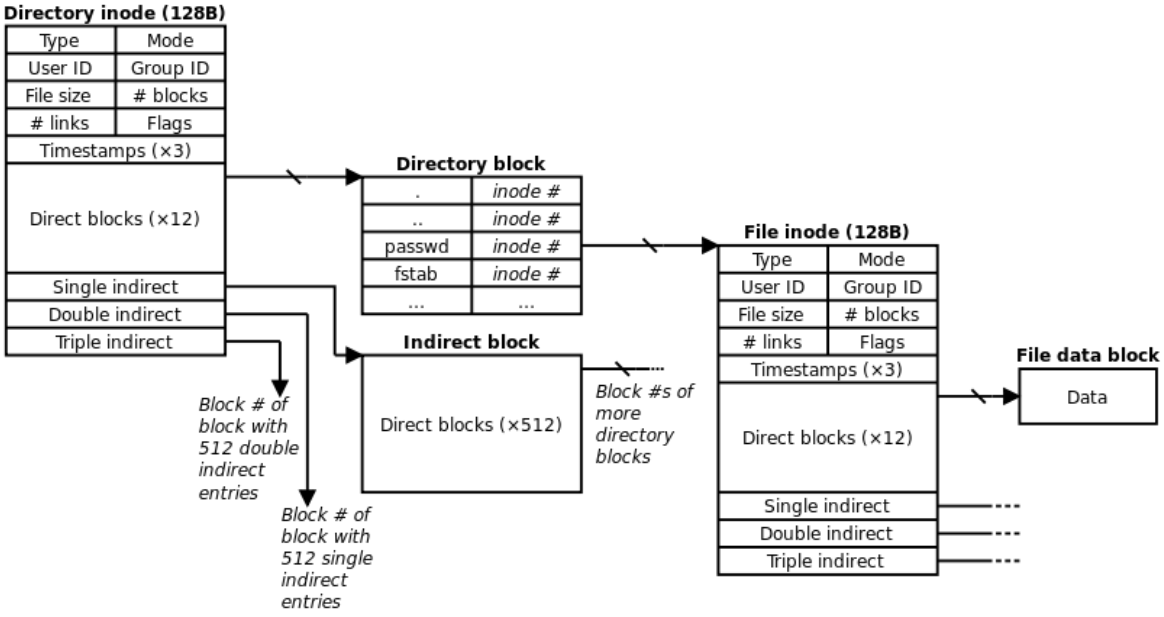
   파일 시스템이 뭐가 있는지, 512Byte짜리 블록이 몇 개인지, 사용되고 있는 용량과 남아있는 용량은 얼마인지, 각각의 파일 시스템은 어디에 마운티드 되어 있는지 등의 정보를 보여준다.

2. inode 블록

   inode 블록은 inode가 모여 있는 공간이다. 모든 파일이나 디렉토리는 하나씩 inode를 가지고 있고, 각 파일마다 inode 번호가 할당되어 있다. 한 개의 inode는 64byte로 이루어져 있으며, 해당 파일의 소유권 허가원, 파일 종류, 실제 데이터의 주소 등을 가지고 있다. 이것이 메타 데이터이다.

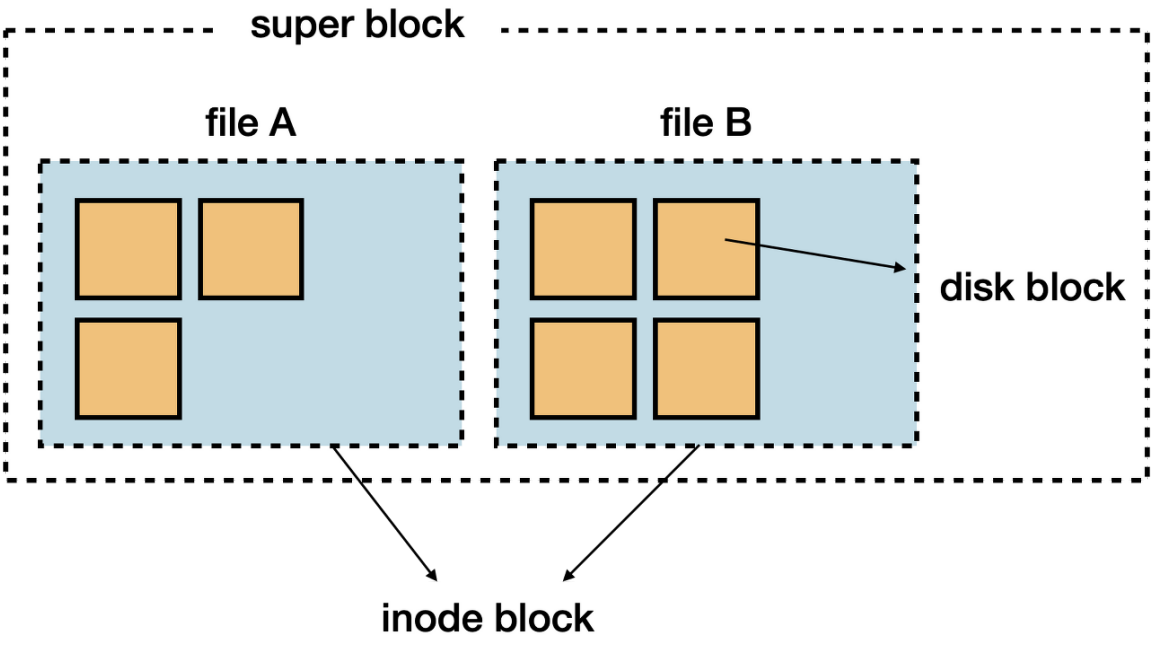
   우리는 외부적으로 파일 이름만 아는 상태로 파일을 읽고 쓰지만, 내부적으로는 inode 정보를 참조해서 처리된다. 마치 프로세스와 프로세스 id가 매칭된 것과 같이 모든 파일에는 inode 번호가 매칭되어 있다. 프로세스 처리를 위해 PCB에 있는 정보를 기반으로

처리하듯이, 응용 프로그램이 특정 파일에 엑세스하기 위해서는 해당 파일의 정보가 담긴 inode를 참조한다.



3. 데이터 블록

   실제 파일 내용이 저장되는 디스크 블록의 집합을 의미한다.



References :

https://rebro.kr/181

https://eunjinii.tistory.com/147