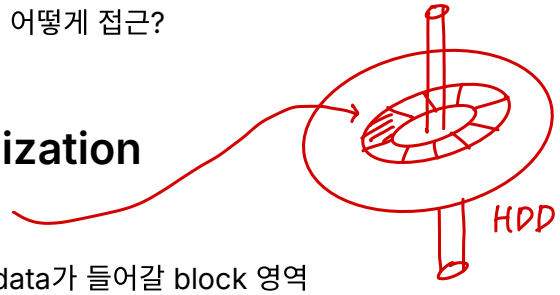
# 23. File System Implementation

실제 내부 구조와 어떻게 연결되는지 알아보자!

## ▼ How to implement a simple file system

- file system is pure software
  - cpu와 memory virtualization과 다르게 file system이 더 일을 잘하도록 h/w적인 기능을 추가하지 않음.
  - data structure → 어떻게 저장?
  - access methods → 어떻게 접근?

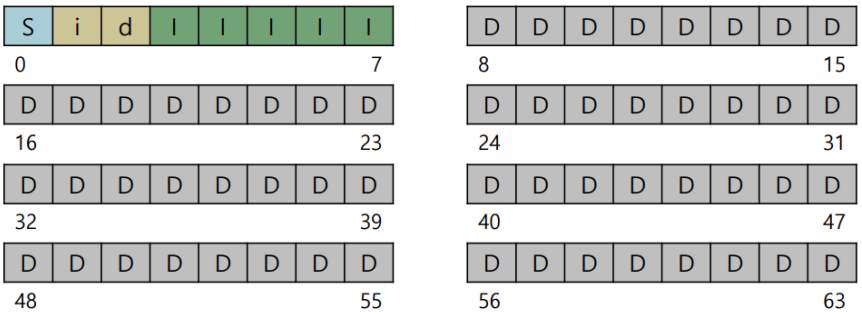



## ▼ file system organization

- **blocks** : 하나의 sector
- **data region** : block 중 data가 들어갈 block 영역
- **metadata** ⇒ inode region
  - 각 file에 대한 track information
    - file size, owner와 접근 권한, 접근 및 수정 시간 등
  - inode(index node)
    - metadata를 저장하고 있는 영역 → disk에서 영역 할당함
- **allocation structures**
  - inode와 data의 어떤 block이 할당되고 free인지 표기하는 부분 필요

  ⇒ free list 혹은 bitmap으로 표시함
- **superblock**
  - 전체 file system에 대한 information 포함
  - 얼마나 많은 inode, data block이 file system에 있는지/어디부터 inode table 시작하는지를 표기함
  - mounting a file system일 때 → 어떤 volume을 mount하려할 때
    - OS : superblock을 읽은 뒤 mount함 ⇒ OS가 이해할 수 있는 형태여야 함

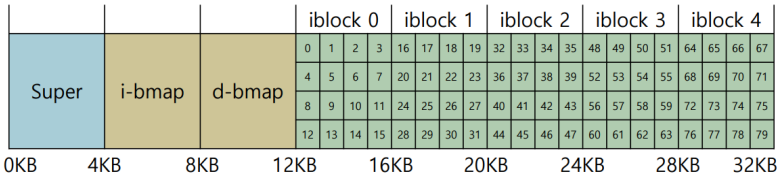
- example → 실제보다 간소화한 예시




Block size = 4kB
전체 Block = 4kB x 64 = 256kB partition

inode size = 256B
inode 개수 = $\frac{4kB}{256} = \frac{2^2 \times 2^{10}}{2^8} = 2^4 = 16$개 per block
전체 inode = 5개의 block x 16 = 80개 (최대 80개의 file 존재?)
↳ data block이 5개까지만 가능
⇒ inode도 5개까지만 저장 가능

## ▼ inode




### ▼ i-number

- inode의 offset, index 개념으로 사용됨
- 각 inode는 number 가짐 (명시적)
- i-number에 따라 해당 inode의 disk 위치를 계산할 수 있음

- example
  ⇒ file system 내에서는 어디에 위치?
  - read inode number 32
    - offset(inode 영역) : 32 * sizeof(inode) = 32 * 256B = 8KB
    - start address(inode table) : 12KB

  ⇒ offset(전체 file system) : 12KB + 8KB = 20KB
  ⇒ disk에서 어디에 위치?
  - disk → byte addressable하지 않음. sector 단위로 addressable

- 보통 하나의 sector ⇒ 512B
- sector address : (20 * 1024) / 512 = 40번째 sector에 존재 (disk 관점)
  (20 * 1024) → 20KB (전체 file system offset)

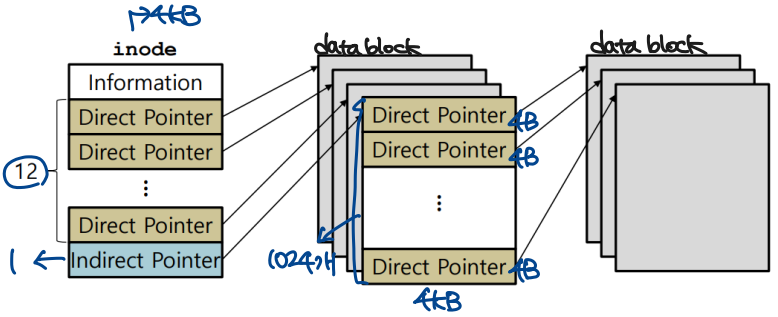
## ▼ inode는 data block의 어디를 나타내는가?

- inode → multi-level index
  - 고정된 수의 direct pointer와 하나의 indirect pointer를 포함함
- indirect pointer ⇒ pointer의 pointer
  - user data를 포함하는 block을 가리키지 않음

    → 각각 user data를 point하는 더 많은 pointer를 가진 block 자체를 pointer

    → file이 적당히 커지면 indirect block이 할당됨
  - double indirect pointer → file system이 어떻게 정의하냐에 따라 구현 내용이 달라짐
    - 더 큰 file 지원할 수 있음
    - indirect block의 pointer를 포함하는 block 가리킴

- example(double indirect pointer 미포함 예제)




# ▼ Directory organization

## ▼ Directory

file의 special type → dir 내부에서 inode 정보 얻어옴

- directory : inode의 type field ⇒ directory로 표시되어 있음

↔ file : '' ⇒ regular file

- (entry name, i-number) 쌍을 data block 내부에 list로 포함하고 있음

data block은 내부에 저장되어 있는 정보

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 12 | 2 | . |
| 2 | 12 | 3 | .. |
| 12 | 12 | 4 | foo |
| 13 | 12 | 4 | bar |
| 24 | 36 | 28 | foobar_is_a_pretty_long_name |

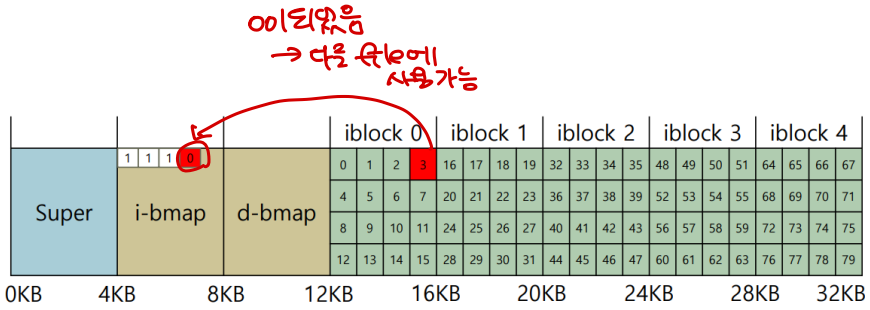
  - record length(reclen) : name의 총 byte 수
    - 보통 4의 배수로 표현 → 실제 byte 수와 가장 가까운 4의 배수만큼 할당 (실제 byte ≤ 할당한 byte)
  - string length(strlen) : name의 실제 길이

- delete file → dir 중간에 빈공간 만들어버릴 수 있음
  - disk는 작업 속도가 느리기에 file 지울 때 inode num만 0으로 바꿔버림
  - 나중에 새 file 삽입 시에 해당 record 크기 내에 등록이 가능하다면 0인 inode 정보에 새 파일 overwrite로 저장 가능
  - example

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 12 | 2 | . |
| 2 | 12 | 3 | .. |
| 0 | 12 | 4 | foo |
| 13 | 12 | 4 | bar |
| 24 | 36 | 28 | foobar_is_a_pretty_long_name |

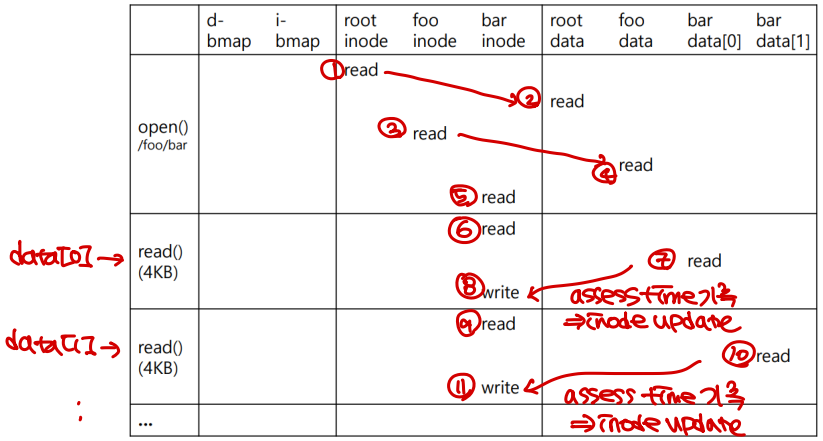
재사용 / 중복이 foo 포함 가능

- create file → free space를 위한 bitmap block 어떻게 사용?
  - file system : free한 inode를 bitmap 탐색으로 찾은 뒤 file 할당 (data block도 유사한 방식으로 찾은 뒤 할당)
    - 해당 inode bitmap을 1로 바꿈
  - 몇몇 file system : 새로운 file이 생성되고 data block을 필요로 할 때 연속적으로 free한 data block 찾음 → H/W적인 성능이 좋아짐




# ▼ Reading/Writng

## ▼ Reading a file from disk



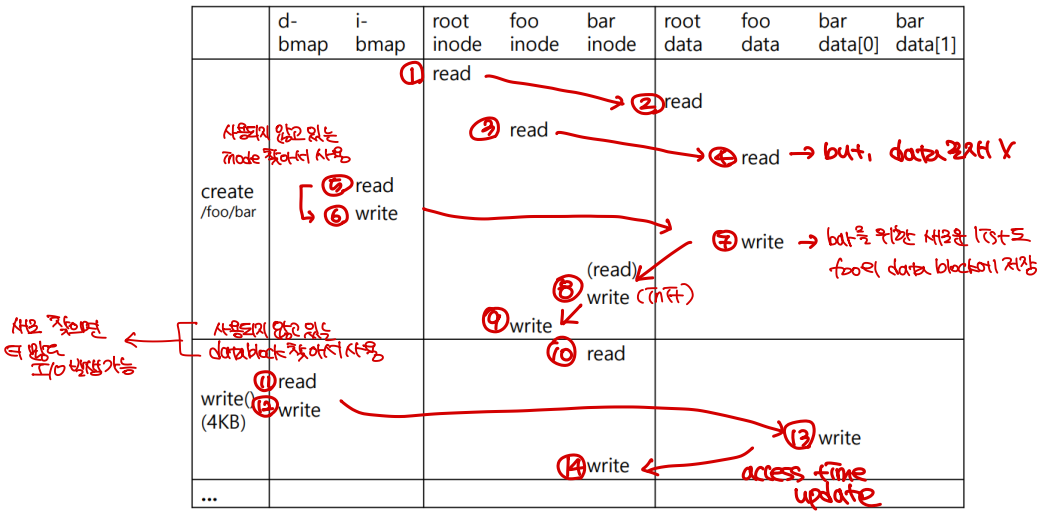

- open("/foo/bar", O RDONLY)
  1. /foo/bar에 있는 file inde 찾기
     a. root dir 접근 → root dir inode 먼저 읽음
        i. root : 보통 자주 접근하기에 고정된 inode num 사용함 (ex. 2)

           0 → no inode, 1 → bad blocks)
        ii. 하나 혹은 그 이상의 directory data block 읽음 → foo를 위한 entry 찾음 (foo의 i-number도 찾음)
     b. foo의 inode 읽음 → directory data 읽음
     c. bar의 inode 읽음
  2. memory에서 bar의 inode 읽음
  3. 접근 권한 있는지 확인
  4. process 당 open file table 내에서 해당 process의 file descriptor 할당
  5. user에게 반환
- read()
  - file의 첫번째 block read→ inode로!
  - inode가 가리키는 file의 metadat 중 마지막 접근 시간 update

- file offset update

- close()
  - fd → deallocated
  - disk I/O x

## ▼ Writing a file to disk




- write() ⇒ read보다 훨씬 복잡
  - 새 파일에 작성할 때
    - 각 write는 disk에 데이터를 쓸 뿐만 아니라 file에 할당할 block을 먼저 결정
    - 그에 따라 disk의 다른 구조를 업데이트해야 함 ex) data bitmap & inode
  - 각 write는 다섯개의 i/o를 생성함
    - data bitmap read
    - bitmap에 write
    - inode에 read
    - inode에 write
    - 실제 block에 write

## ▼ Caching and Buffering

- reading and writing files → 비쌀 수 있음 ⇒ 느린 disk에 많은 i/o 발생 가능
  - 모든 file open은 directory hierarchy 내에서 모든 level에 대해 적어도 두번의 read를 필요로 함(inode read + data read)

해결 → page cache (page 단위로 메모리 할당) → file, open, read를 효율적으로!!

  - 첫번째 open → dir inode와 data를 읽을 때 많은 i/o traffic 발생할 수 있음

    - 같은 file에 대해서 연속적인 file open이 일어난다면 cache에 hit
- write buffering
    - write delay에 의해 file system은 더 작은 i/o 집합으로 batch가 일어날 수 있음
    - 연속적인 i/o로 schedule될 수 있음 → 몇몇 wirte는 delay 되는 걸 막음