

# 10. File and File System

## • File

- "A named collection of related information"
- 일반적으로 비휘발성의 보조 기억 장치에 저장
- 운영체제는 다양한 저장 장치를 file 이라는 동일한 논리적 단위로 볼 수 있게 해줌
- Operation
  - Create, read, write, reposition(seek), delete, open, close 등

파일을 접근하는 위치를  
수정해 줄 때  
파일의 meta 정보를  
매번 바꿔 쓰는 일

## • File attribute (혹은 파일의 meta data)

- 파일 자체의 내용이 아니라 파일을 관리하기 위한 각종 정보들
- 파일 이름, 유형, 저장된 위치, 파일 사이즈
- 접근 순서(읽기, 쓰기, 실행), 시간(생성, 변경, 사용), 소유자 등

## • File System

Software

- 운영체제에서 파일을 관리하는 부분
- 파일 및 파일의 메타데이터, 디렉토리 정보 등을 관리
- 파일의 저장방법 결정
- 파일 보호 등

# Directory and Logical Disk

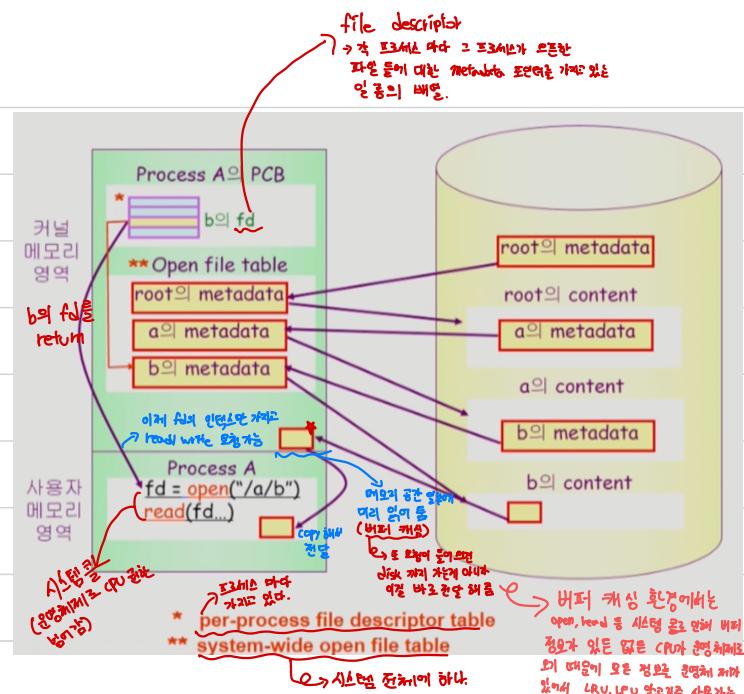
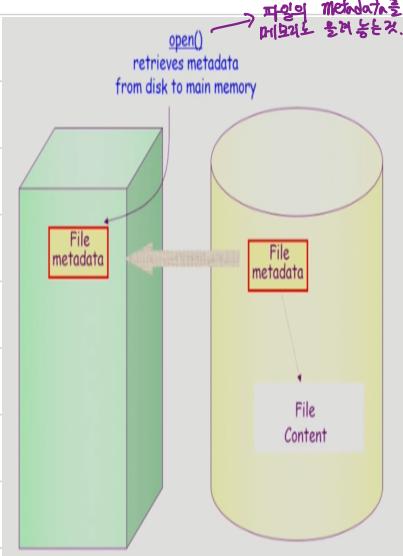
## • Directory

- 파일의 메타데이터 중 일부를 포함하고 있는 일종의 특별한 파일
- 그 디렉토리에 속한 파일 이름 및 파일 attribute 들
- Operation
  - Search for a file, create a file, delete a file
  - list a directory, rename a file, traverse the file system  
전체 탐색

## • Partition (= Logical Disk)

- 하나의 (물리적) 디스크 안에 여러 파티션을 두는게 일반적
- 여러개의 물리적인 디스크를 하나의 파티션으로 구성하기도 함
- (물리적) 디스크를 파티션으로 구성한 뒤 각각의 파티션에  
file system을 할거나 swapping 등 다른 용도로 사용할 수 있음

# open( )



## • open("a/b/c")

v 디스크로부터 파일 C의 메타데이터를 메모리로 가지고 옴

v 이를 위하여 directory path를 search

- 루트 디렉토리 "/"를 open하고 그 안에서 파일 "a"의 위치 획득
- 파일 "a"를 open한 후 read 하여 그 안에서 파일 "b"의 위치 획득
- 파일 "b"를 open하는 동안 read 하여 그 안에서 파일 "c"의 위치 획득
- 파일 "c"를 open 한다.

v Directory path의 search가 너무 많은 시간 소요

- open을 read/write와 별도로 두는 이유임.
- 한번 open한 파일은 read/write 시 directory search 불필요

## v Open file table

- 현재 open된 파일들의 메타데이터 보관소 (in memory)

- 디스크의 메타 데이터 보다 빠르게 정보 추가

- open 한 프로세스의 수

- file offset: 파일 어느 위치 접근 풍선지 표시 (별도 테이블 필요)

## v File descriptor (file handle, file control block)

- Open file table에 대한 위치 정보 (프로세스 별)

# File Protection

- 각 파일에 대해 누구에게 어떤 유형의 접근(read/write/execution)을 허락할 것인가?

## Access control 방법

### Access control Matrix

	file 1	file 2	file 3
User 1	rw	rw	r
User 2	rw	r	r
User 3	r		

ACL (Access Control List)  
linked list

Capability  
linked list

행렬 관 관리 → linked list로 관리

- Access control list: 파일 별로 누군에게 어떤 접근 권한이 있는지 표시

- Capability: 사용자 별로 자신이 접근 권한을 가진 파일 및 해당 권한 표시  
→ 어떤 방법보다  
하나만 사용하면 됨

but, 이렇게 해도 overhead 큼.  
그래서 일반 운영 체제에서 Grouping 사용

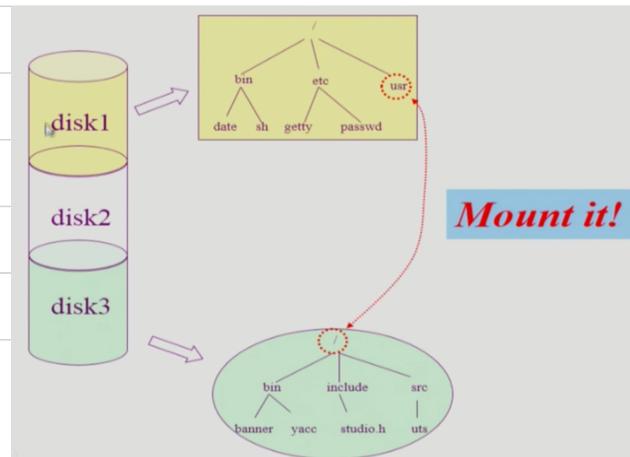
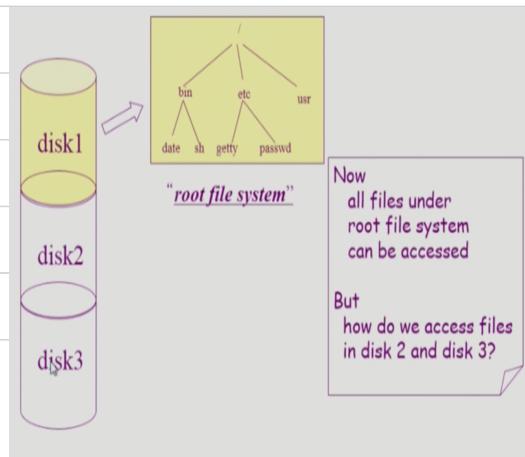
### Grouping

- 전체 USER를 owner, group, public의 세 그룹으로 구분
- 각 파일에 대해서 세 그룹의 접근 권한 (rwx)을 비트셋으로 표시
- (예) UNIX 

### Password

- 각 파일마다 password를 두는 방법 (디렉토리 파일에 두는 방법도 가능)
- 모든 접근 권한이 대해 하나의 password: all-or-nothing
- 접근 권한 별 password: 암기 문제, 관리 문제

# File System의 Mounting



# Access Methods

- 시스템이 제공하는 파일 정보의 접근 방식

- ▼ 순차 접근 (sequential access)

- 커서트 레코드를 사용하는 방식처럼 접근
    - 일제히 쓰면 애드레스 자동적으로 증가

- ▼ 직접 접근 (direct access, random access)

- LP 레코드 번호 같이 접근 하도록 함
    - 파일을 구성하는 레코드를 일의의 순서로 접근 할 수 있음.

# Allocation of File Data in Disk

- Contiguous Allocation

연속  
연속

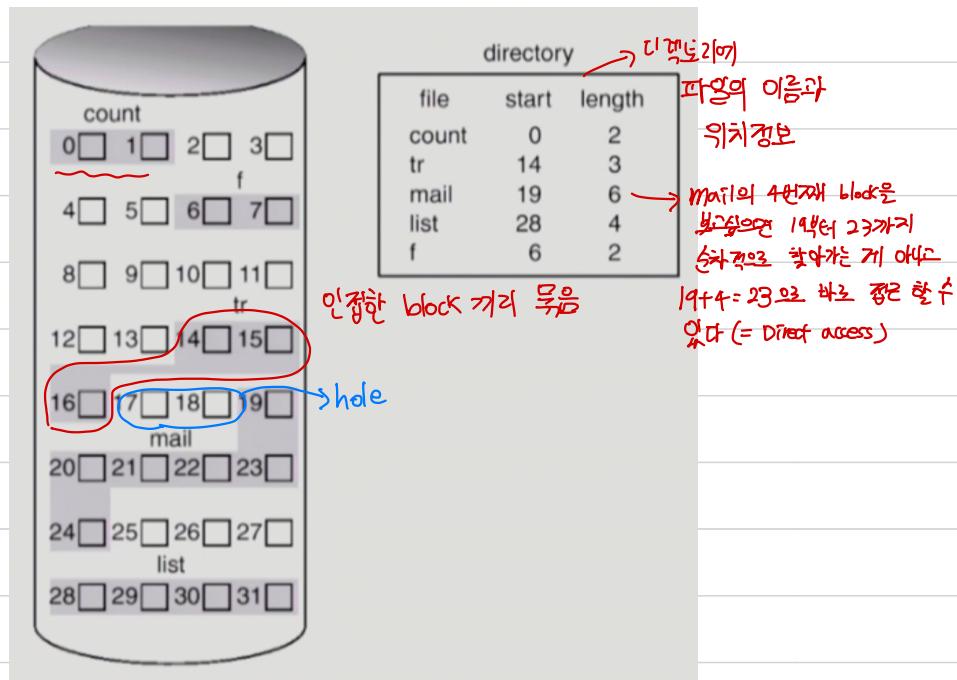
- Linked Allocation

링크  
링크

- Indexed Allocation

인덱스  
인덱스

# Contiguous Allocation



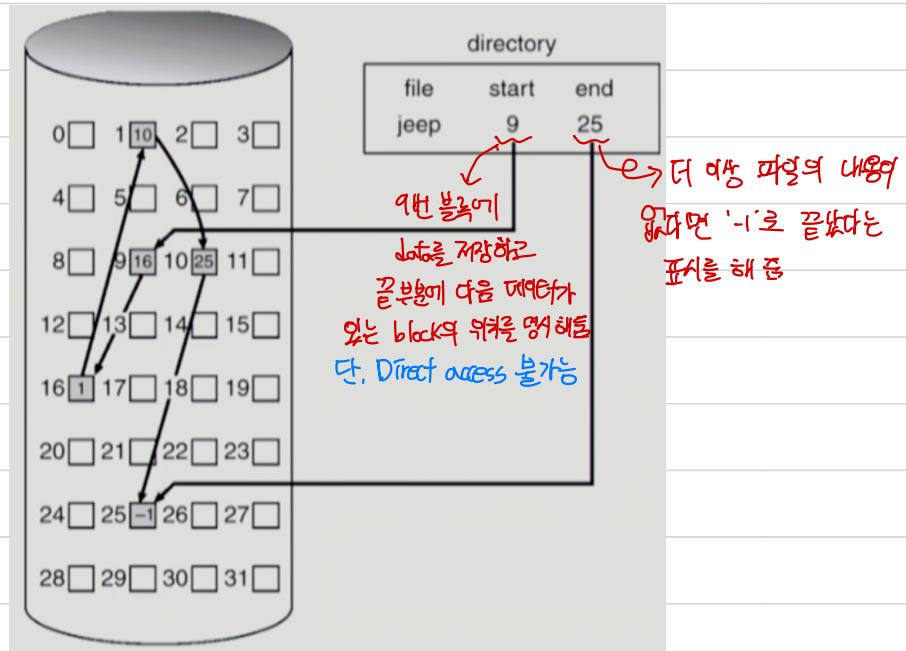
## • 단점

- ✓ external fragmentation
- ✓ File growth 어려움
  - file 생성시 빈거나 큰 hole을 배당할 것인가?
  - grow 가능 VS 낭비 (internal fragmentation)

## • 장점

- ✓ Fast I/O
  - 한 번의 Seek/rotation으로 많은 바이트 transfer
  - Real time file 용으로, 또는 여러 run 풋이던 Process Swap 용  
→ 공간 효율성 보다는 시간 효율성이 좋아야 하는 데이터 ⇒ 빠른 I/O
- ✓ Direct access (= random access) 가능

# Linked Allocation



## • 장점

- External fragmentation 발생 안 함

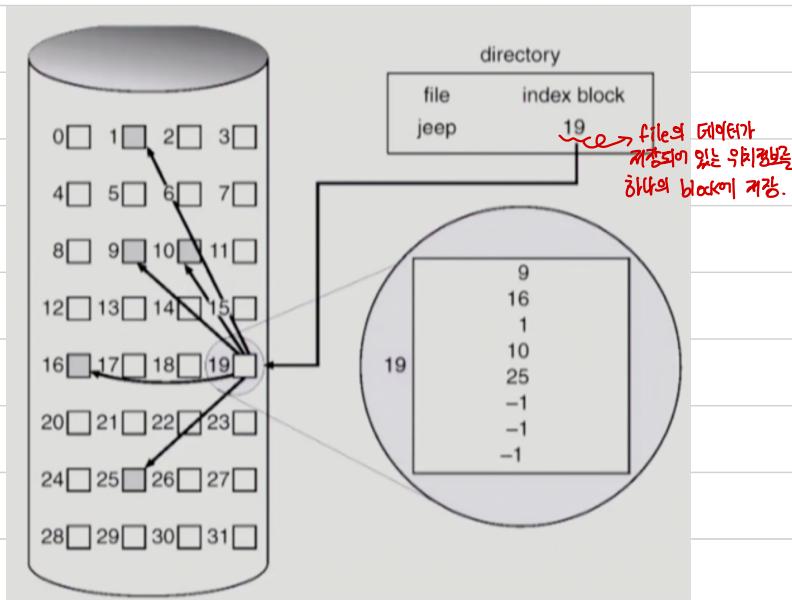
## • 단점

- No random access
- Reliability 문제
  - 한 Sector가 고장나 pointer가 유효 되면 많은 부분을 잃음
  - Pointer를 위한 공간이 block의 일부가 되어 공간 효율성을 떨어뜨림
    - 설계로는 512bytes/Sector, 4 bytes/pointer
    - 512-4

## • 변형

- File-allocation table (FAT) 파일 시스템
  - 포인터를 별도의 위치에 보관하여 reliability와 공간 효율성 문제 해결

# Indexed Allocation



## • 장점

- ✓ External fragmentation이 발생하지 않음
- ✓ Direct access 가능

## • 단점

- ✓ Small file의 경우 공간 낭비 (실제로 많은 파일들이 small)
- ✓ Too Large file의 경우 하나의 block으로 index를 저장하기에 부족
  - 해결 방안
    - 1. linked scheme → index block의 마지막에 또 다른 index block의 위치 삽입.
    - 2. multi-level index

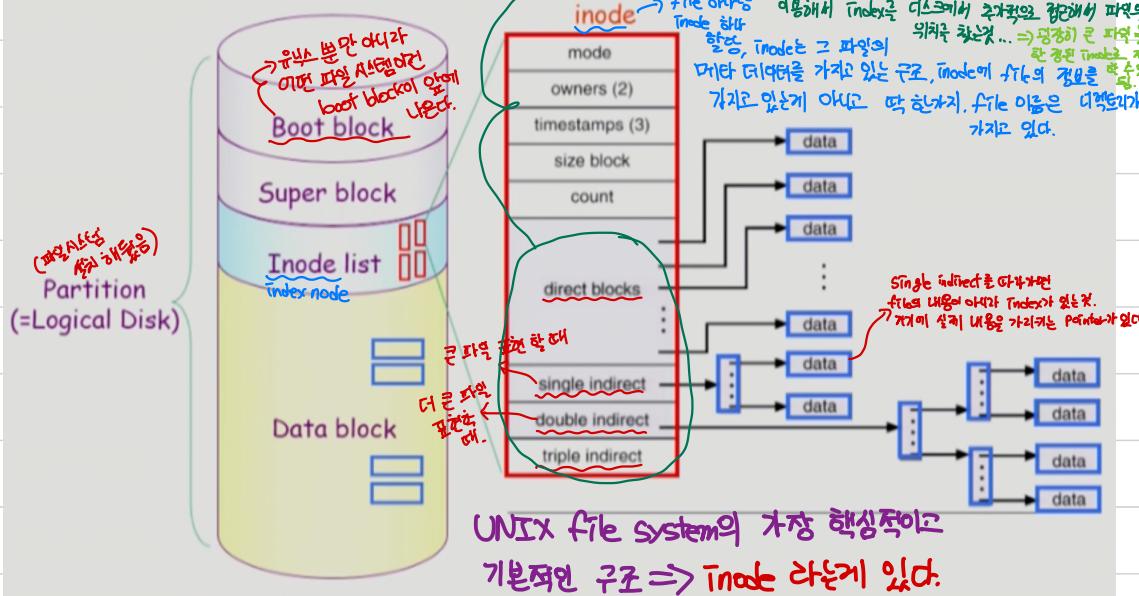
# UNIX 파일 시스템의 구조

이게 왜 효율적인가?

대부분의 file은 크기 아주 작다.

작은 file들은 한 번의 pointer 접근으로, inode만  
메모리에 둘러 놓으면 파일의 위치를 바로 알 수 있고

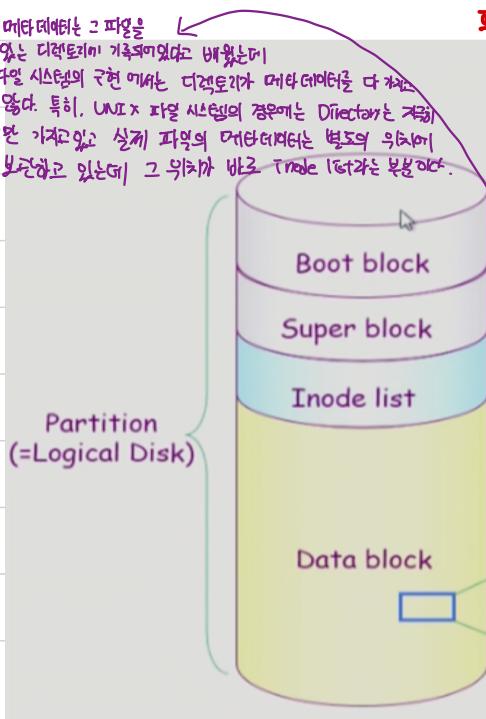
가끔 발생하지만 큰 파일이면 indirect block들을  
file 헤더를 이용해서 index를 디스커버 추가로 접근해서 파일의  
크기를 찾는다. => 굉장히 큰 파일을  
파악하는 데에는 그 파일의  
메타 데이터를 가지고 있는 구조, inode에 file의 정보를  
기하고 있는게 아니고 딱 한 가지, file 이름은 디렉토리가  
기하고 있다.



파일의 위치정보는 indexed allocation 방식으로 사용

◆ 유닉스 파일 시스템의 중요 개념

- ✓ Boot block => boot block
  - 부팅에 필요한 정보 (bootstrap loader)
- ✓ Superblock
  - 파일 시스템에 관한 종합적인 정보를 담고 있다.
- ✓ Inode
  - 파일 이름을 제외한 파일의 모든 메타 데이터를 저장
- ✓ Data block
  - 파일의 실제 내용을 보관



directory file

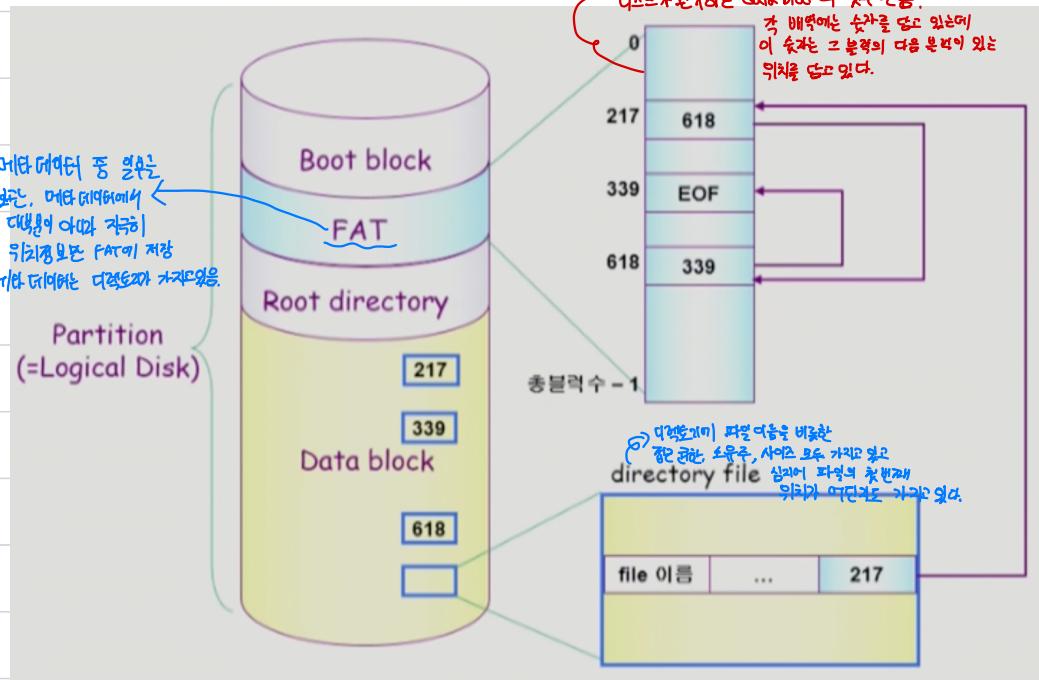
file 이름 | inode 번호

# FAT File System

Microsoft가  
처음으로 MS-DOS를  
만들었을 때 만든 시스템.

파일의 메타데이터 중 일부는  
FAT에 보존, 메타데이터에서  
얻을 수 있는 대체로 이미 가능하  
제한적인 유동성을 FAT에 저장  
내마지막 예상 데이터에는 디렉토리가 가지지 않음.

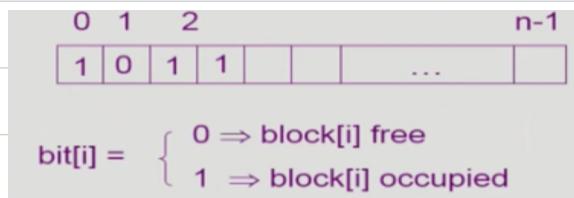
Partition  
(=Logical Disk)



FAT File System의 또 다른 장점 ⇒ 직접 접근이 가능하다. linked allocation에선 중간 위치의 데이터를 보려면 처음 부분부터 따라가야 해서 직접 접근이 안 되는데, FAT file system에서 FAT은 이미 메모리에 올라가 있는 작은 데이터들이기 때문에 찾고자 하는 데이터가 있는 위치를 FAT 테이블에서 찾고 바로 경로를 찾기 때문에 가능하다.  
 ⇒ Linked allocation의 단점 모두 극복!  
 ↗ data block과 FAT는 같은 불리티 있다.  
 reliability 문제 → 포인터 하나가 dead sector가 나타나도 FAT에 내용이 있고 FAT은 매우 중요하게  
 때문에 보통 2개 이상을 copy하여 저장하고 있다.

# Free - Space Management

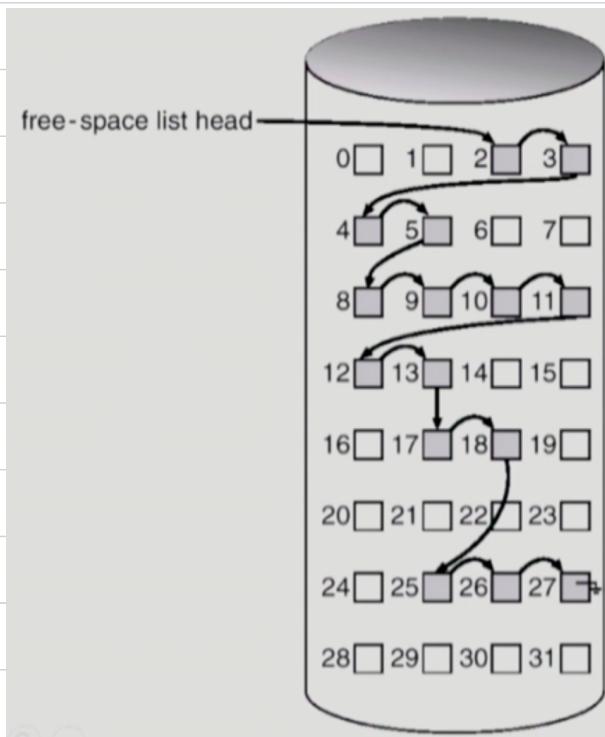
- Bit map or bit vector



- v Bit map은 부가적인 공간을 필요로 함. ↗ 단점이지만 단점, 하지만 그렇게 많은 공간을 차지하지 않는다.
- v 연속적인 가격의 free block을 찾는데 효과적

## • Linked list

- v 모든 free block들을 링크로 연결 (free list)
- v 연속적인 사용 공간을 찾는 것은 쉽지 않다. ↗ 이론적으로 이런 방법이 있다.  
그러나 쓰인 습지 않을 것이다.
- v 공간의 낭비가 없다.



비어있는 block을 찾기че linked list 보다 효과적이지만

연속적으로 비어있는 block을 찾기че는 쪽 효과적이지 못하다.

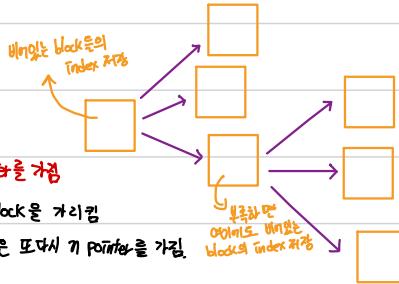
## • Grouping

✓ linked list 방법의 단점

✓ 첫 번째 free block이 까지의 pointer를 가짐

• 1-1 pointer는 free data block을 가리킴

• 마지막 pointer가 가리키는 block은 또다시 기 pointer를 가짐.



## • Counting

✓ 프로그램들이 종종 여러 개의 연속적인 block을 할당하고

반납한번은 성질에 확인

✓ (first free block, # of contiguous free blocks)을 구지

빈블록의 첫 위치) 몇개가 연속적인지)

# Directory Implementation

## • Linear list

✓ <file name, file의 metadata>의 list

✓ 구현이 간단

✓ 디렉토리 내에 파일이 있는지 찾기 위해서는 linear search 필요 (time-consuming)

## • Hash Table

✓ linear list + hashing

✓ Hash table은 file name을 이 파일의 linear list의 위치로 빼주어 줌

✓ Search time은 없음.

✓ Collision 발생 가능

directory file의 구조 (linear list)	
file name	file의 metadata
aaa	aaa의 metadata
cccc	cccc의 metadata
bbba	bbba의 metadata
abc	abc의 metadata
:	:

Hash 함수  
충돌 발생하는 경우

directory file의 구조 (hash table)	
F (file name)	file의 metadata
1	
2	
3	
:	:
n	

파일 이름 hash 함수 적용  
→ 해시槽의 metadata에  
입수해 줌

## • File의 metadata의 보관 위치

✓ 디렉토리 내에 직접 보관

✓ 디렉토리에는 포인터를 두고 다른 곳에 보관

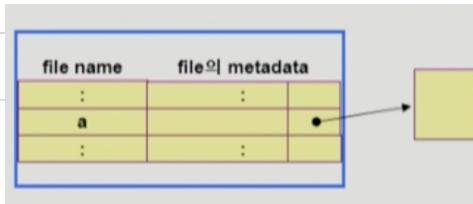
• inode, FAT 등

## • Long file name의 지원

✓ <file name, file의 metadata>의 list에서 각 entry는 일반적으로 고정 크기

✓ file name이 고정크기의 entry 길이보다 길어지는 경우 마지막 부분에  
이름의 뒷 부분이 위치한 곳의 포인터를 두는 방법

✓ 이름의 나머지 부분은 동일한 directory file의 일후에 존재



실제 이름	file name	file의 metadata
aaabb	a a a	pointer
bb	b b	10
ccaa	c c a	pointer
	:	:
	10 a 10 b b	

# VFS and NFS

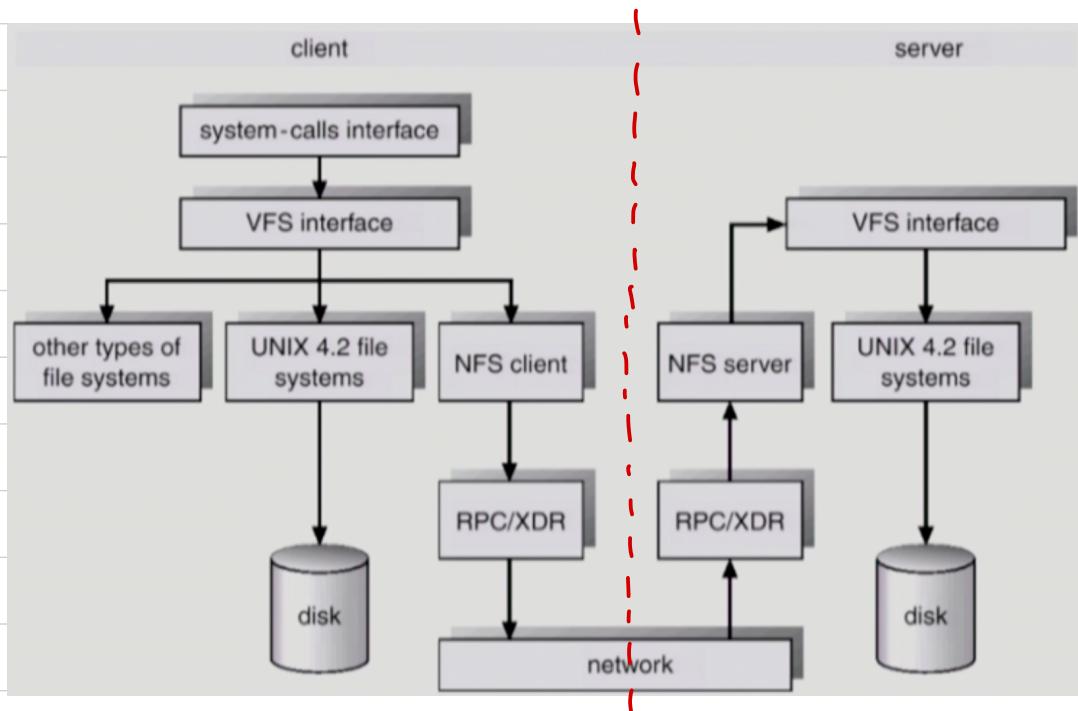
## • Virtual File System (VFS)

✓ 서로 다른 다양한 file system이 대해 통일한 시스템을 인터페이스(API)를 통해 접근할 수 있게 해주는 OS의 layer

## • Network File System (NFS)

✓ 분산 시스템에서는 네트워크를 통해 파일이 공유될 수 있음

✓ NFS는 분산환경에서의 대표적인 파일공유 방법임.



# Page Cache and Buffer Cache

## • Page Cache

- Virtual memory의 paging system에서 사용하는 page frame을 caching의 관점에서 설명하는 용어
- Memory-Mapped I/O를 쓰는 경우 file의 I/O에서도 Page cache 사용

## • Memory-Mapped I/O (File)

- file의 일부를 virtual memory에 mapping 시킴
- 매핑시엔 영역에 대한 메모리 접근 연산은 파일의 입출력을 수행하게 함.

## • Buffer Cache

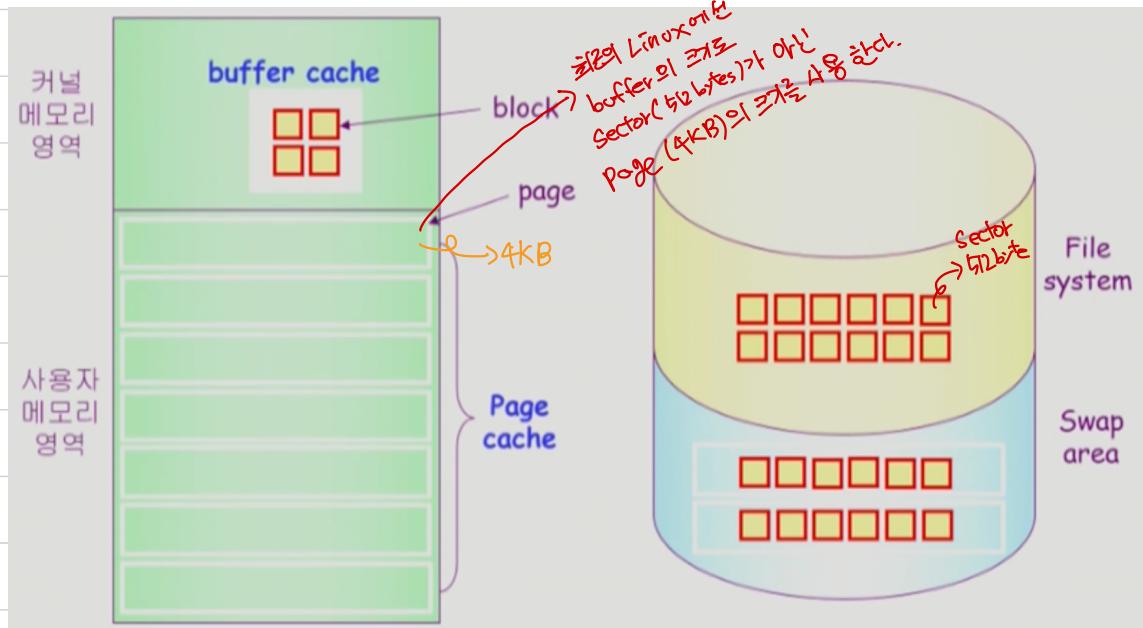
- 파일 시스템을 통한 I/O 연산은 메모리의 특정 영역인 buffer cache 사용
- file 사용의 locality 활용
  - 한번 읽어온 block에 대한 후속 요청시 buffer cache에 즉시 전달
- 모든 프로세스가 공유으로 사용
- Replacement algorithm 필요(LRU, LFU 등)

## • Unified Buffer Cache

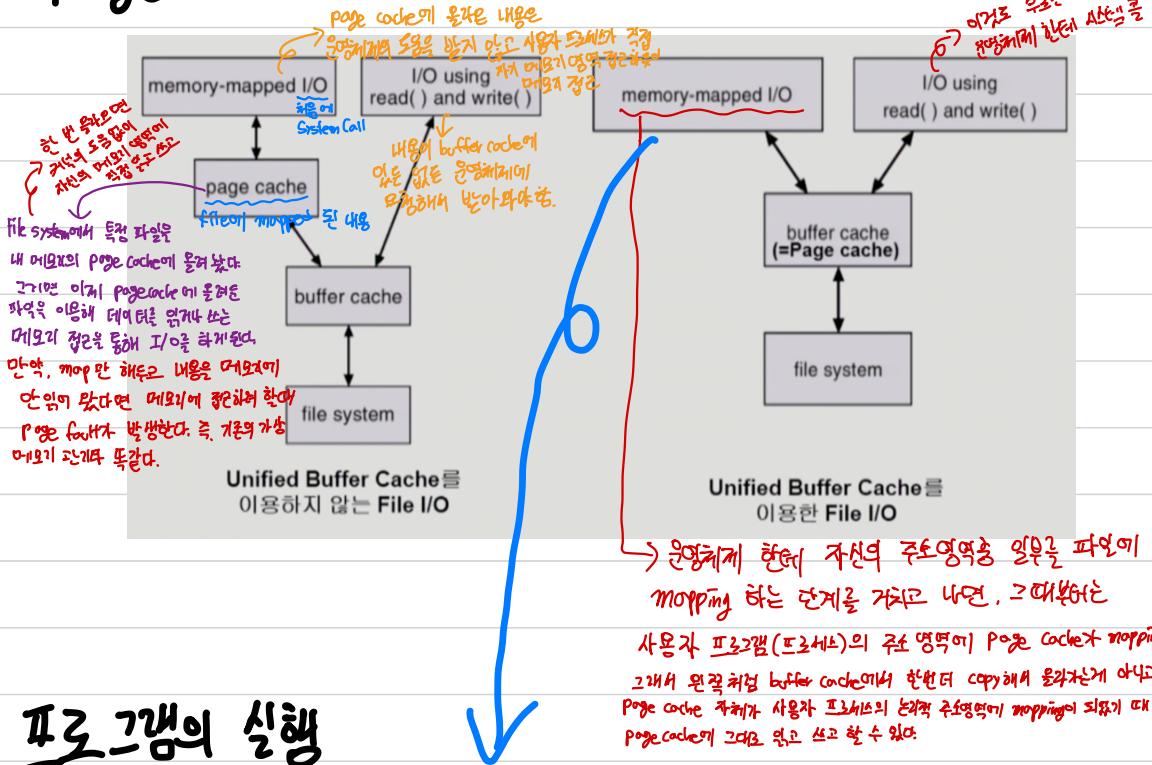
- 최근의 OS에서는 기존의 buffer cache와 page cache가 통합됨

Buffer cache에 올라와 있으나  
File system에 있으나도 차이 나님.

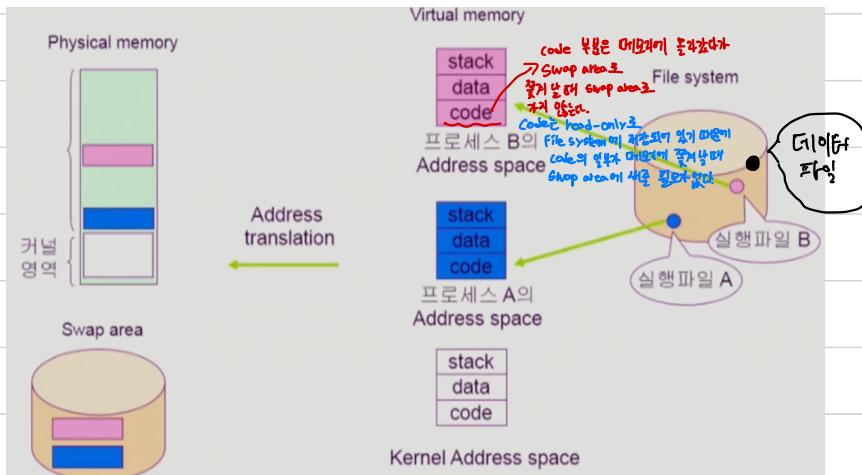
별도로 구분을 하지 않고, 특별히 Page 단위로 구분을  
하면서 파일 내용을 충분히 꾸로하면 Page cache를 활용  
해서 buffer cache 용도로 쓰고, process의 주소공간을  
위한 page 필요하면 page를 주소공간으로 할당해서  
사용. 즉, 별도로 구분해 두지 않고 풀로 쓰다가 필요할 때  
할당을 해서 쓴다.

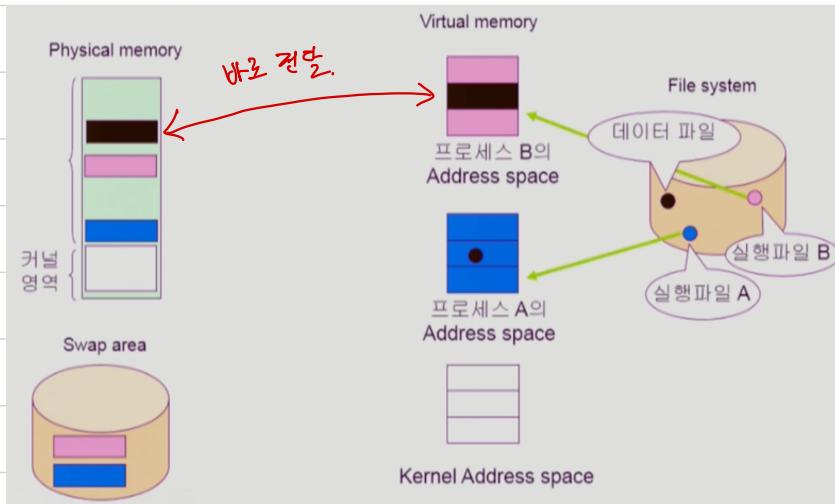


# Page Cache and Buffer Cache



## 프로그램의 실행





단점: 데이터는 메모리 공간에 mapping 해서 쓰다 보니 다른 프로세스와 함께 쓰다가 일관성 문제가 생길 수 있다.