11장. 파일 시스템 구현

목표

- local 파일 시스템 및 디렉토리 구조의 구현을 설명
- remote 파일 시스템 구현을 설명
- 블록 할당과 자유 블록 알고리즘 논의

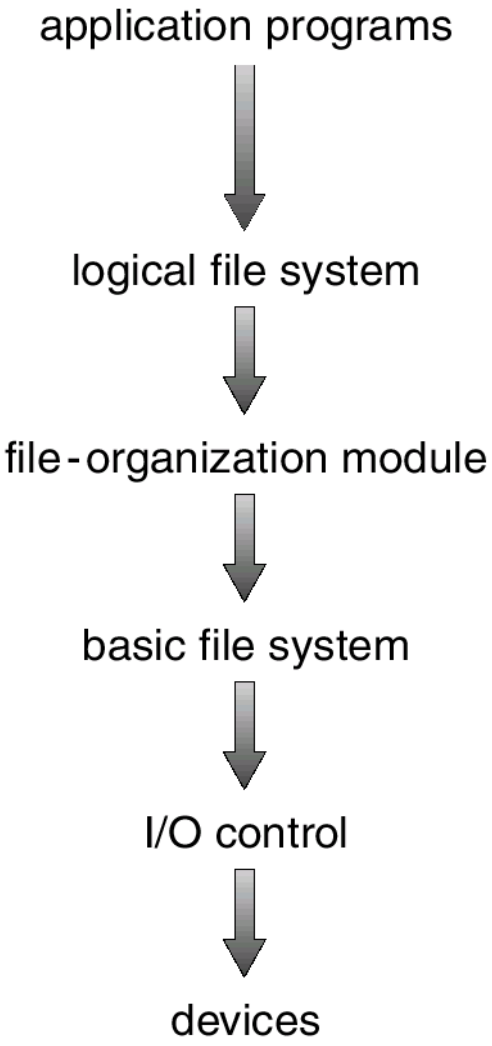
11.1 File-System 구조

- File system은 보조 저장장치(디스크)에 위치.
  - 블록 단위 전송 → I/O 효율성 향상
  - block size: one or more sectors
  - sector size: 32 ~ 4KB (usually 512B)
- File systems은 디스크에 대한 효율적이고 편리한 접근을 제공함

- 파일 시스템의 2가지 설계 문제
  - 파일 시스템이 사용자에게 보여지는 방법을 정의
    - 파일, 파일 속성, 파일 연산, 디렉토리 구조
  - 논리 파일 시스템을 물리적 보조저장장치로 맵핑하는 방법 설계
    - 알고리즘, 자료구조

- 파일 시스템의 계층적 설계 ➔ layered file system

계층적 파일 시스템(Layered File System)



메타데이터 정보 관리, 심볼릭 파일 이름
디렉토리 구조, file control block(FCB) - inode
protection/security

logical block을 physical block으로 변환
file 공간 관리자

device driver에게 명령어 제공
physical block(drive, cylinder, track, sector) 사용
memory buffers와 caches 관리

device driver, interrupt handler
- 메모리와 디스크 간의 데이터 전송

File systems의 종류

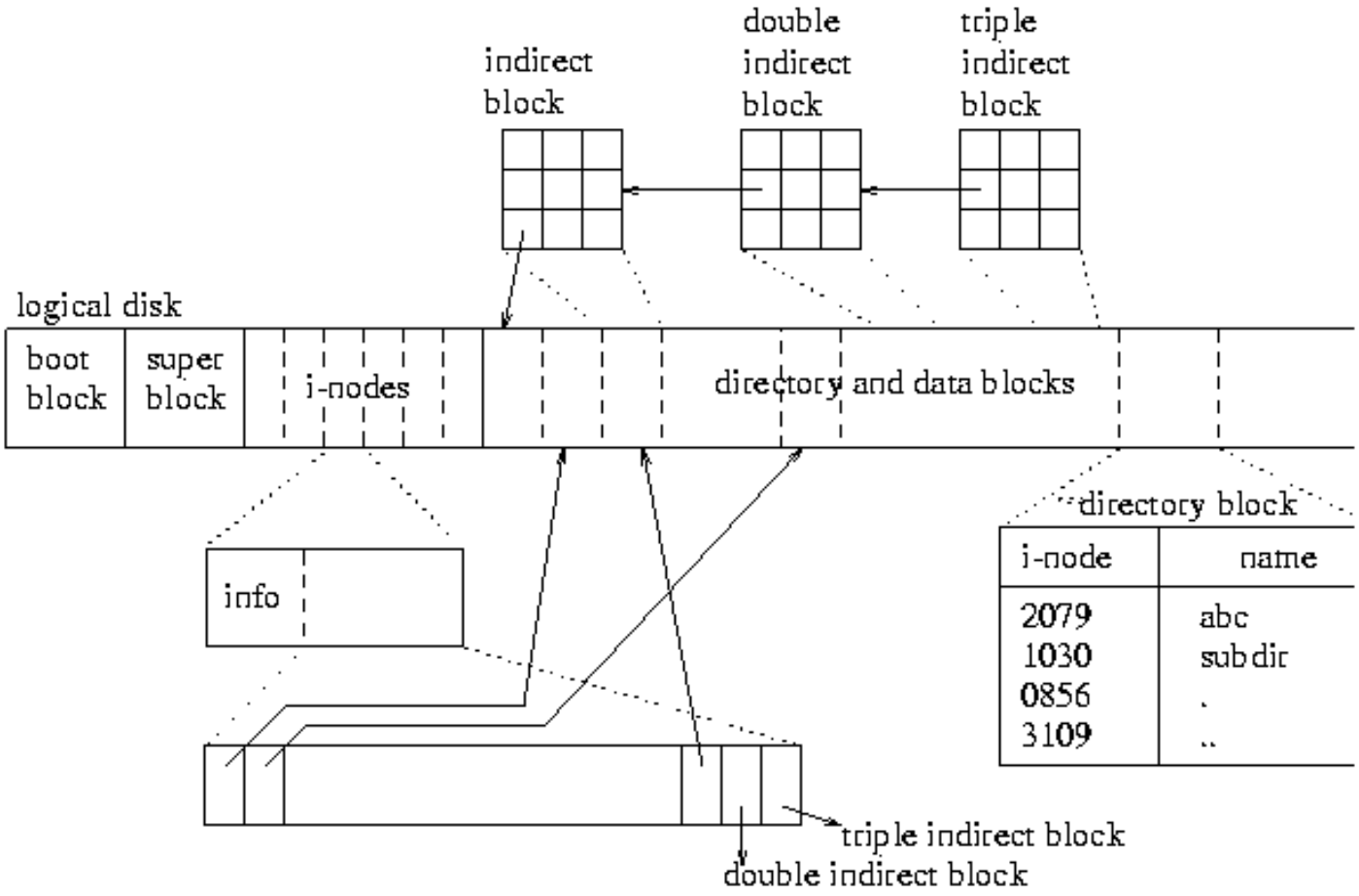
- 현재 많은 파일 시스템들이 사용되고 있음
  - 대부분의 운영체제가 2개 이상의 파일 시스템 지원

- 이동 가능한(removable) 파일 시스템 – CD-ROM, DVD, floppy disk
  - CD-ROM : ISO 9660 format
- 디스크 기반 파일 시스템
  - **UNIX** : Berkeley Fast File system(FFS)에 기반을 둔 Unix 파일 시스템
  - **Windows** : FAT, FAT32, NTFS
  - **Linux** : ext2, ext3, ext4(extended file system),
    40개 이상의 파일시스템 지원
- 분산 파일 시스템
  - 파일 서버에 있는 파일시스템을 네트워크로 연결된 클라이언트 컴퓨터에서 마운트하여 사용함

11.2 File-System 구현

- 파일 시스템 구현을 위한 자료구조
  - on-disk 자료구조
  - in-memory 자료 구조
- On-disk 구조
  - 부트 제어 블럭(boot control block) (per 파티션/볼륨)
    - UNIX – 부트 블럭
  - 볼륨 제어 블록(volume control block) (per 파티션/볼륨)
    - contains # of blocks, block size, free block count, free block pointer, free FCB count, FCB pointer …
    - UNIX - superblock, Windows NTFS - master file table
  - 디렉토리 구조 (per 파일시스템)
    - file names, associated inode numbers 포함 (NTFS: in master file table)
  - 파일 제어 블록 (per 파일)
    - UNIX's inode (NTFS: in master file table)
  - 파일 데이터

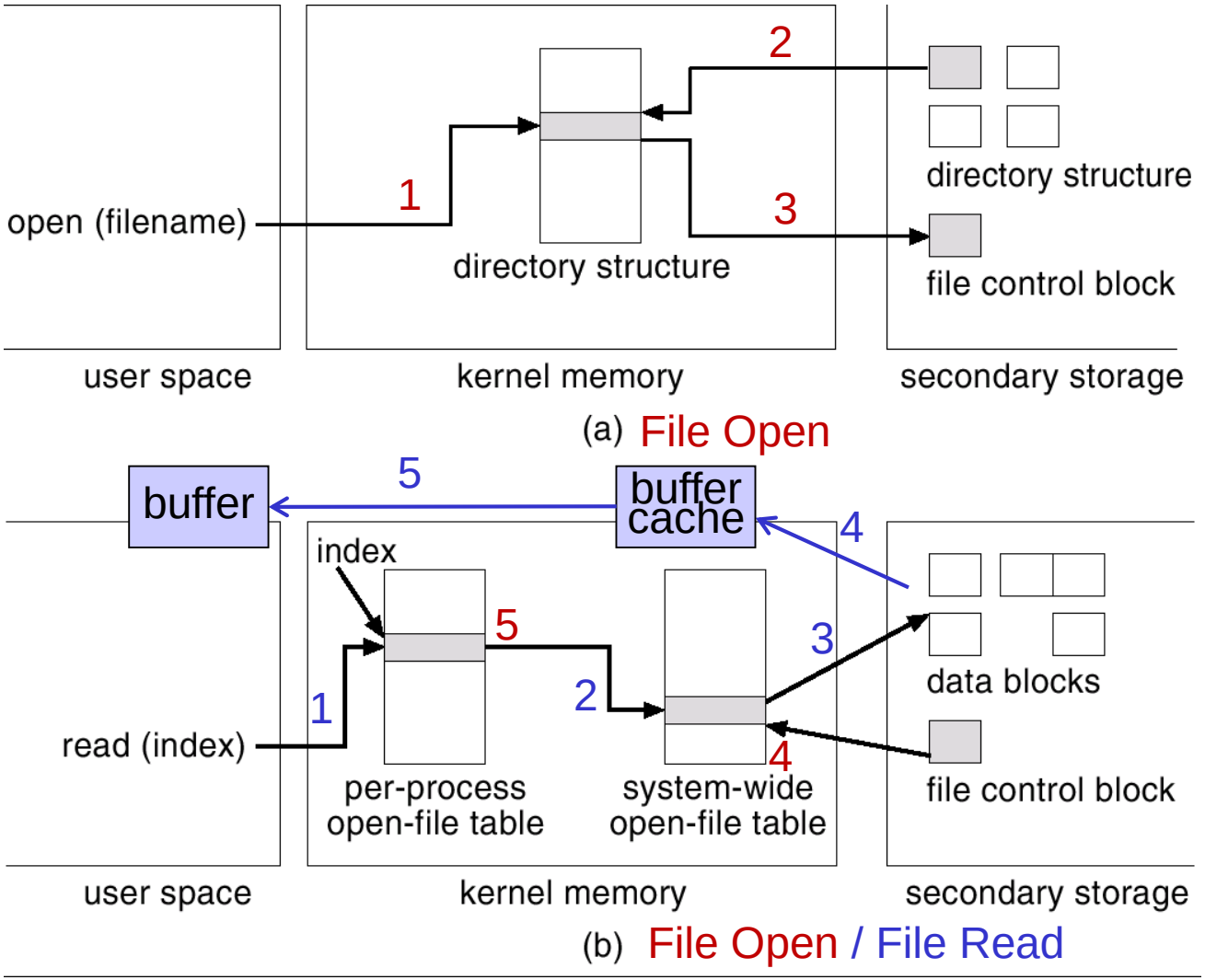
UNIX file system*



In-Memory 자료구조

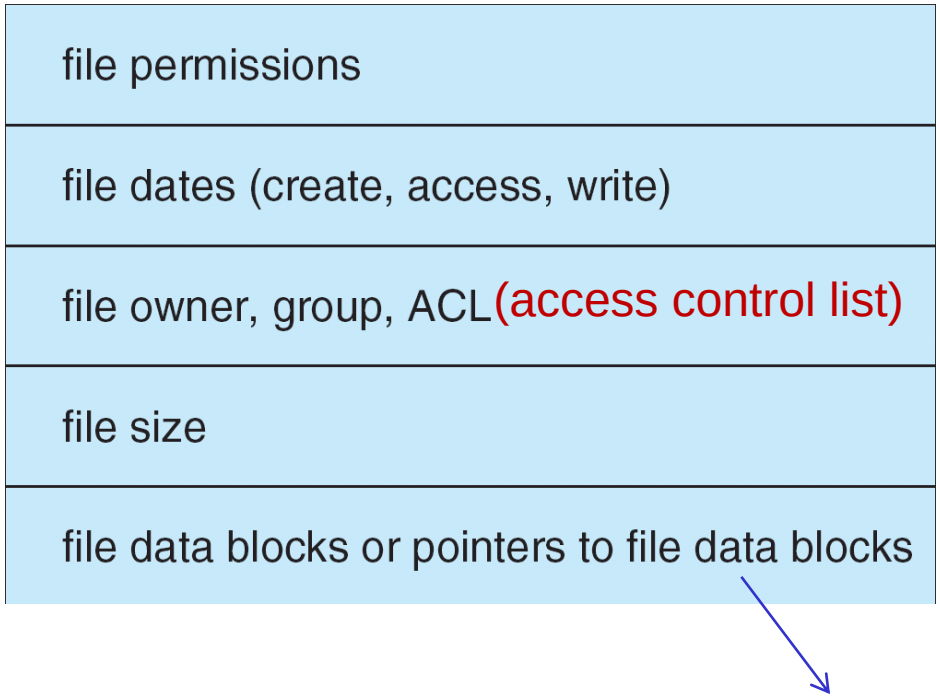
- In-memory 구조
  - in-memory mount table (partition table)
  - in-memory directory structure
    - 최근에 접근한 디렉토리들에 대한 정보
  - system-wide open file table
    - open file들에 대한 파일 제어 블럭(FCB) 복사본
  - per-process open file table
    - the system-wide open table의 entry에 대한 포인터
    - UNIX - file descriptor, Windows - file handle

In-Memory File System 구조



File Control Block

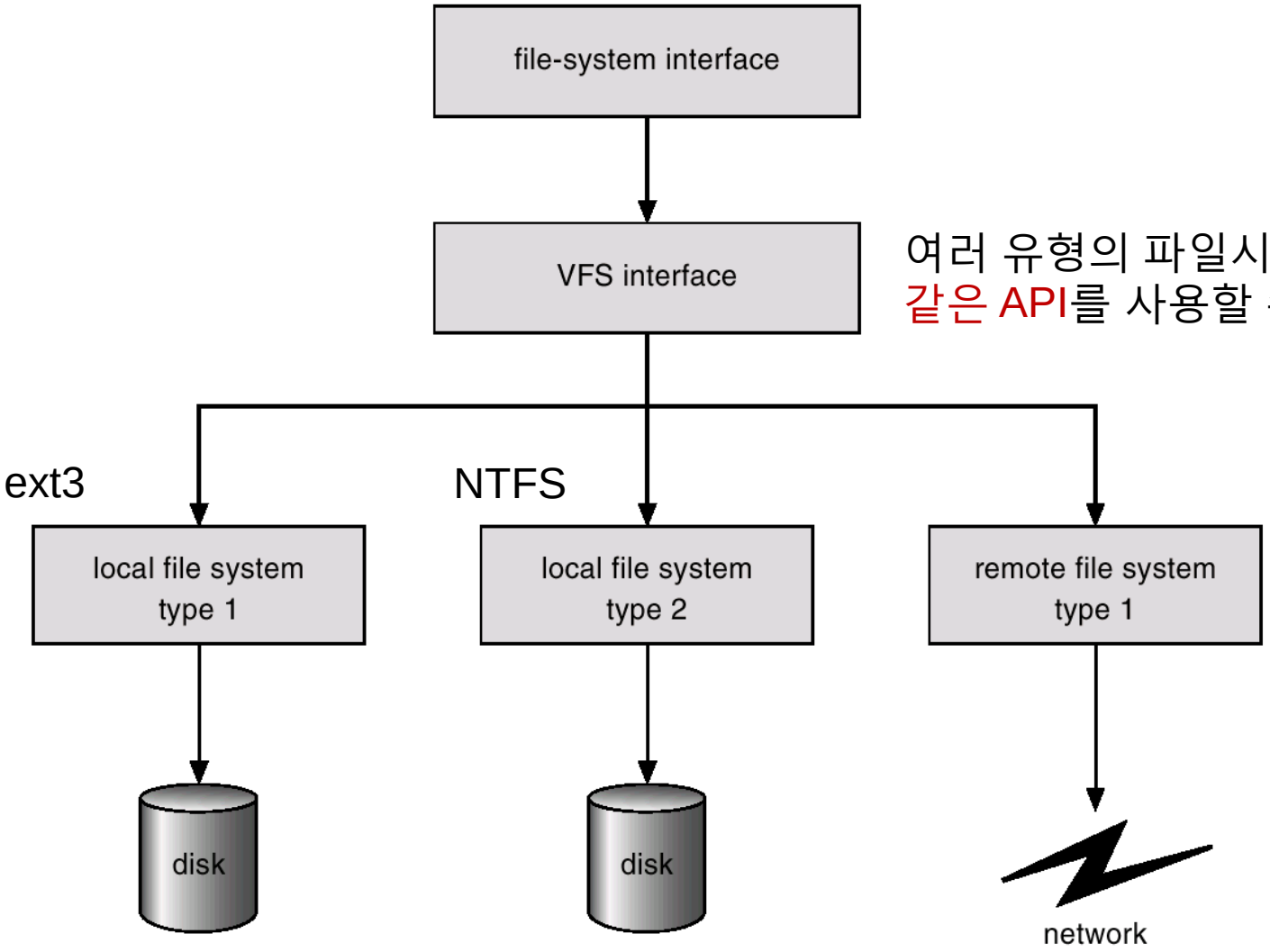
- File control block (FCB)
  - 파일에 정보로 구성되는 storage structure
- 전형적인 file control block



Partitions과 Mounting

- partition은 “raw”(unformatted) 또는 “cooked”(formatted)일 수 있음
  - raw partition : file system이 없음
    - usages: UNIX swap space, some databases, boot information
      RAID system information
  - cooked partition : a file system을 포함
- Mounting
  - root partition이 부트 시간에 마운트됨
  - 다른 partition들은 부트 시간에 자동적으로 마운트되거나
    나중에 수동으로 마운트될 수 있음
    - Windows는 각 volume을 분리된 이름 공간에 마운트 (예) C:, D: ..
    - UNIX는 partition의 파일시스템을 임의의 디렉토리에 마운트 가능
      → 전체적으로 하나의 디렉토리 구조 유지
  - 마운팅 정보는 in memory **mount table**에 저장됨
    - 각 파티션에 있는 파일시스템의 superblock에 대한 포인터

Virtual File Systems



여러 유형의 파일시스템에 대해서
같은 API를 사용할 수 있도록 함

11.3 Directory 구현

- Linear List
  - 디렉터리를 (파일이름, 포인터)들의 linear list로 구현
  - 구현하기 쉽지만 디렉터리 검색에 시간이 소요됨
- Hash Table
  - 파일이름에 대한 hash 함수 값을 계산하여 linear list의 위치를 결정
  - 디렉터리 검색 시간 감소
  - 여러 파일들이 같은 위치로 hash되는 충돌 상황에 대한 처리가 필요

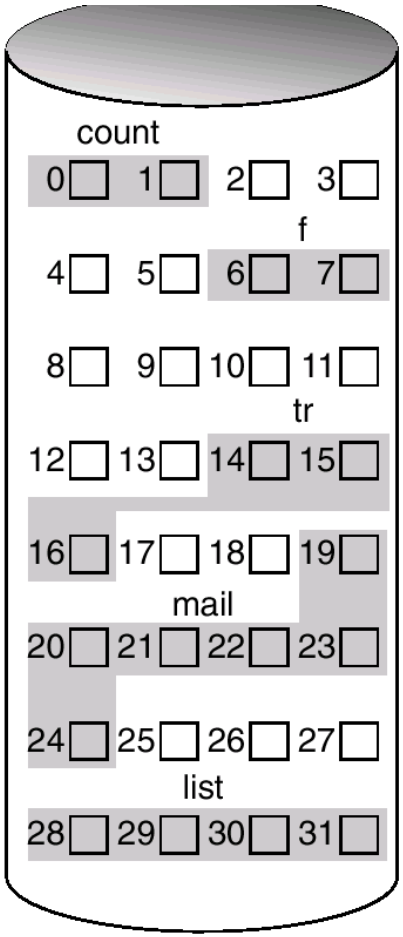
11.4 Allocation 방법

- 파일에 대한 디스크 공간 할당 방법의 고려 사항
  - 디스크 공간의 효율적 이용
  - 파일의 빠른 접근

- 3가지 주요 할당 방법
  - contiguous allocation
  - linked allocation
  - indexed allocation

연속 할당(Contiguous Allocation)

- 각 파일은 디스크의 연속된 블록들의 집합을 점유



directory

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

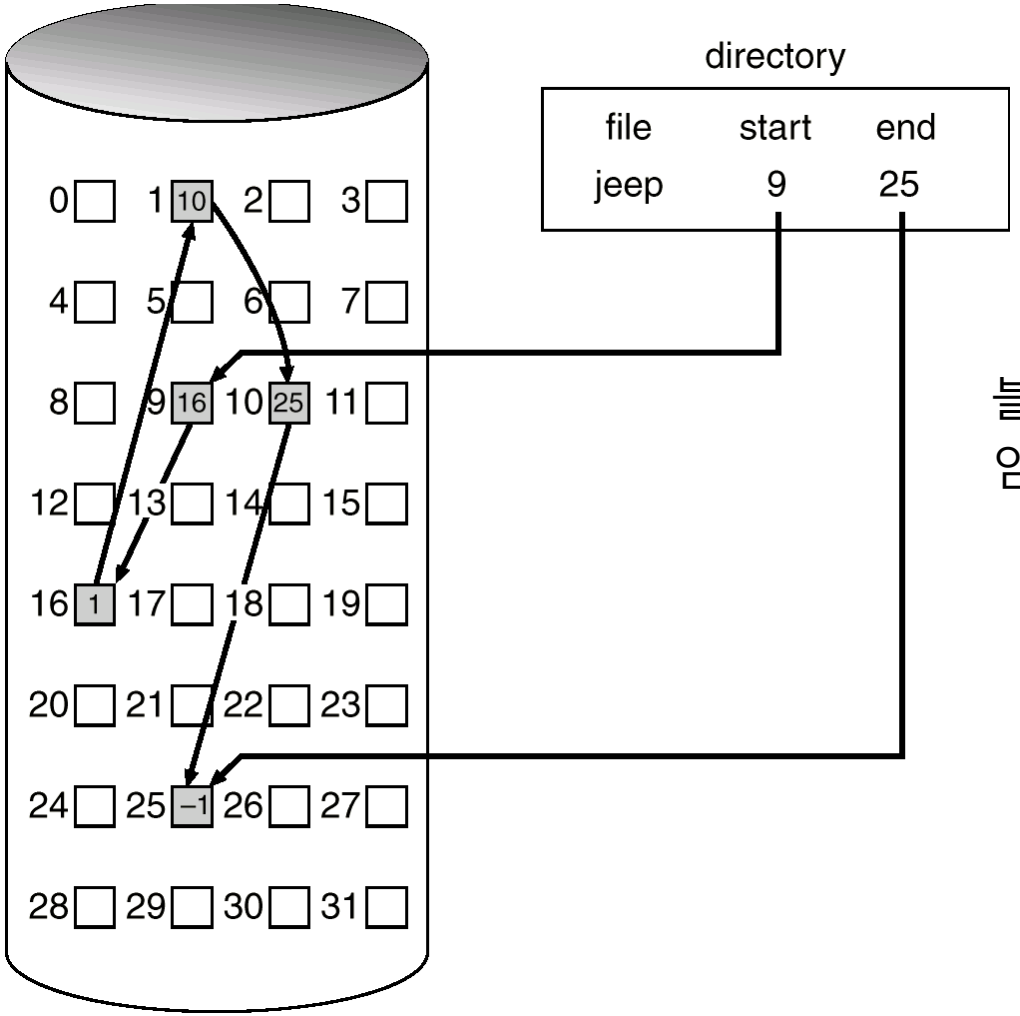
if start block = b,
block **i** of a file → access block **b+i**

Contiguous Allocation (계속)

- 장점
  - 최소 탐색 시간(seek time)
  - 간단함 – 디렉토리는 시작위치와 크기만 필요로 함
  - Random access : 파일 접근이 쉬움
    - 파일의 블록 i의 위치 = b + i (b는 파일의 start 디스크 블록번호)
- 단점
  - 외부 단편화로 인한 저장공간 낭비 – 동적 저장공간 할당 문제
  - 파일 크기를 증가시킬 수 없음
    - 해결책 : (1) pre-allocation ➔ but, internal fragmentation
      (2) 새로운 큰 연속적인 공간을 찾아서 저장 ➔ 속도 저하

연결 할당(Linked Allocation)

- 각 파일은 디스크 블록들의 linked list 로 구성



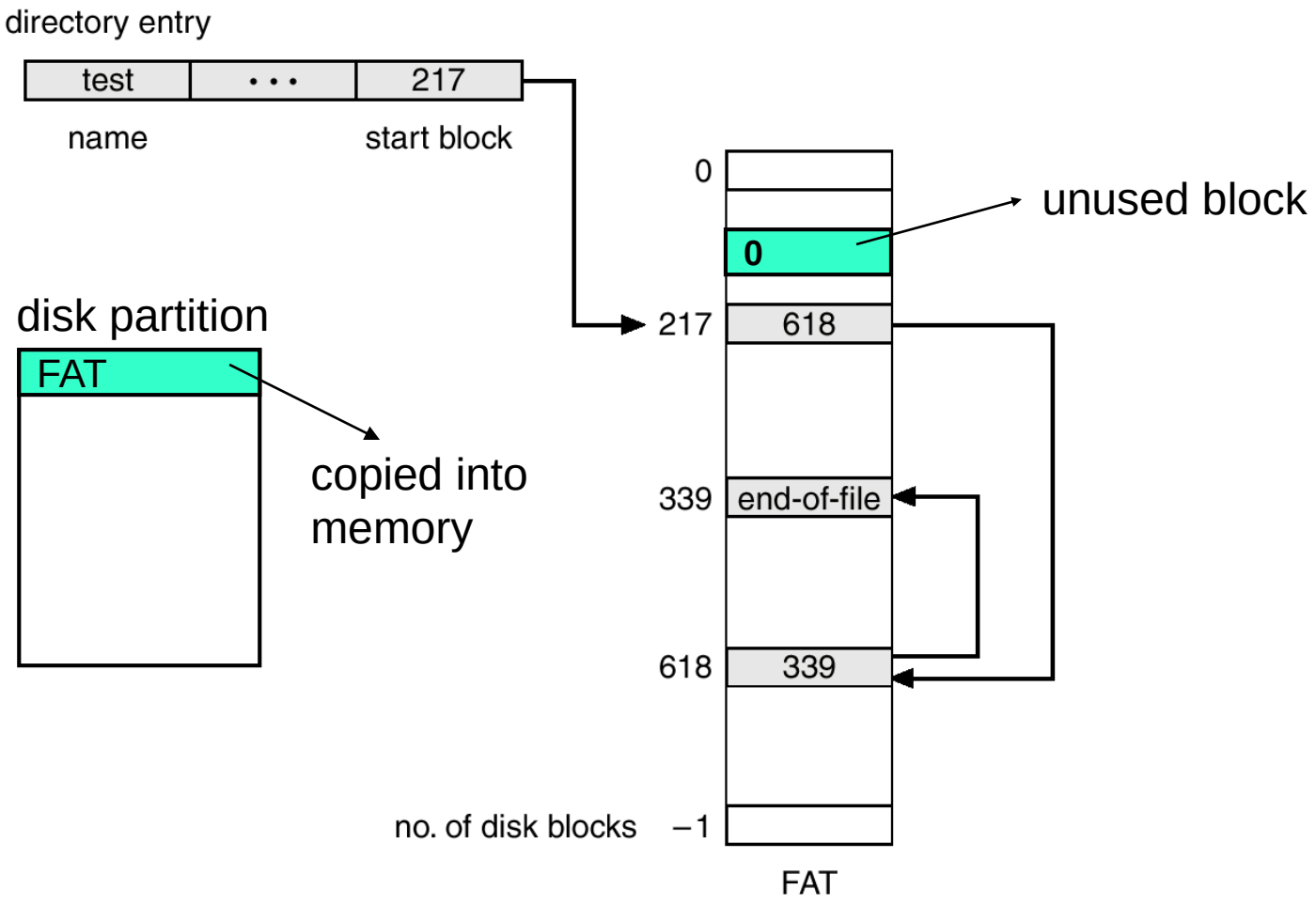
block = pointer

블록들이 디스크의
임의의 위치에 분산 가능

Linked Allocation (계속)

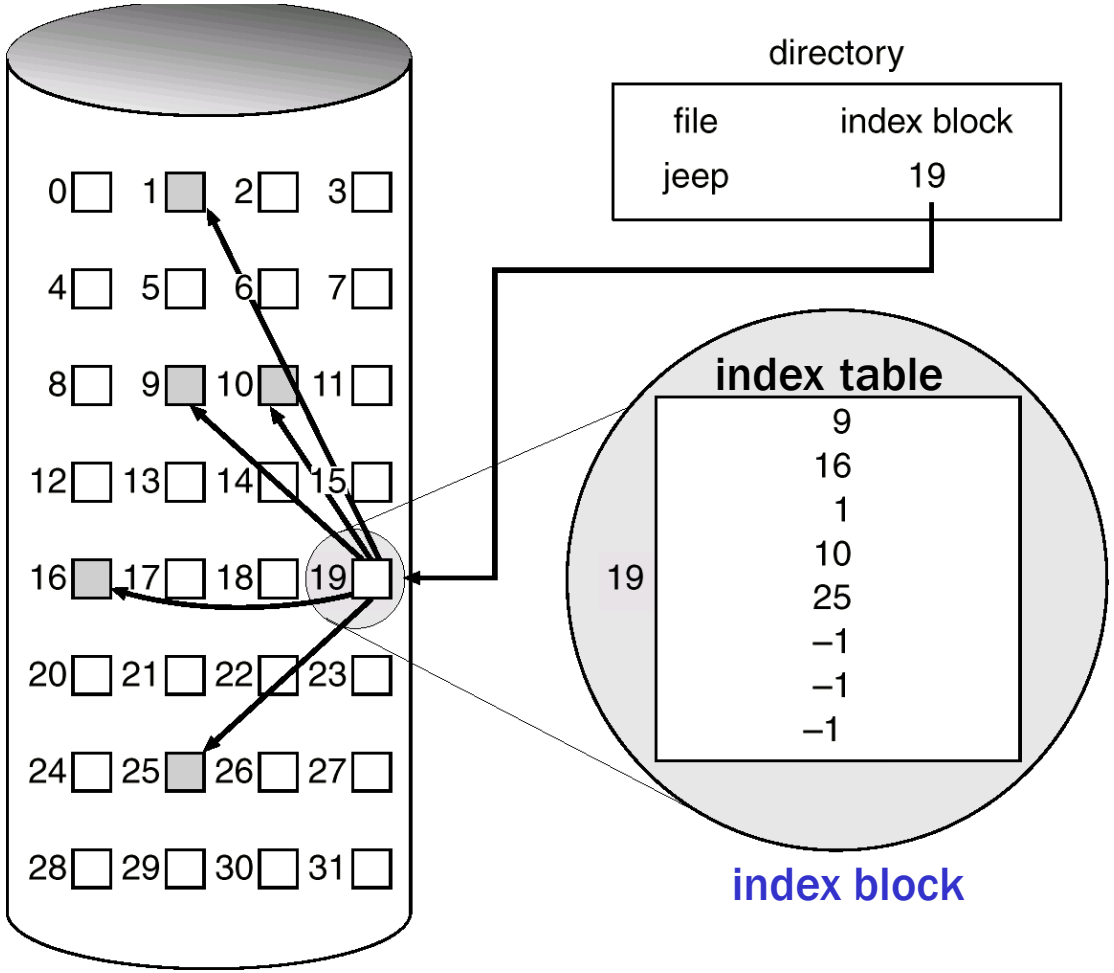
- 장점
  - 간단함 – 디렉토리는 시작위치(와 마지막 위치)만 필요로 함
  - 공간 낭비가 없음 – 연속공간 불필요, 외부 단편화 없음
- 단점
  - random access 불가능 – 임의 위치 접근이 비효율적
    - 뒤에 있는 블록은 여러 번의 디스크 접근 필요
  - 연결 포인터를 위한 공간이 필요
  - 해결책: 여러 개의 연속 블록인 clusters 단위로 할당
    (디스크 throughput 향상, 내부 단편화 발생)
- File-allocation table (FAT) 방식
  - MS-DOS에서 사용하는 디스크 공간 할당 방식
  - linked allocation의 변형
    - FAT는 파티션의 시작에 위치함
    - FAT는 각 디스크 블록에 대해 한 entry를 제공
      → 각 디스크 블록의 next block number를 저장
  - FAT는 메모리에 캐쉬되어 사용 ➔ 효율적 파일 접근

File Allocation Table (FAT)



인덱스 할당(Indexed Allocation)

- 인덱스 블록 사용 – 디스크 블록의 pointer을 저장하는 디스크 블록

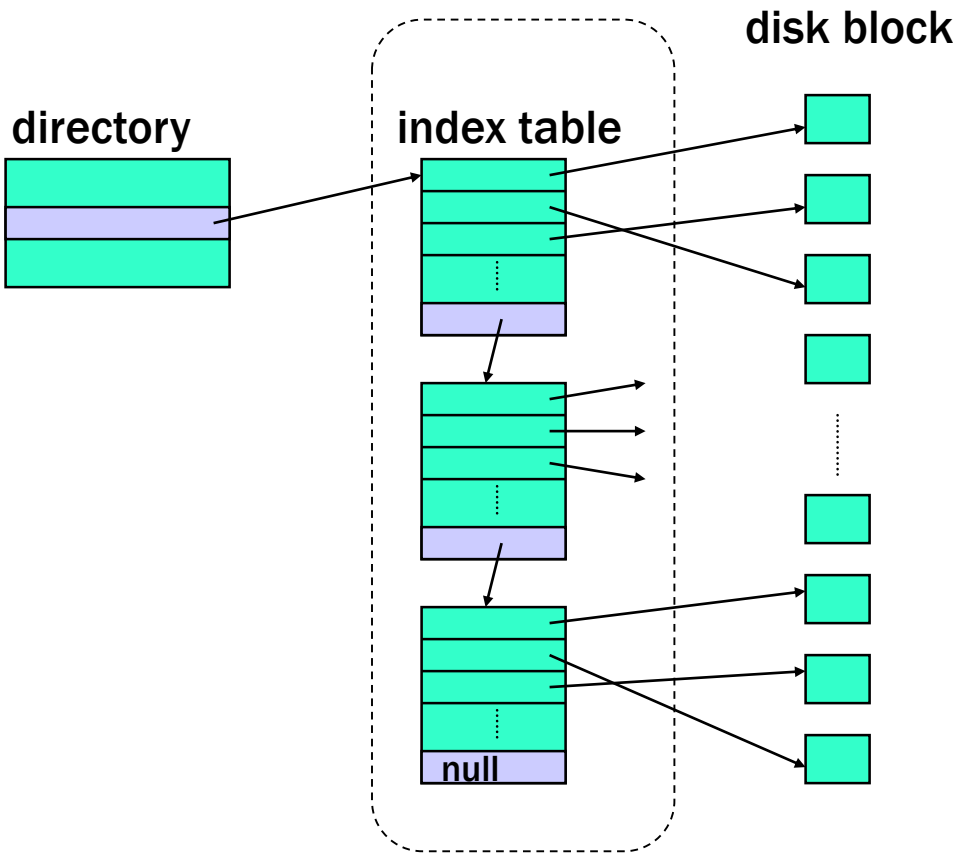


Indexed Allocation (계속)

- 장점
  - random access 가능 (contiguous allocation의 장점)
  - 외부 단편화 없음 (linked allocation의 장점)
  - 디렉토리는 인덱스 블록 위치만 저장
- 단점
  - index block이 필요함
    - linked allocation보다 pointer overhead가 더 큼
  - 1개의 index block → 파일 크기에 제한
    - (예) 512 word/block : 1 block은 512 block의 포인터 저장
      → 최대 파일크기 512 x 512 = 256K word
- 크기 제한의 해결책 - 여러 개의 index block을 사용
  - linked scheme
  - multilevel index
  - combined scheme

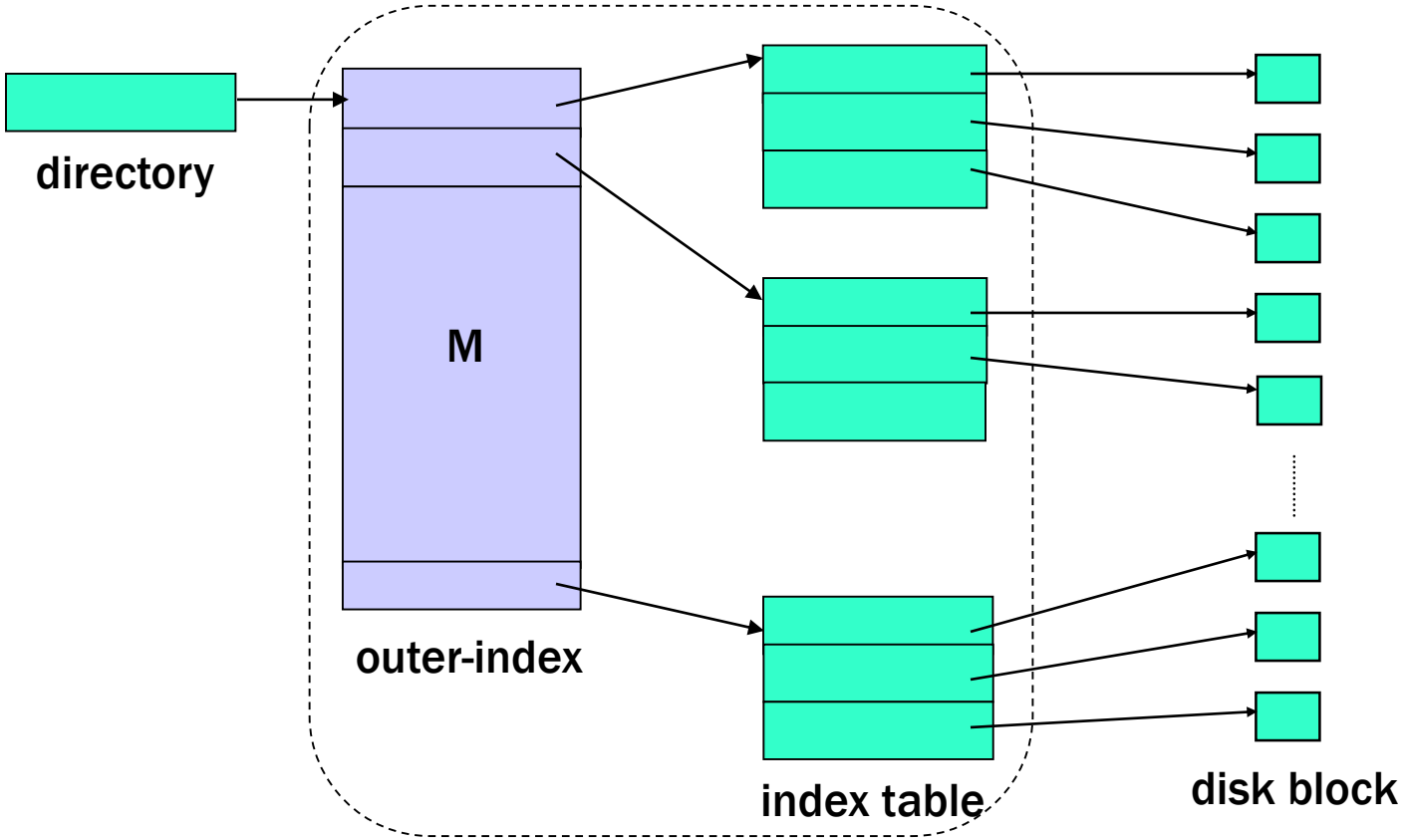
Indexed Allocation - Linked scheme

- Link blocks of index table – 파일 크기 제한 없음



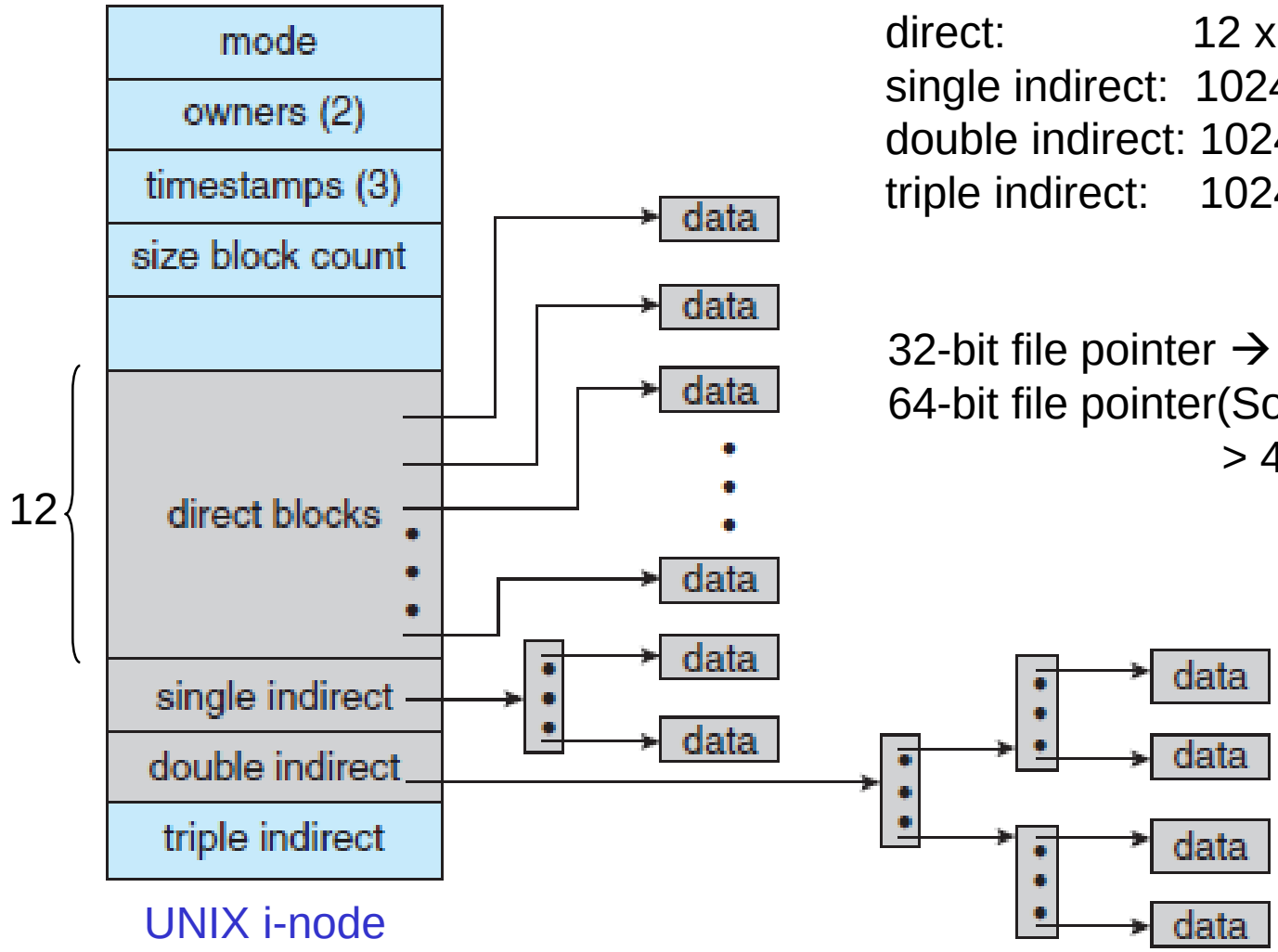
Indexed Allocation - Multilevel index

- Two level index



4KB/block, 1024 4byte pointers ➔ max file size: 1024^2 x 4KB = 4GB

Combined Scheme: UNIX (4KB / block)



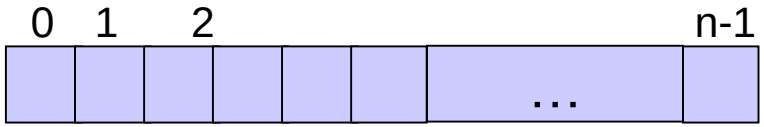
UNIX i-node

direct: 12 x 4KB = 48KB
single indirect: 1024 x 4KB = 4MB
double indirect: 1024^2 x 4KB = 4GB
triple indirect: 1024^3 x 4KB = 4TB

32-bit file pointer → 4GB
64-bit file pointer(Solaris, AIX) > 4GB

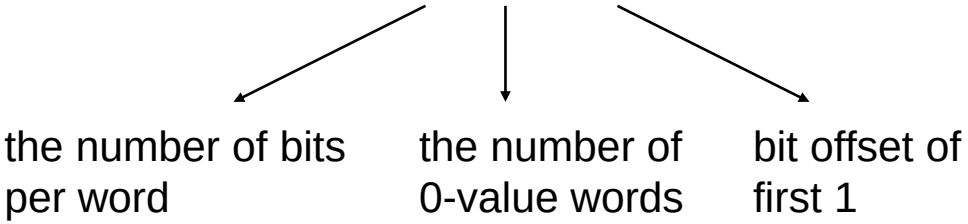
11.5 Free-Space 관리

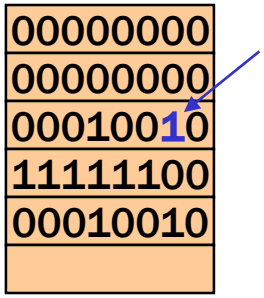
- Bit vector
  - free space list를 **bit map** 또는 **bit vector**로 구현



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

  - block number 계산
    - block number = 8 x 2 + 1 = 17





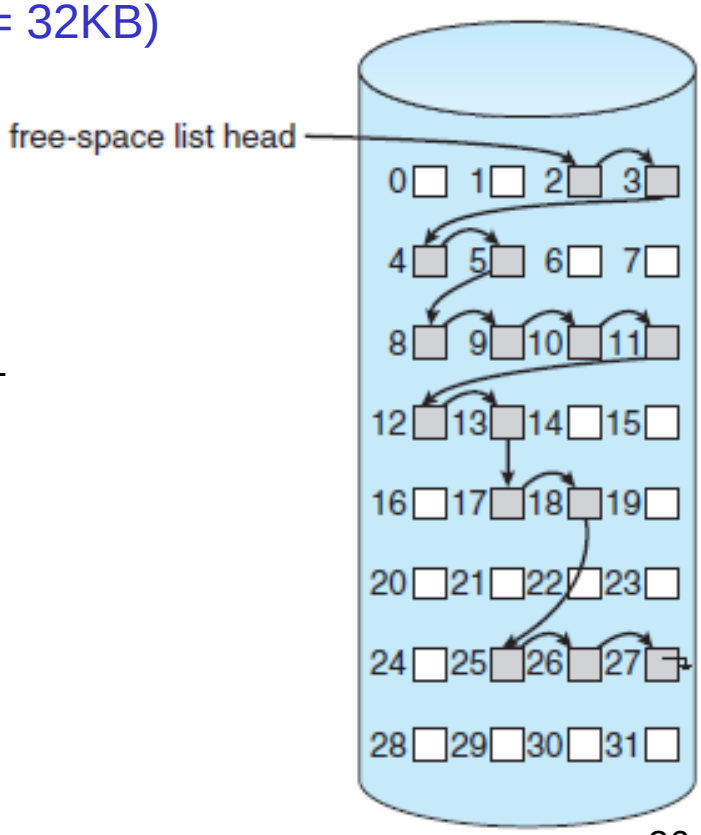
Free-Space 관리(계속)

- Bit vector 방식 (계속)
  - 장점 - 비교적 간단, 첫째 free block 및 n개의 연속 free block 찾기 쉬움
  - 단점 - Bit map은 추가 공간을 필요로 함

    block size = 2^{12} B (4KB), disk size = 2^{30} B (1GB)인 경우

    $n = 2^{30}/2^{12} = 2^{18}$ bits (or 2^{15} B = 32KB)

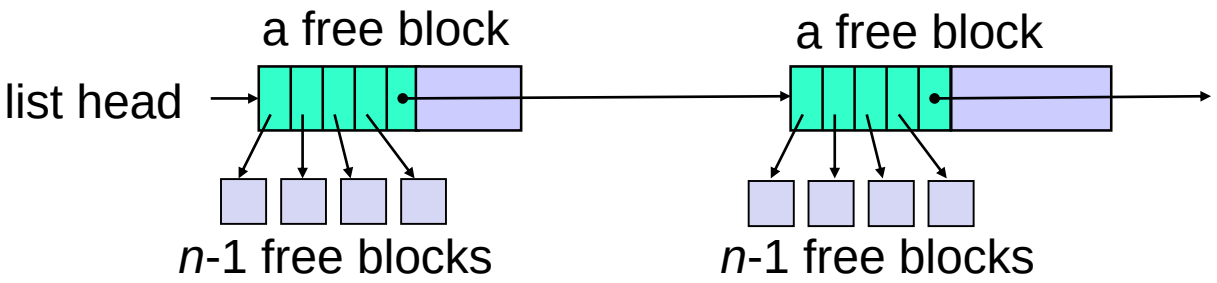
    - 전체 bit vector를 메모리에 적재할 수 없으면 비효율적



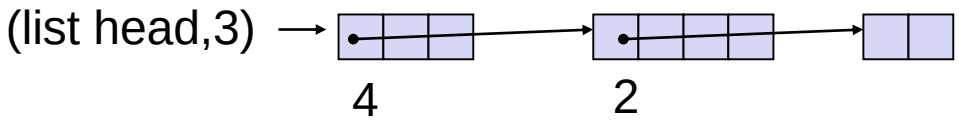
- Linked list (free-list)
  - 단점 - 연속공간을 쉽게 얻을 수 없음
  - 장점 – 공간 낭비가 없음

Free Space 관리

- Grouping – free list 방식의 수정
  - 첫 번째 free block에 n개의 free block의 주소 저장



- Counting
  - free list는 연속된 free block의 첫 번째 주소와 블록 개수를 저장
  - free list 길이가 짧아짐



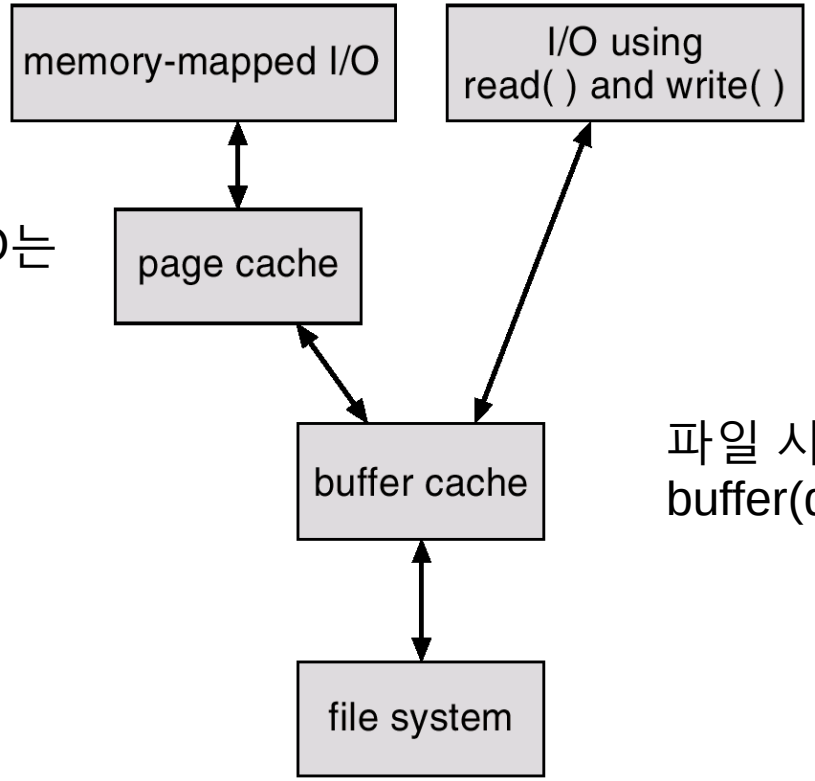
11.6 효율(Efficiency)과 성능(Performance)

- 효율
  - **디스크 할당** 및 **디렉토리 알고리즘**이 디스크 공간의 효율적 사용에 영향을 줌.
  - 디렉토리 entry에 저장되는 데이터 종류를 고려해야 함
    - 마지막 접근 시간을 기록한다면 비효율적임
- 성능
  - disk cache
    - 빈번히 사용되는 disk block을 저장하는 메모리의 부분
    - buffer cache, page cache (가상메모리와 결합하여 사용)
  - free-behind 및 read-ahead
    - 순차 접근을 최적화하기 위한 기법

Page Cache

- page cache
  - 파일 데이터를 파일 블록이 아닌 가상 메모리의 page 단위로 캐쉬
  - double caching이 발생함 – 메모리 공간 낭비

Memory-mapped I/O는
page cache를 사용

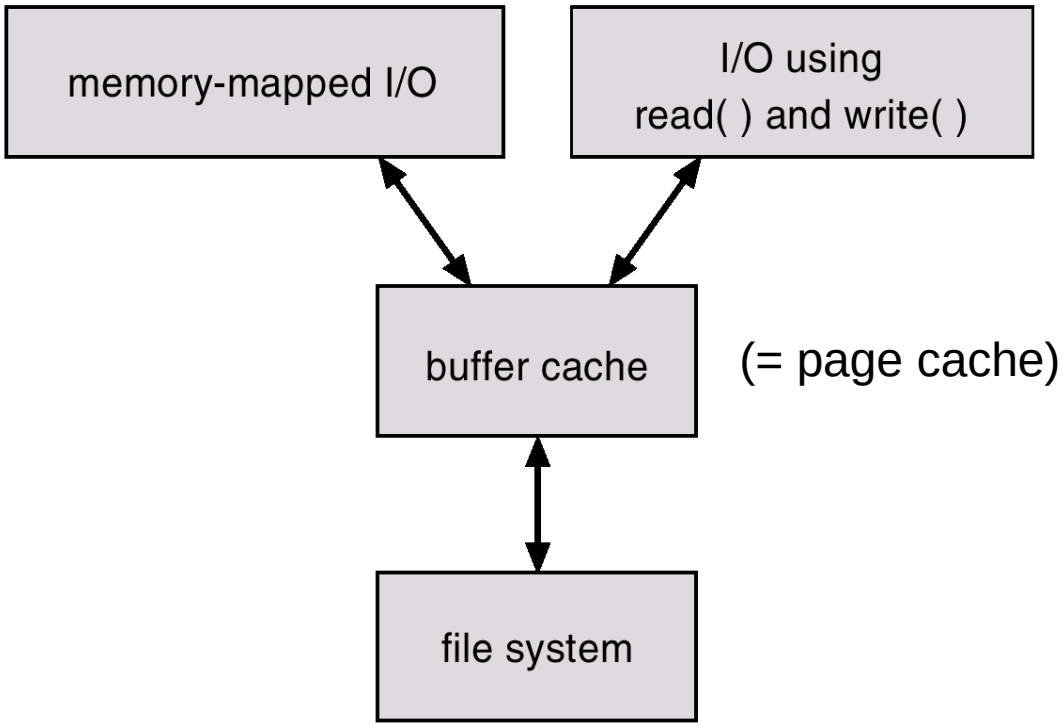


파일 시스템 I/O 루틴은
buffer(disk) cache를 사용

I/O Without a Unified Buffer Cache

Unified Buffer Cache

- A unified buffer cache
  - memory-mapped pages와 보통 file system I/O를 복사하는 데 모두 같은 page cache를 사용 → double caching 발생하지 않음



11.7 Recovery

- 파일과 디렉토리는 주메모리와 디스크에 모두 저장됨
  - ➔ system failure는 데이터 손실 및 데이터 비일관성(inconsistency)을 가져올 수 있음
- 일관성 검사(Consistency checker)
  - 디렉터리 구조에 있는 데이터와 디스크의 데이터 블록을 비교하여 inconsistency를 찾고, inconsistency를 고침 → 항상 가능하지는 않음
- Backup
  - 시스템 프로그램을 사용하여 디스크 데이터를 다른 보조저장장치(테이프 등)로 백업
  - full backup → incremental backup → . . . → incremental backup

- Restore
  - 백업 장치로부터 데이터를 복원하여 손실된 파일/디스크를 복구함

로그 구조(Log Structured) 파일 시스템

- 로그 구조 파일 시스템 (Journaling 파일시스템)
  - 파일 시스템에 대한 각 update를 transaction으로 기록
- 모든 트랜잭션은 로그에 기록됨
  - 트랜잭션이 로그에 기록되면, 트랜잭션은 commit(확정)된 것으로 간주
  - 이 때에, 파일시스템은 아직 갱신되지 않을 수 있음
- 로그에 있는 트랜잭션은 비동기적으로 파일시스템에 기록됨
  - 파일 시스템이 수정되면, 해당 트랜잭션은 로그에서 삭제됨
- 파일시스템에 기록되는 동안 시스템이 고장 나면, 로그에 있는 모든 남아있는 트랜잭션은 다시 수행될 수 있음
- 로그에 기록되는 동안 시스템이 고장 나면, 해당 트랜잭션은 실행되지 않음 – 파일시스템 일관성은 유지됨