10. File System & Implementation

2023.07.13 신석철, 박영서, 임수형

Index

01 File and File System

- File

  논리적인 저장 단위, 관련된 정보 자료들의 집합에 이름을 붙인 것

  메모리는 주소를 통해 접근하지만 파일은 이름으로 접근

  일반적으로 비휘발성 보조기억장치에 저장

  연산 -> (생성, 읽기, 쓰기, reposition(lseek), 삭제, 열기, 닫기)

- MetaData

  파일을 관리하기 위한 각종 정보들. 파일 자체 내용은 아님파일 이름, 유형, 저장된 위치, 파일 사이즈,

  접근 권한, 소유자 등 파일에 대한 전반적인 정보

- File System

  운영체제에서 파일을 관리하는 소프트웨어 부분

  시스템 내의 모든 파일에 관한 정보를 제공하는 계층적 디렉터리 구조

  파일의 저장 방법, 관리, 보호 등을 담당

01 File and File System

- Directory file

  디렉토리도 하나의 파일

  하지만 디렉토리 파일의 내용은 디렉토리 파일 내에 위치한 파일들의 메타데이터

- 디렉토리 연산

  search for a file, create a file, delete a file

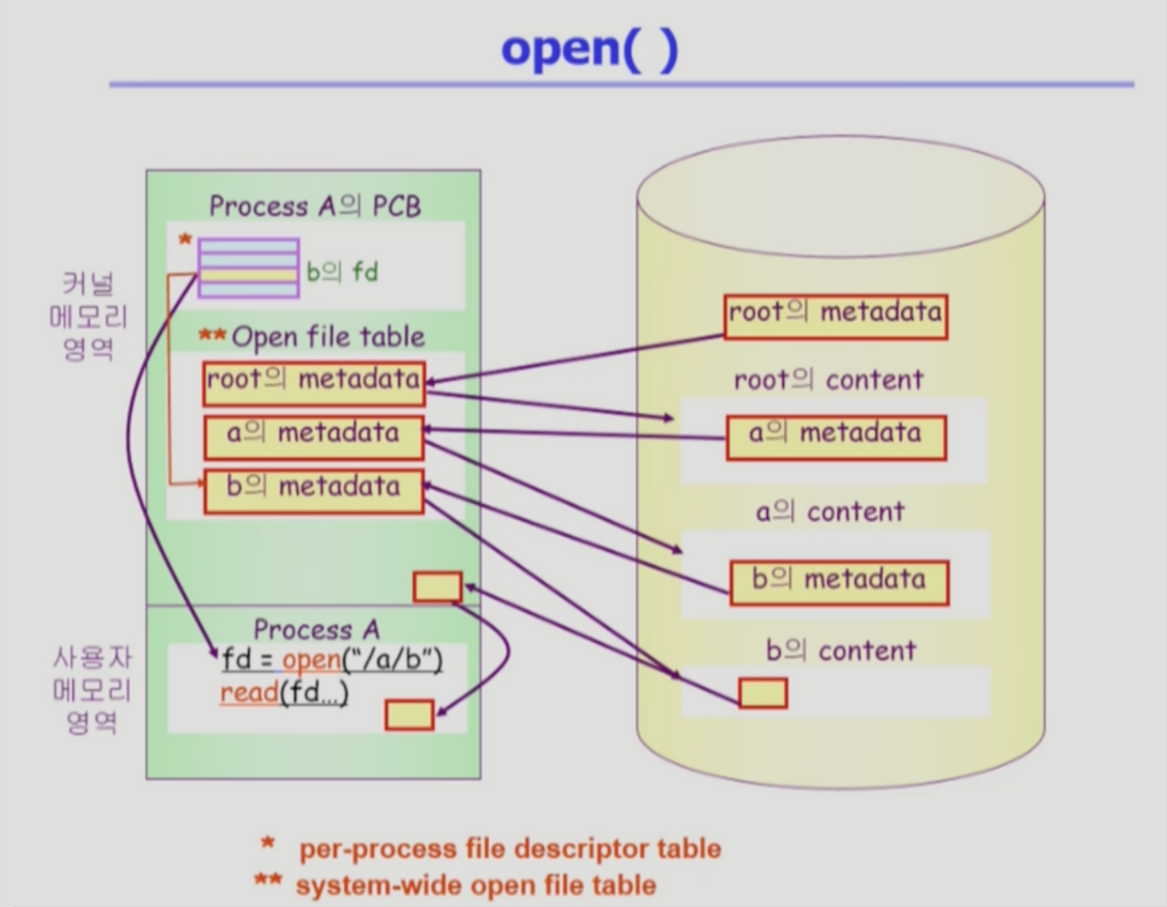
  list a directory, rename a file, traverse the file system

- Partition (Logical Disk)

  논리적 디스크 _ 하나의 물리적 디스크를 여러 partition으로 나누면 각각이 논리적 디스크가 됨

  물리적 디스크를 파티션으로 구성한 뒤 각각의 파티션에 file system을 깔거나 swapping 등 다른 용도로 사용할 수 있음

01 파일의 Open 연산 과정

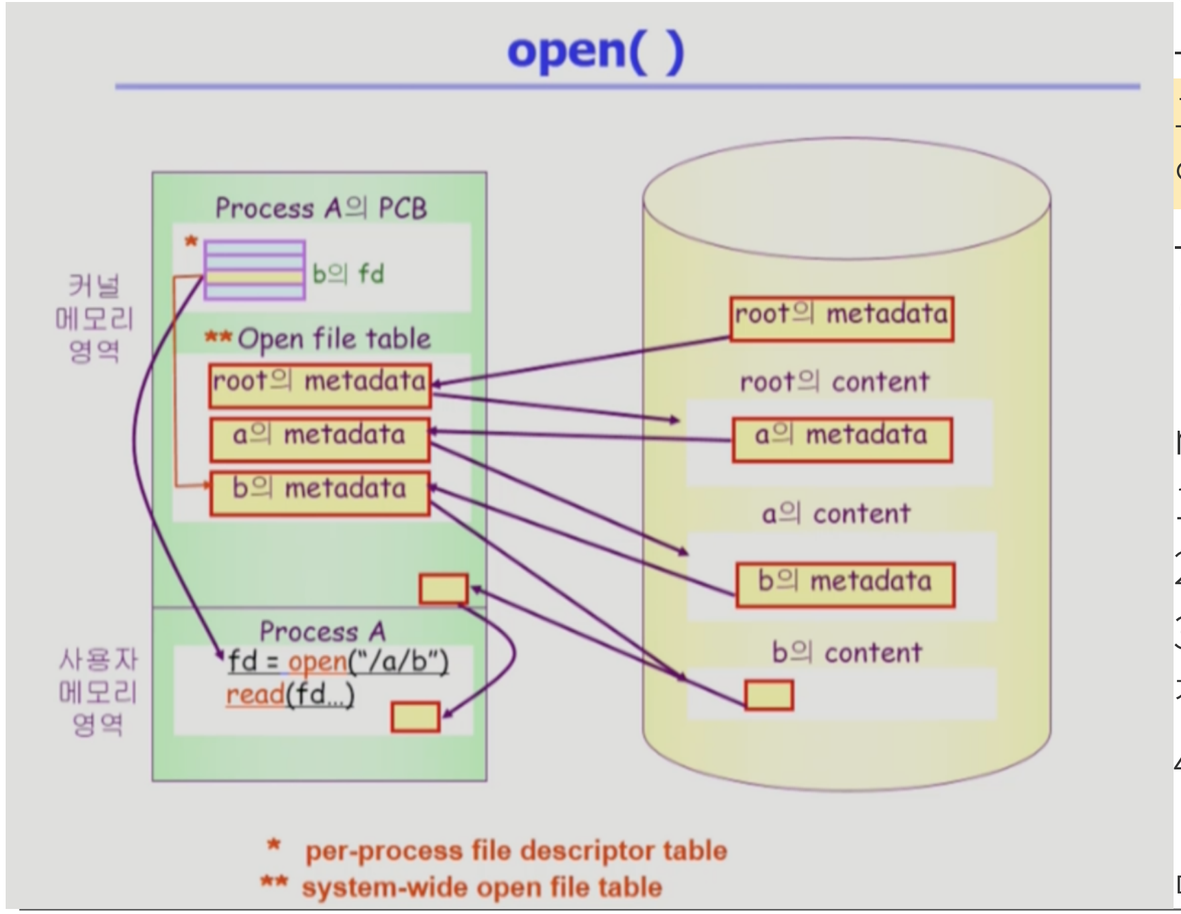


- 파일의 메타데이터를 디스크에서 메모리에 올려두는 것

open(/a/b) -> b 의 메타데이터가 메모리에 올라옴

1. 사용자 프로그램이 시스템콜(open도 시스템콜)
0. root 메타데이터 메모리에 가져오기
2. root를 먼저 open해 a의 메타데이터를 가져옴
3. a의 메타데이터에 있는 a의 내용을 보고 a를 open
4. a를 open해서 b의 메타데이터를 메모리에 가져옴
5. b의 메타데이터를 open하고 메모리에 올리기

01 파일의 Open 연산 결과 반환



open 과정 후 시스템콜을 했기 때문에 결과값을 return 해야 함
- 각 프로세스마다 오픈한 파일들의 메타데이터에 대한 포인터를 가지고 있는 배열이 있다. 거기에서 있는 거기에서 요청한 파일의 포인터의 인덱스를 반환한다
- 해당 배열에서 몇 번째 인덱스인지 리턴 (file descriptor)
- 사용자 프로세스는 fd만 가지고 읽기 쓰기 요청 가능

read(fd)
1. 프로세스 A가 해당 fd를 가지는 파일을 읽어오라고 하면
2. A의 PCB에 가서 discriptor에 대응하는 파일의 메타데이터 찾음
3. b의 내용을 읽어 올 것인데 그 내용을 읽어서 사용자 프로그램에 직접 주는 것이 아니라 운영체제가 자신의 메모리 공간 일부에 먼저 저장
4. 사용자 프로그램에게 그 내용을 copy해서 전달

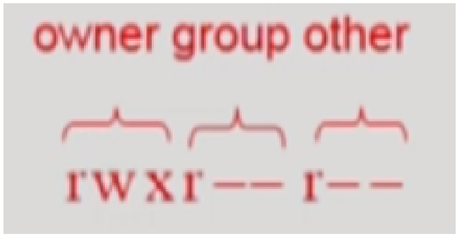
만약 다른 프로세스가 동일한 파일의 동일한 위치를 요청하면 다시 디스크까지 가지 않고 운영체제가 읽어 놓은 것을 전달 -> (버퍼캐싱)

02 File Protection

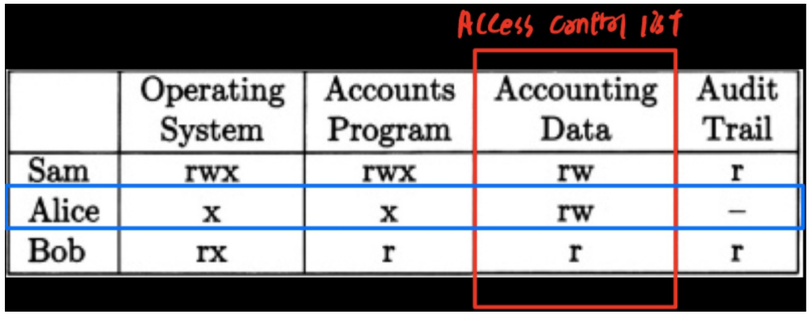
각 파일에 대해 누구에게 어떤 유형의 접근(read, write, execution)을 허락할 것인가?

	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	rw	r
Alice	x	x	rw	–
Bob	rx	r	r	r

Access Control Matrix



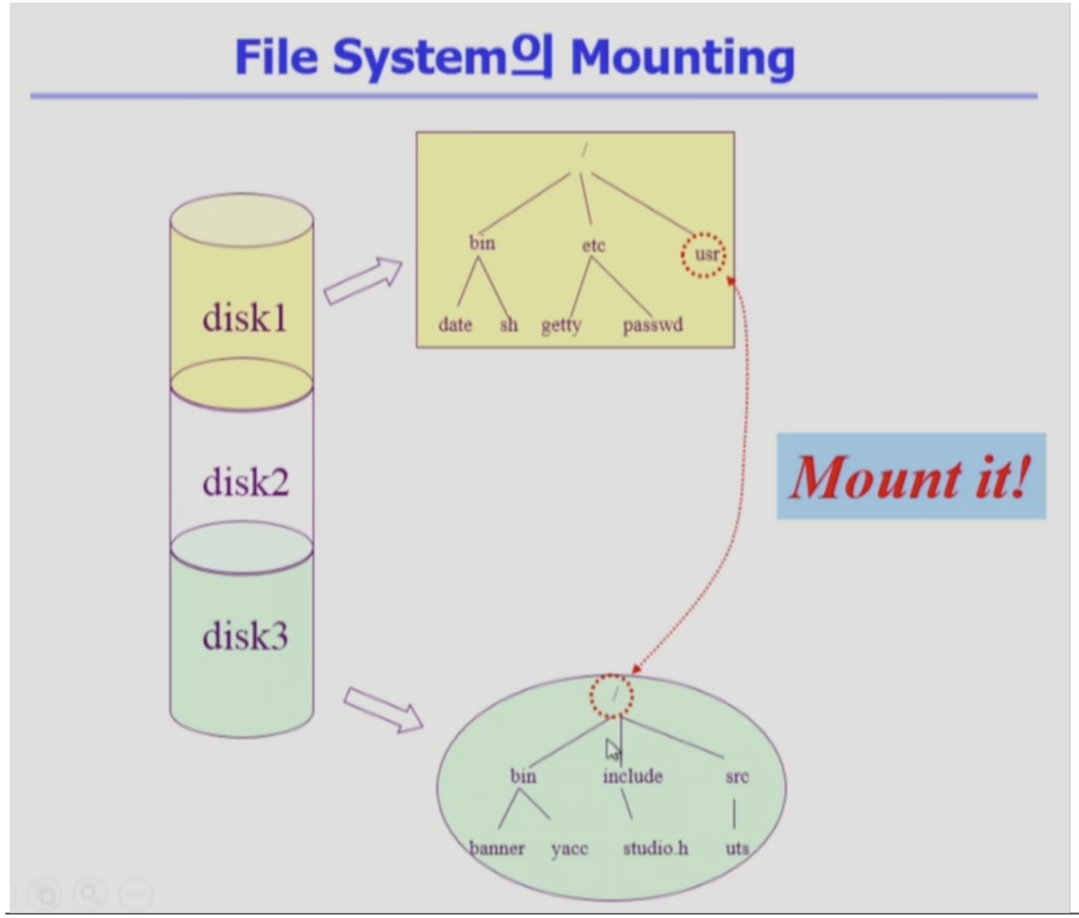
Grouping
사용자에 대해서 owner, group, public의 세 그룹으로 구분
각 파일에 대해 세 그룹의 접근 권한을 3비트씩으로 표시



Access Control list _ 파일 기준
Capability _ 사용자 기준

Password
파일마다 password를 두는 방식
암기와 관리 문제 발생

02 File System의 Mounting



하나의 물리적인 디스크를 파티셔닝을 통해서 여러 논리적인 디스크로 나눌 수 있음
각각의 논리적인 디스크에는 파일 시스템을 설치해서 사용할 수 있음

처음 디스크는 root를 통해 접근할 수 있는데 다른 파티션에 있는 디스크에 접근해야 한다면 Mounting이라는 연산 사용

루트 파일 시스템의 특정 디렉토리의 이름에 다른 파티션의 파일 시스템을 Mount 해주면 그 디렉토리 접근이 다른 파일 시스템의 루트 디렉토리에 접근하는 형식이 되는 것

서로 다른 파티션에 존재하는 파일 시스템을 접근할 수 있게 된다

02 File access (파일 접근)

- 순차 접근
  카세트 테이프를 사용하는 것처럼 접근
  읽거나 쓰면 offset은 자동적으로 증가

- 직접 접근
  LP레코드판과 같이 접근
  파일을 구성하는 레코드를 임의의 순서로 접근할 수 있음

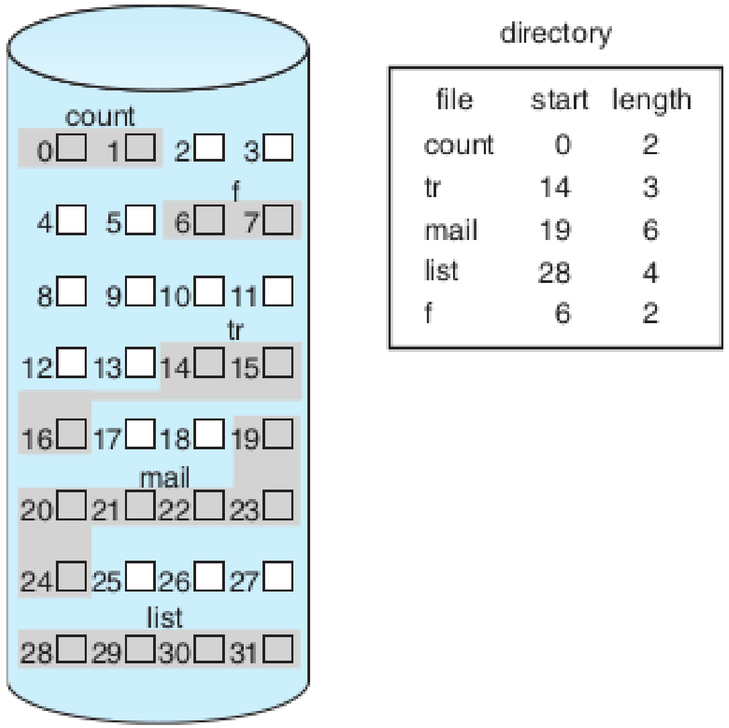
A B C 가 있는데, A를 보고 C를 보고 싶다
순차 접근 -> ABC 순으로 접근
직접 접근 -> AC 순으로 접근

02 Allocation of File Data in Disk

Contiguous Allocation

n번에서부터 연속적으로 m개를 할당

- 단점
  - External Fragmentation
  - File grow의 어려움
    - file은 사용하면서 더 커져야 할 수도 있다.
      - 연속 할당의 경우 뒤 공간이 없어서 더 키우지 못할 수도 있다
    - 이를 방지하기 위해 물론 미리 뒤에 여유 공간을 할당해놓을 수도 있다.
      - 근데 file 생성시 얼마나 큰 hole을 배당할지?
      - grow 가능 vs 낭비 (internal fragmentation)
- 장점
  - I/O가 빠르고 Random Access가 가능하다
    - 한번 seek/rotation으로 많은 바이트를 transfer 가능
    - Real-time file용, 혹은 이미 run 중인 process의 swapping용
    - 중간에 있는 블럭까지 순차적으로 접근할 필요없이 바로 접근 가능



directory

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

Figure 14.4 Contiguous allocation of disk space.

02 Allocation of File Data in Disk

Linked Allocation

파일에는 start, end만이 주어져 있고, 각 블럭에 가면 데이터와 함께 다음 블럭이 어딨는지, 그 포인터가 함께 있다

- 장점
  - External fragmentation이 발생하지 않는다.
- 단점
  - Random access 불가 : 반드시 시작점부터의 순차적인 접근만이 가능하다.
  - Reliability 문제
    - 한 sector가 고장나 pointer가 유실되면 많은 부분을 잃게 된다 (그 뒤쪽에는 정상적 접근이 불가)
  - Pointer를 위한 공간이 block의 일부가 되어 낮은 공간 효율성
    - 512-bytes/sector, 4-bytes/pointer.
    - 그런데 디스크에 저장할 때의 용량 단위는 (인터페이스 상) 512-bytes의 배수로 정해져있다.
    - 그런데 4-bytes를 포인터를 위해 사용해야하기 때문에 한 섹터에 저장할 수 있었을 데이터를 두 섹터에 나눠서 저장해야하는 경우가 발생할 수도 있다
- 변형 : File-Allocation Table(FAT) 파일 시스템
  - 포인터를 섹터 내가 아닌 별도의 위치에 보관하여 reliability와 공간효율성 문제 해결

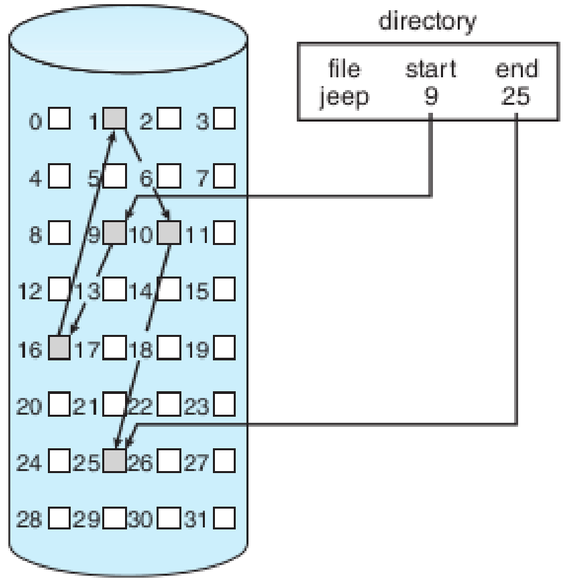


Figure 14.5 Linked allocation of disk space.

02 Allocation of File Data in Disk

Indexed Allocation

직접 접근을 가능하게 하기 위해 디렉토리에 파일의 블럭이 아니라 해당 파일의 인덱스 블럭을 저장한다.
이 인덱스 블럭에는 이 파일의 몇 번째 블럭이 어디에 있는지, 그 위치 포인터를 담아놓는다.

- 장점
  - External fragmentation이 발생하지 않음
  - Random access가 가능
- 단점
  - 많은 file들은 실제로는 매우 작은데, Indexed Allocation에서는 이러한 파일들을 위해서도 두 섹터 이상을 할당
  - 너무 큰 파일의 경우 한 블럭에 모든 인덱스를 저장하기에는 부족
    - 해결 1) linked scheme
      - 인덱스 블럭의 끝에 다른 인덱스 블럭의 위치를 담음
    - 해결 2) multi-level index
      - 최초 n개의 인덱스 블럭의 각 내용이 다음 레벨의 인덱스 블럭을 가리키게 한다. (이후는 데이터)

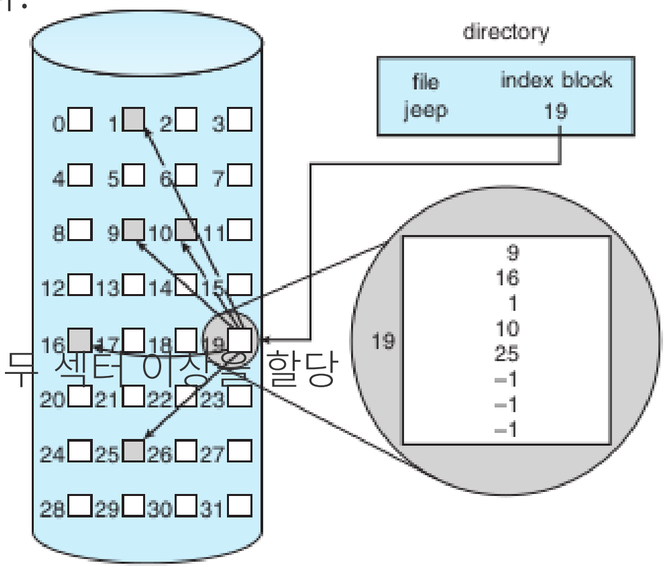


Figure 14.7 Indexed allocation of disk space.

02 UNIX File System & FAT File System

UNIX File System

Boot Block	Super Block
부팅에 필요한 정보 (bootstrap loader)	파일 시스템에 관한 총체적 정보
Inode List	**Data Block**
파일 이름을 제외한 파일의 모든 메타 데이터 디렉토리가 메타데이터를 다 가지고 있지 않음 index allocation 변형해서 사용	파일 이름의 실제 내용. 여기의 directory는 파일 이름과 inode index를 가지고 있음

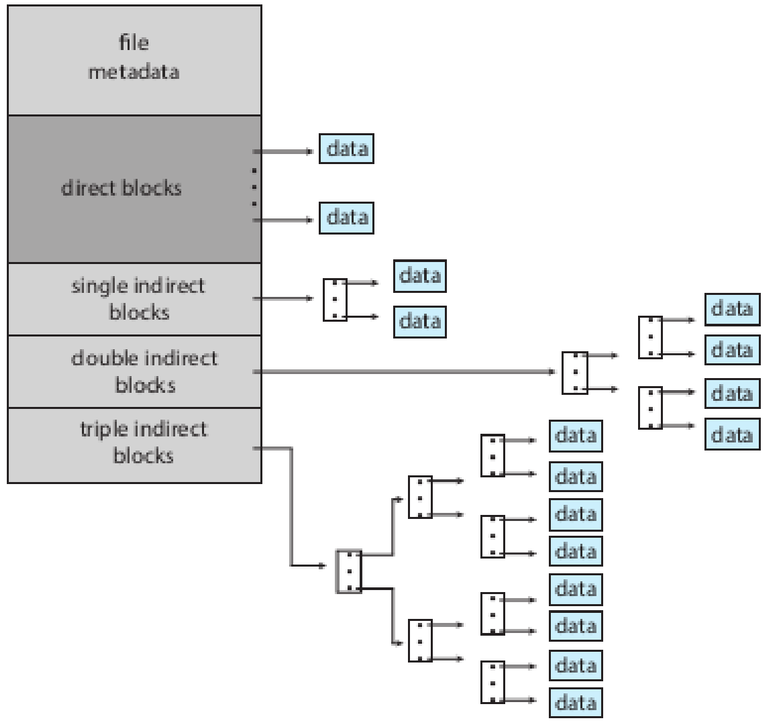


Figure 14.8 The UNIX inode.

02 UNIX File System & FAT File System

FAT File System

Boot Block	FAT	Root Directory	Data Block
-	파일의 위치 데이터	-	디렉토리는 파일의 이름을 비롯한 모든 메타데이터들을 가지고 있음. 파일의 첫 번째 위치가 어디인지도.

FAT은 메모리에 캐시될 수 있다. 메모리에 캐시된 FAT을 가지고
해당 블럭의 위치를 빠르게 계산하면, Disk head를 여러 번 움직이지 않고
random access가 가능.

만약 FAT이 캐시되어 있지 않다면 해당 블럭의 위치를 찾기 위해
Disk head를 여러 번 움직여야 해서 성능 하락이 발생할 수 있다.

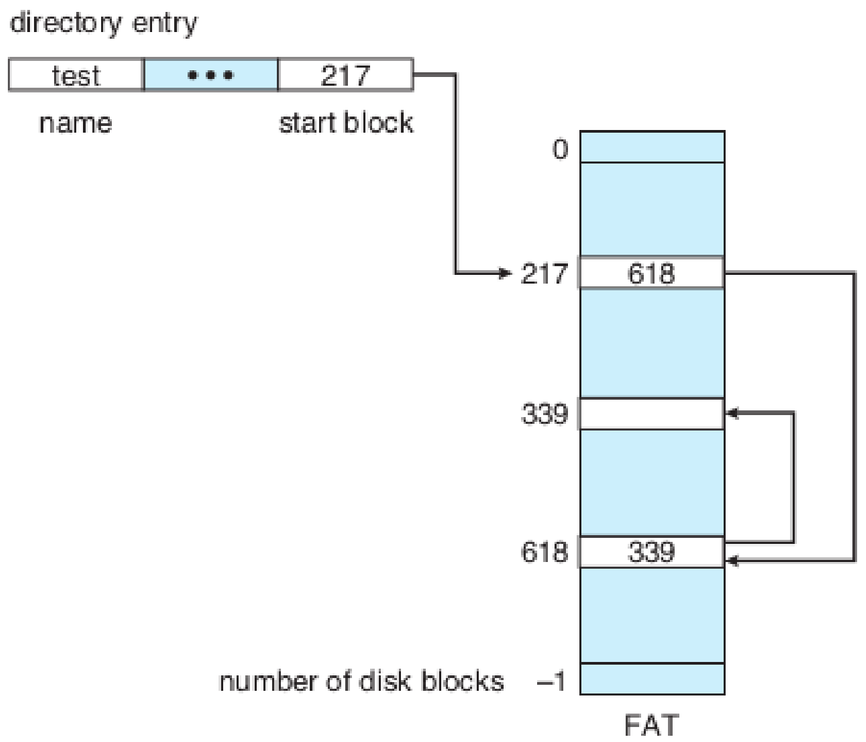


Figure 14.6 File-allocation table.

02 Free-Space Management

이때까지가 자료가 든 공간들을 관리하는 방법들이었다면, 이제는 가용 공간을 관리하는 방법들이다.

Bit map or bit vector
- 디스크의 크기만큼의 bit vector를 만든다.
- 한 비트가 0이면 이에 해당하는 블럭은 free, 1이면 occupied
- 부가적인 공간을 필요로 하다
- 연속적인 n개의 free block을 찾는 데 효과적

Linked list
- 모든 free block들을 링크로 연결한다
- free-space list head로부터 시작해서 다음 가용 공간을 가리키게 함
- 연속적인 가용공간을 찾는 것은 쉽지 않음
- 공간 낭비가 없다

02 Free-Space Management

Grouping
- Linked list 방법의 변형
- 첫 번째 free block이 n개의 pointer를 가짐
- n-1 pointer는 free data block을 가리킴
- 마지막 pointer가 가리키는 block은 또 다시 n pointer를 가짐

Counting
- 연속적인 빈 블럭을 찾는 데에 유리
- 프로그램들이 종종 여러 개의 연속적인 block을 할당/반납한다는 성질에 착안
- (first free block, # of contiguous free blocks)를 쌍으로 관리, 몇 번째 블럭부터 몇 개가 가용 블럭인지를 표시.

03 Directory Implementation

- Linear List (linked)

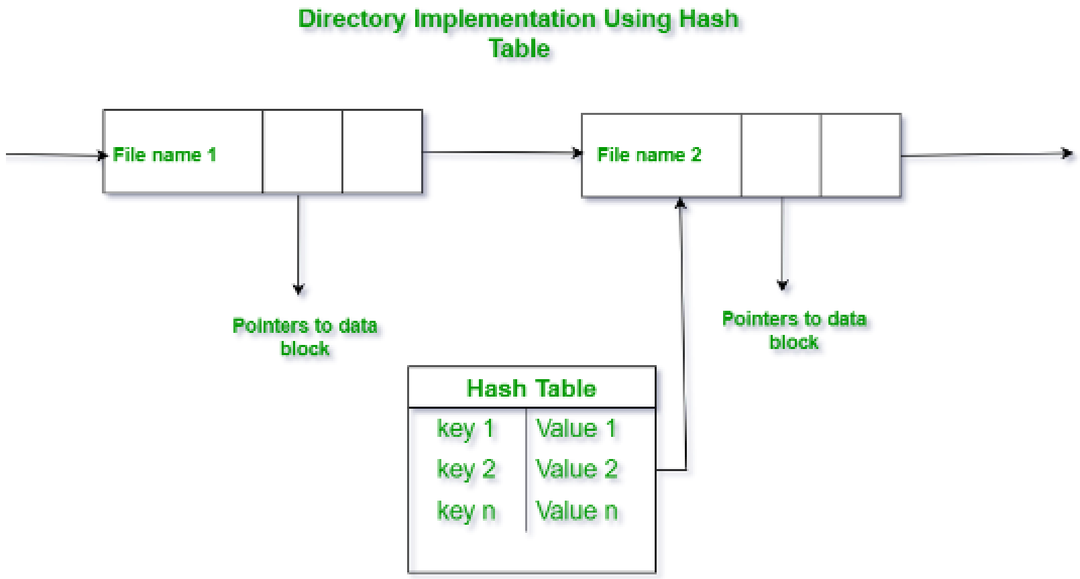
- 선형 탐색시간.

- 메타 데이터 크기 고정.

- Hash Table

- key : File Name / value : pointer

- Hash 사용에 따라 Collision 발생가능



- Metadata

- 디렉토리에 직접 보관

- 포인터를 두고 일부 다른 곳에 보관

-> 파일시스템에 따라 상이

- Long file name

파일 메타데이터 리스트에서 각 entry는 일반적으로 고정 크기를 가짐. LFN인 경우 entry의 마지막 부분에 포인터를 두고. 나머지 부분은 directory 파일 일부에 둠.

01 VFS & NFS

- Virtual File System

실제 파일시스템에 관계 없이 공통된 인터페이스로 파일시스템에 접근하도록 하는 계층

VFS interface(OS 의 Layer)

티스크의 각 파티션마다 다른 파일시스템 채택가능.

-> 각기 다른 명령어를 호출 해야함.

https://aroundck.tistory.com/774

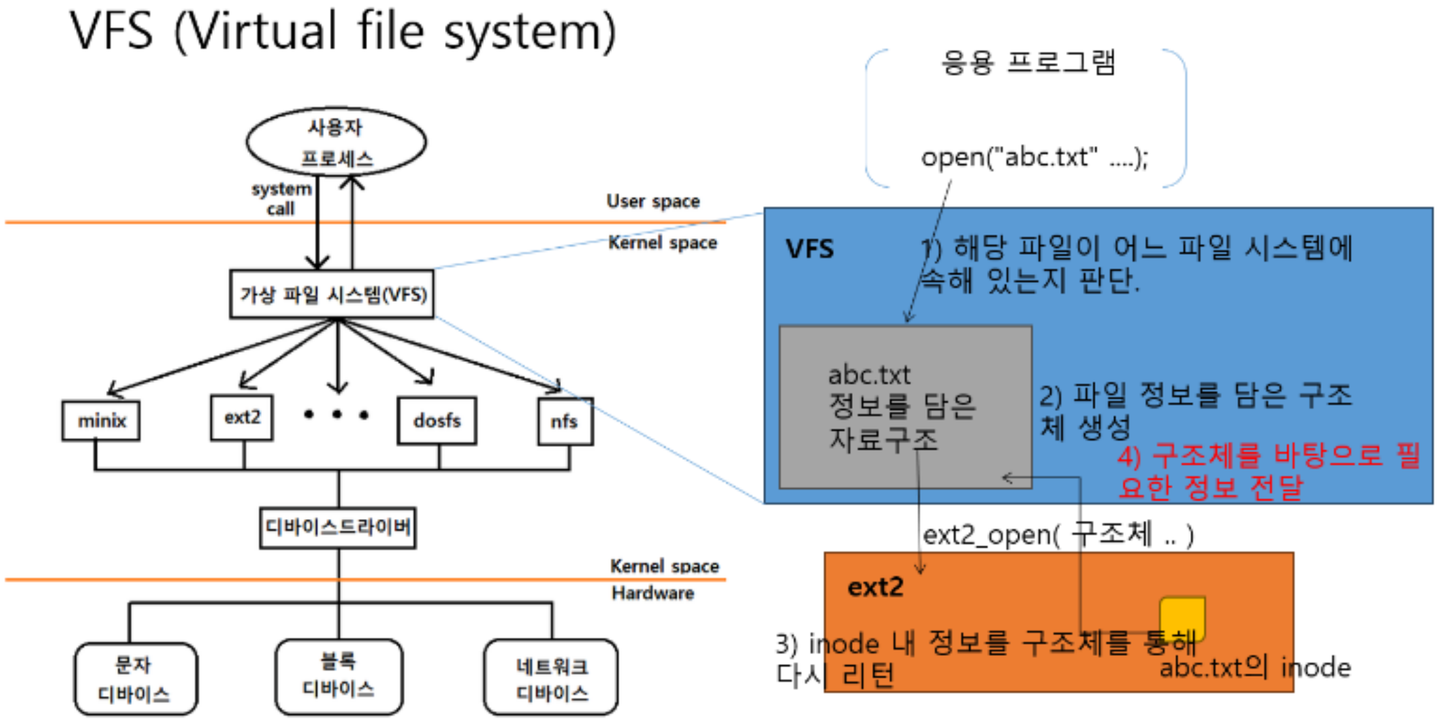
- Network File System

분산 시스템에서는 네트워크를 통해 파일이 공유될 수 있다.

NFS는 분산 환경에서의 대표적인 파일 공유 방법

마치 로컬 저장소에 있는 것처럼 파일에 접근한다.

주의해서 써야함 https://www.time-travellers.org/shane/papers/NFS_considered_harmful.html



III. List of Concerns

a. Time Synchronization
b. File Locking Semantics
c. File Locking API
d. Exclusive File Creation
e. Delayed Write Caching
f. Read Caching and File Access Time
g. Indestructable Files
h. User and Group Names and Numbers
i. Superuser Account
j. Security
k. Unkillable Processes
l. Timeouts on Mount and Unmount
m. Overlapping Exports
n. Automount

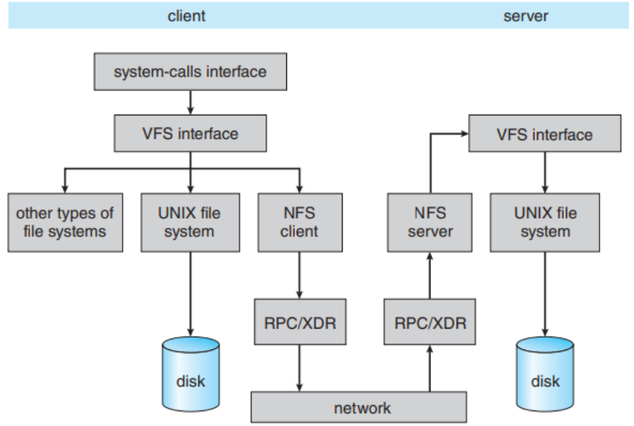


Figure 15.8 Schematic view of the NFS architecture.

Systems

01 Performance - Buffer cache, Page cache

- Page cache

(가상)메모리를 관리하는 페이징 기법.

page cache : 파일을 페이지 단위로 메모리에 올려둘 때.

<-> 프로세스 페이지가 swap area에 내려가 있는가.

- Buffer cache

파일 시스템을 통한 I/O 연산시 File 사용의 Locality를 고려해서 block을 메모리의 특정 영역인 Buffer Cache에 올려둠

파일에 엑세스 할 때, 파일 시스템 말고 가상 메모리와 인터페이스 하게 해보자.

Memory-mapped I/O 를 활용해 통합이 가능하다. -> Unified Buffer Cache

(파일의 일부를 가상 메모리에 매핑하여 파일 입출력 수행)

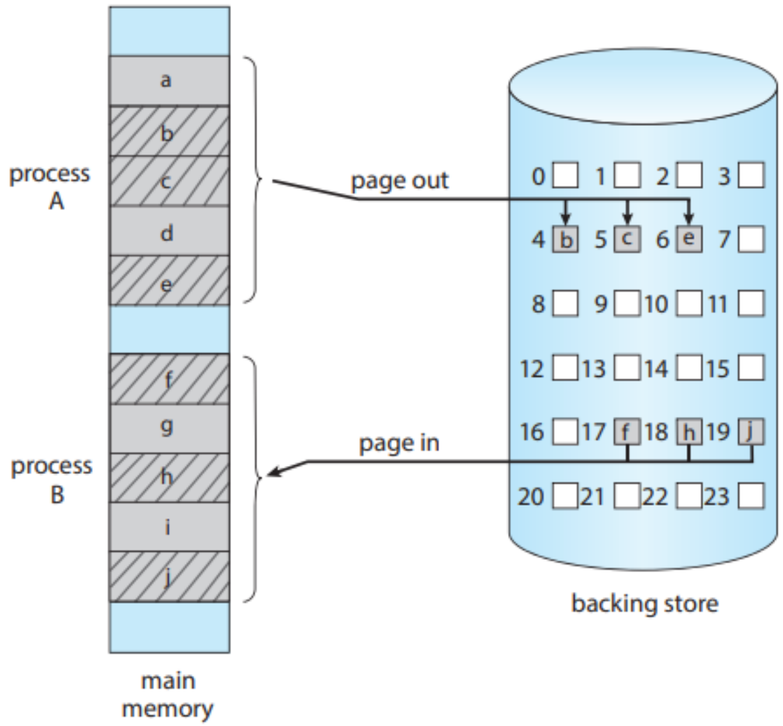


Figure 9.20 Swapping with paging.

01 Performance - Unified Buffer Cache

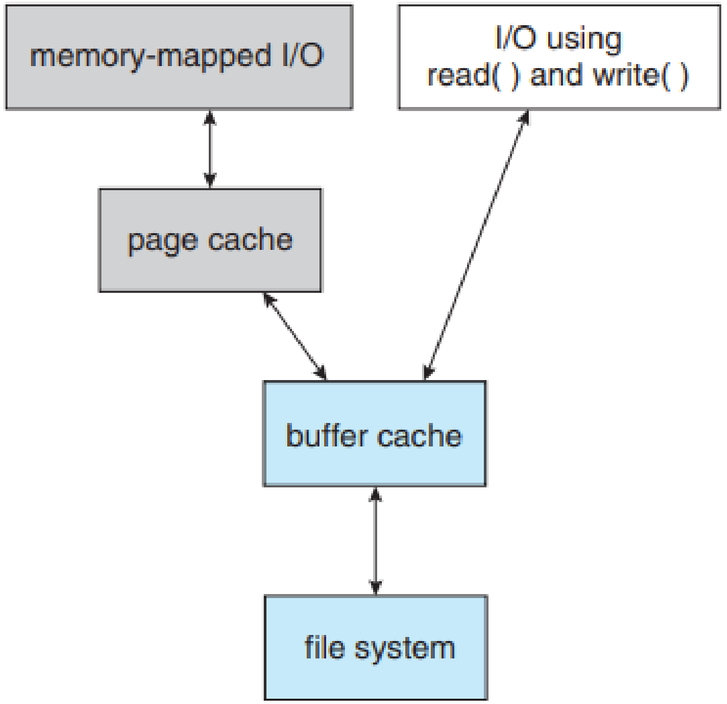


Figure 14.10 I/O without a unified buffer cache.

페이지 캐시와 버퍼 캐시 둘 다에 이중 캐쉬 되는 문제

파일의 일부를 가상메모리에 맵핑 시켜놓고, 해당 메모리 접근 연산은 파일 입출력을 수행하게 한다.

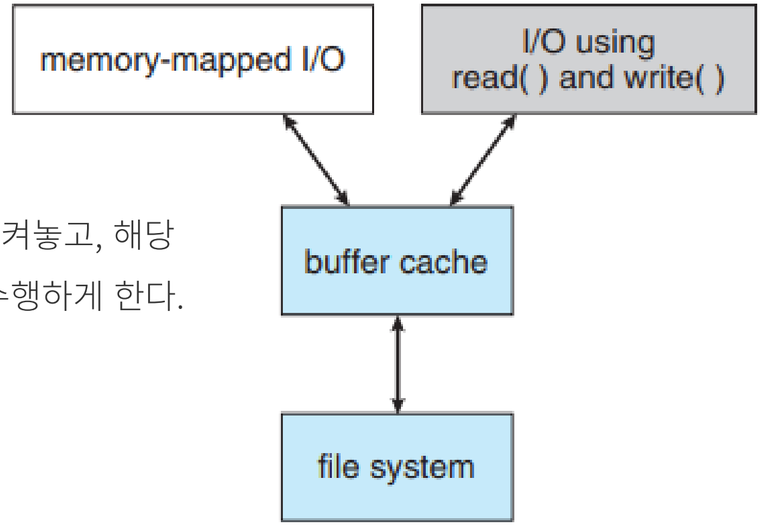


Figure 14.11 I/O using a unified buffer cache.

가상 메모리 시스템이 파일시스템 데이터를 관리할 수 있다.

01 프로그램 실행

1. 프로그램 시작시 각 프로그램의 독자적인 주소 공간이 만들어짐 : Virtual Memory

2. 주소변환에 의해 물리적 메모리에 올라감. 안쓰거나 후순위는 swap area로 내려감

이때, code block은 readonly로 write할 필요가 없기 때문에, swap area에 또 써줄 필요가없음
즉, 프로세스의 코드 블럭은 파일 형태로 프로세스 주소 영역에 매핑됨. 이게 Memory mapped i/o를 쓰는 것과 똑같다.
파일을 매핑해서 쓰는 대표적인 예
+ 프로그램이 실행되다가, 파일을 읽어야 할 경우
system call 이후, 데이터파일의 일부를 프로그램 주소공간에 맵핑해놓음

-> 파일을 접근하는데 본인 메모리에 올라와있기 때문에 시스템 콜 없이 접근가능