

Chapter

11

파일 시스템 관리

1. 파일 시스템과 저장 장치
2. 파일 시스템의 논리 구조
3. 파일 시스템 구축
4. 파일 입출력 연산



교재 11장 편집 오류 정정 알림

2

□ p. 552 하단부, stdio.c 프로그램의 실행 결과 수정 필요

```
$ gcc -o stdio stdio.c  
$ ./stdio  
a  
ahello  
$
```



수정

```
$ gcc -o stdio stdio.c  
$ ./stdio  
a  
hello  
a  
$
```

- stdout(표준출력장치)에 출력되는 내용은 출력 버퍼에 모았다가 한 번에 출력하지만,
- stderr(표준오류장치)에 출력되는 내용은 오류를 전달하기 위해 출력 버퍼를 거치지 않고 바로 출력되므로,
- fprintf(stderr,...)에 출력되는 내용은 fprintf(stdout, ...)보다 화면에 먼저 출력된다.
- 그러므로 hello 문자열이 문자 a보다 먼저 화면에 출력된다.

강의 목표

1. 응용프로그램과 저장 장치 사이의 파일 입출력 과정을 안다.
2. 디렉터리와 파일의 계층 구조에 대해 이해한다.
3. 파일 메타 정보와 파일 시스템 메타 정보에 대해 이해한다.
4. FAT 파일 시스템의 저장 구조에 대해 이해한다.
5. Unix 파일 시스템의 저장 구조에 대해 이해한다.
6. 파일 입출력 연산이 이루어지는 과정을 이해한다.
 - 파일 찾기, 파일 열기, 파일 읽기, 파일 쓰기, 파일 닫기

4

1. 파일 시스템과 저장 장치

파일과 저장 장치

5

□ 파일

- ▣ 사용자나 응용프로그램 관점
 - 정보를 저장하고 관리하는 논리적인 단위
- ▣ 컴퓨터 시스템의 관점에서
 - 정보를 저장하는 컨테이너
 - 파일은 0과 1의 데이터 덩어리
 - 영구 저장 장치나 일시 저장 장치에 저장
 - 디스크 장치, USB 장치, SSD(Solid-State Drive), 테이프 저장 장치
 - 램 디스크(RAM Disk)

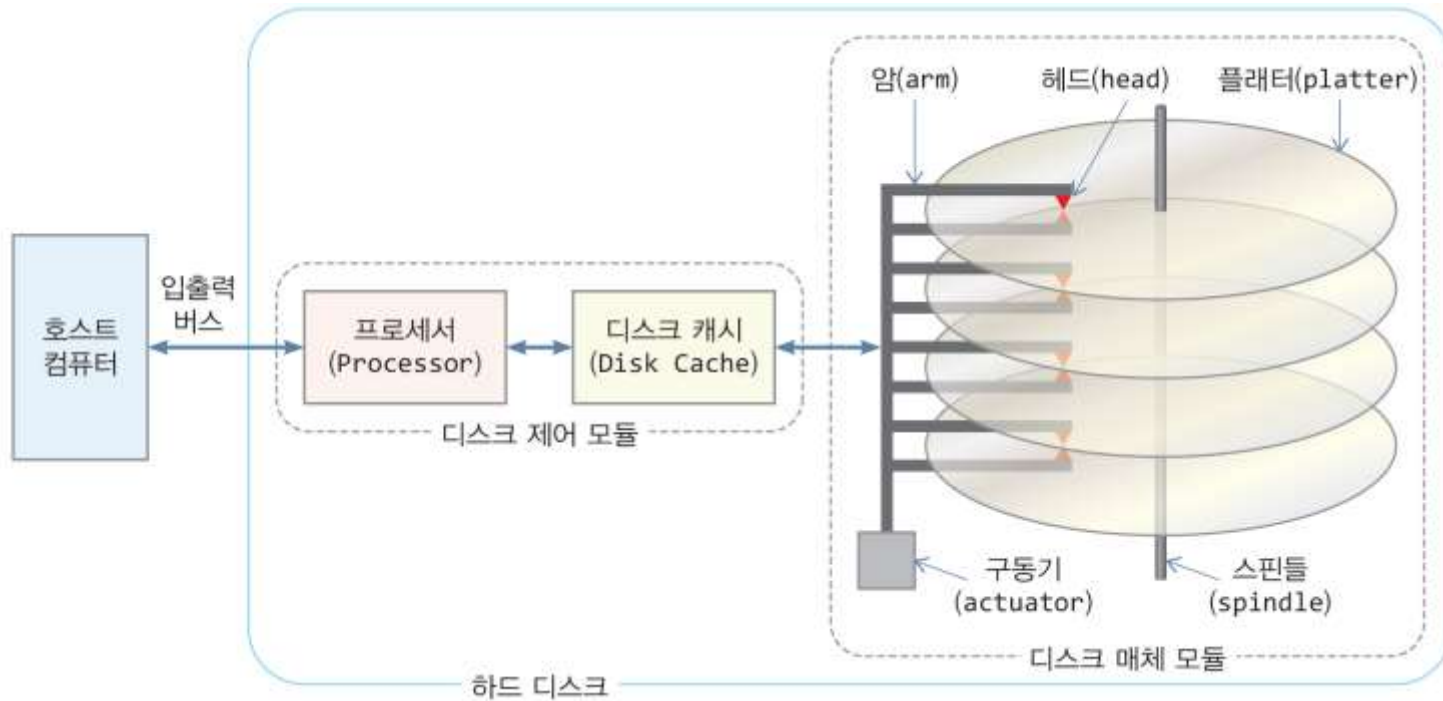
□ 운영체제

- ▣ 파일 생성, 기록, 읽기 모든 과정 통제
- ▣ 응용프로그램은 운영체제 모르게 파일 다루기 불가능
 - 저장 매체, 빈 공간 등의 관리는 모두 운영체제가 수행

하드 디스크 장치의 구조

6

1 플래터 2 헤드



디스크 장치 개요

7

□ 디스크 매체 모듈

▣ 플래터(platter)

- 정보가 저장되는 매체, 원형 판(아래 윗면 모두 저장)

▣ 헤드(head)

- 플래터 한 면당 하나의 헤드(플래터 한 장에 2개의 헤더)
- 플래터에서 정보를 읽고 저장하는 장치

□ 디스크 제어 모듈

▣ 프로세서(processor)

- 호스트로부터 명령을 받고 해석하는 하드웨어 처리기
- 디스크 매체 모듈 제어, 물리적인 디스크 액세스 진행

▣ 디스크 캐시(disk cache)

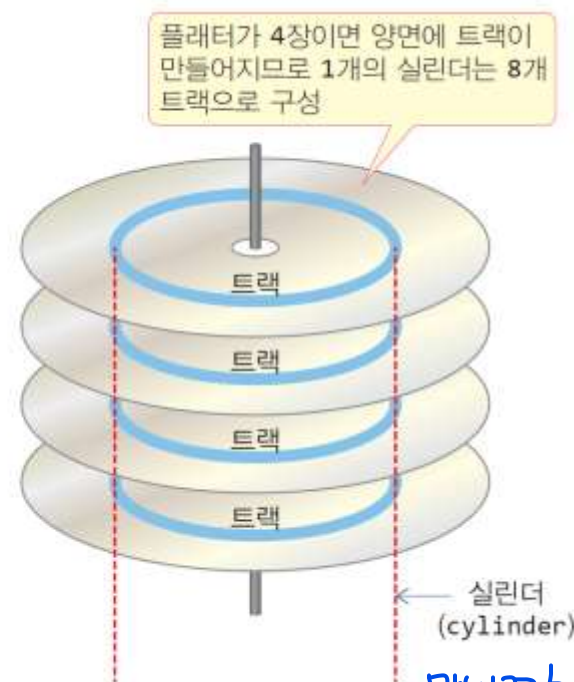
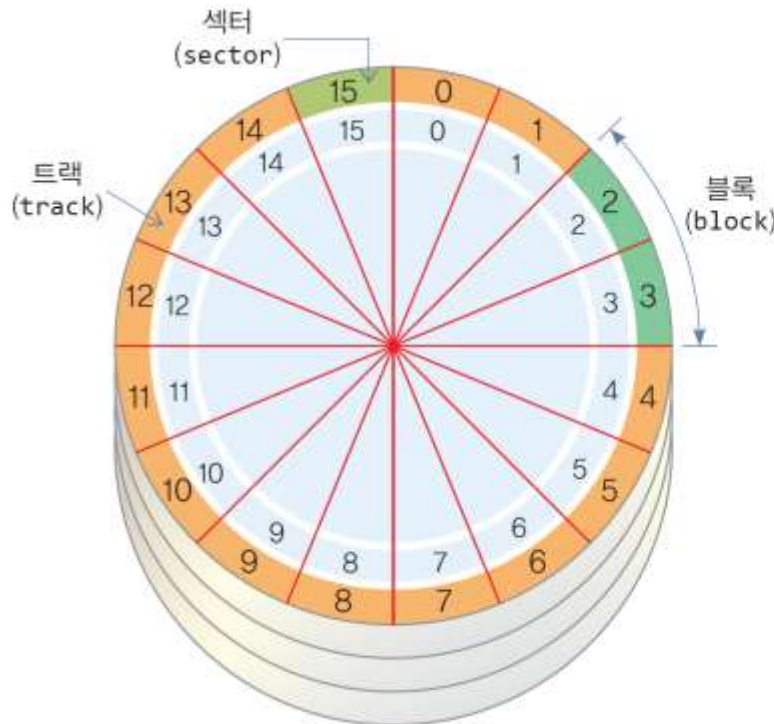
- 1MB~몇 십 MB 크기의 빠른 반도체 메모리
- 호스트와 디스크 매체 모듈 사이의 전송되는 디스크 블록들이 임시 저장되는 중간 버퍼 역할

섹터, 트랙, 실린더, 블록

8

현재

- ❑ **섹터** : 플래터에 정보가 저장되는 최소 단위, 512바이트 혹은 4096바이트
- ❑ **트랙** : 플래터에 정보가 저장되는 하나의 동심원, 여러 개의 섹터들 포함
- ❑ **실린더** : 같은 반지름을 가진 모든 트랙 집합
 - 예) 헤드가 8개인 디스크에서 8개의 트랙을 묶어 실린더라고 함
- ❑ **블록** : 운영체제가 파일 데이터를 입출력하는 논리적인 단위. 몇 개의 섹터로 구성



맨바깥: 0번 실린더

파일 입출력 주소

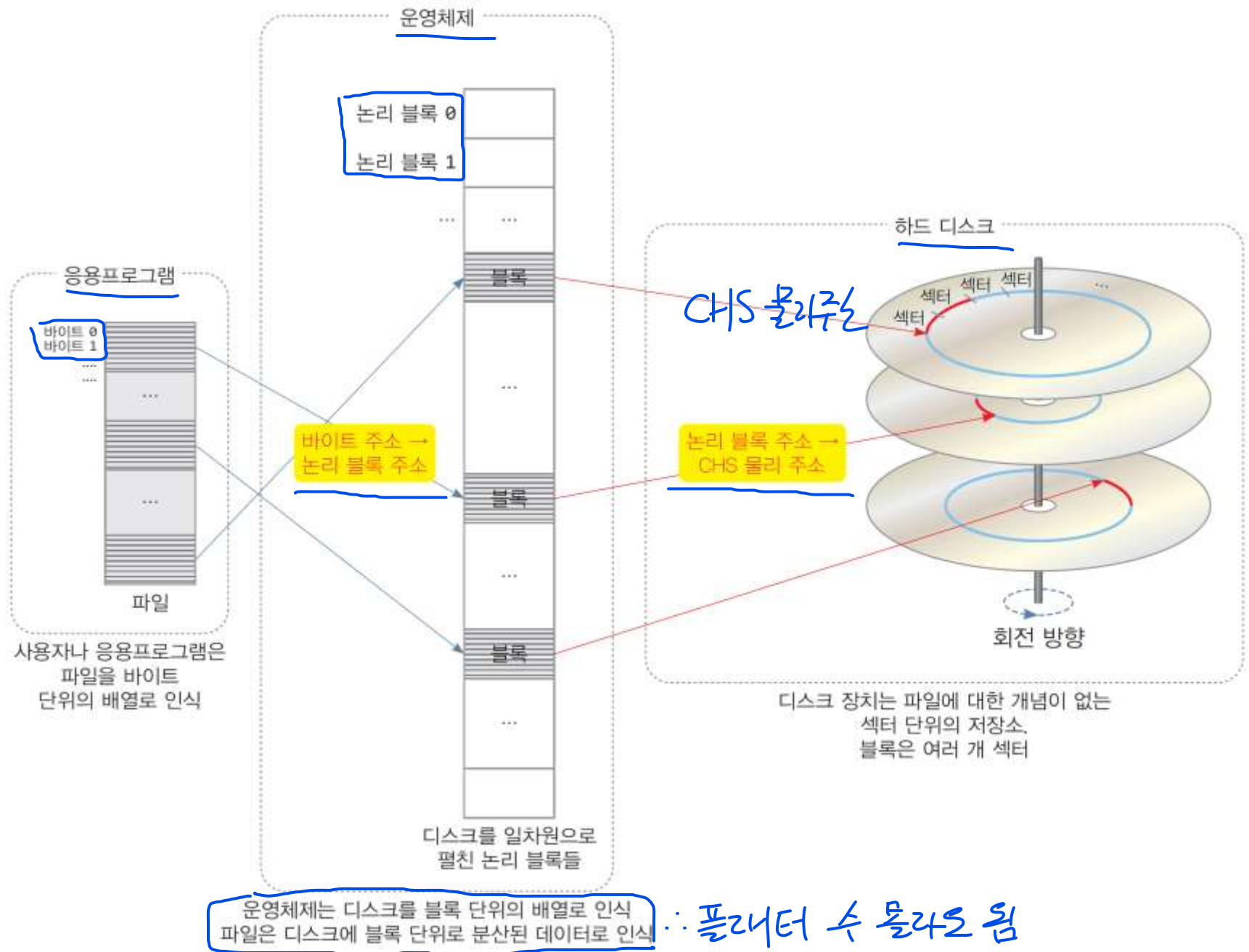
9

- 디스크 장치는 디스크 물리 주소 사용
 - ▣ 물리 주소(physical disk address)
 - 디스크의 섹터 위치를 나타내는 주소
 - CHS(Cylinder-head-sector) 물리 주소 = (cylinder 번호 , head 번호, sector 번호)
 - ▣ 물리 주소의 단위는 섹터

- 운영체제는 논리 블록 주소 사용
 - ▣ 논리 주소(Logical Block Address, LBA)
 - 저장 매체를 1차원의 연속된 데이터 블록들로 봄
 - 저장 매체의 종류에 관계 없음
 - 모든 블록들을 0번부터 시작하는 블록 번호 매김
 - 디스크의 경우 맨 바깥쪽 실린더에서 안쪽으로, 맨 위의 트랙에서 아래로

- 응용프로그램은 파일 내 바이트 주소 사용
 - ▣ 바이트 주소란 파일 내 바이트 위치(옵셋)

응용프로그램, 운영체제, 디스크 장치 사이의 계층화와 주소 변환



파일 주소 변환

11

□ 사용자나 응용프로그램

- ▣ 파일 데이터가 바이트 단위로 연속하여 저장된다고 생각

□ 운영체제

- ▣ 파일을 블록 크기로 분할하고 각 블록을 디스크에 분산 저장
- ▣ 블록은 운영체제가 입출력하는 단위

□ 파일 주소 변환

파일 내 바이트 주소 -> 논리 블록 주소 -> CHS 물리 주소

- ▣ 운영체제는 파일 내 바이트 주소를 논리 블록 주소로 변환
- ▣ 디스크 장치의 펌웨어가 논리 블록 주소를 CHS 물리 주소로 변환

주소 계층화 의미

12

▣ 각 계층의 독립적 구현 용이

- 사용자나 응용프로그램, 운영체제, 그리고 디스크 장치는 각각 독립적으로 정의된 기능 수행

▣ 응용프로그램 개발

- 파일을 바이트 단위로 보고 입출력하는 코드 작성
- 운영체제의 종류나 특징에 관계없이 파일 입출력 응용프로그램 개발
- 저장 매체의 종류나 특징, 저장 위치에 무관하게 작성 가능

▣ 운영체제 개발

- 저장 매체를 1차원 배열의 논리 블록들로 다루어,
- (논리 블록 번호를 이용하여 디스크 입출력 시행 -> 저장 매체의 종류나 하드웨어 특징에 관계없이 운영체제 구현 가능
- (운영체제는 바이트 주소를 논리 블록 주소로 바꿈 -> 응용프로그램을 장치로부터 독립

▣ 저장 장치(디스크 장치) 개발

- (논리 블록 번호를 CHS 물리 주소로 바꾸어 디스크 입출력 시행 -> 응용 프로그램이나 운영체제의 특성과 무관하게 저장 장치 개발
 - 예) 운영체제는 저장 장치가 디스크인지, SSD인지 알 필요 없음
 - 예) 운영체제는 저장 장치에 몇 개의 실린더가 있는지 헤드가 몇개인지 알 필요 없음

파일 시스템의 정의와 범위

13

□ 파일 시스템 정의

- 저장 매체에 파일을 생성하고 저장하고 읽고 쓰는 운영체제의 기능을 통칭

□ 파일 시스템의 학습 범위

1. 파일 시스템의 논리 구조 - 수십만 개의 파일들을 다루기 위한 계층 구조

- 디렉터리와 파일로 이루어지는 트리 형태의 계층 구조

2. 저장소에 파일 시스템 구축 - 파일을 어디에 어떻게 저장할 것인가?

- 저장 매체에 파일을 저장하는 방법과 위치 구성
- 저장 매체 속의 빈 블록 유지 관리
- 각 파일이 저장된 위치 관리 기능 구현

3. 커널 내 파일 입출력 구현 - 파일을 읽고 쓰는 등의 기능

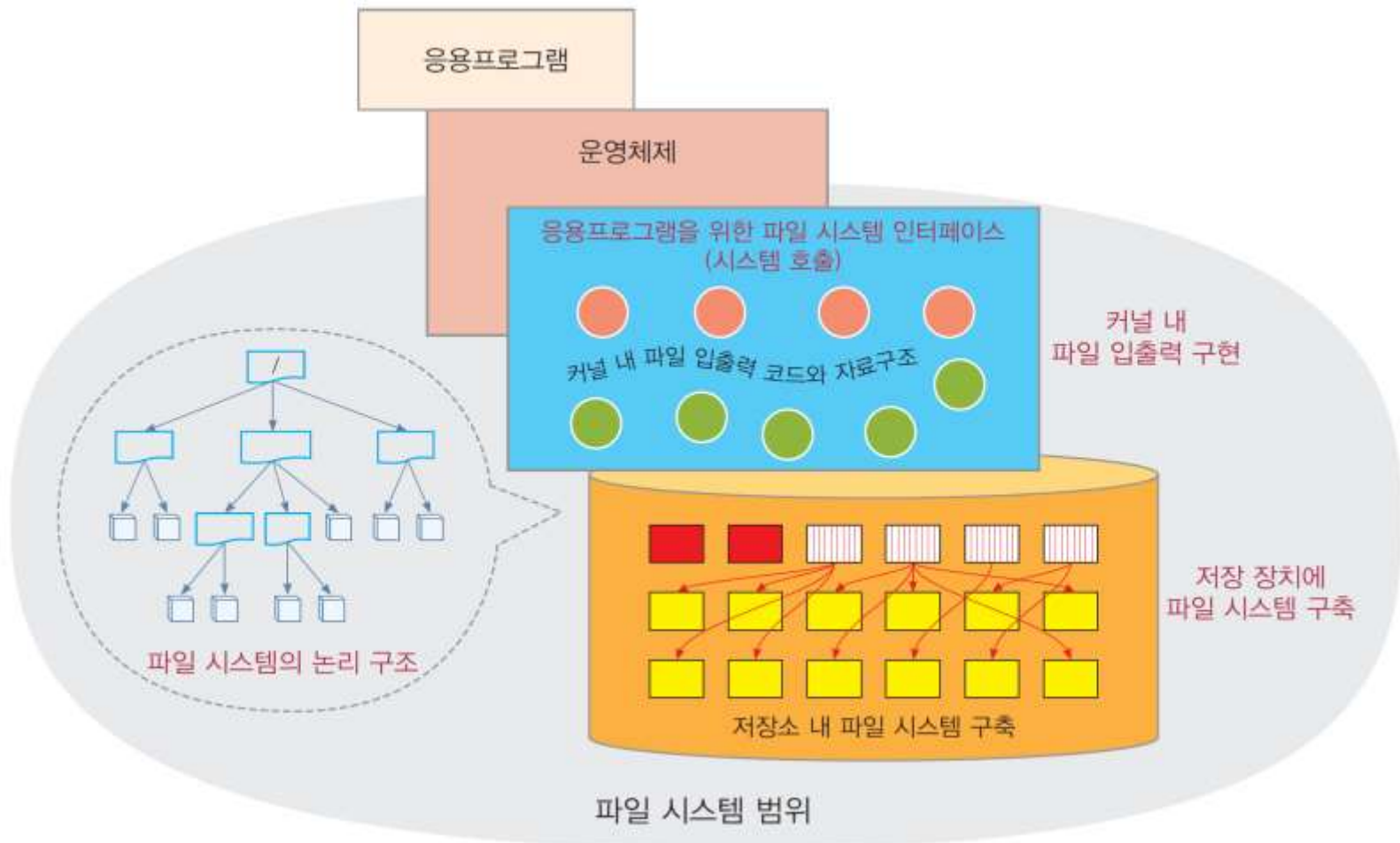
- 파일 생성
- 파일 열기
- 파일 읽기
- 파일 쓰기
- 파일 닫기
- 파일 삭제
- 파일 메타 정보 읽기/변경

4. 응용프로그램을 위한 파일 시스템 인터페이스(시스템 호출)

- 파일 생성, 읽기, 쓰기 등 커널에 구현된 기능을 응용프로그램이 활용할 수 있는 시스템 호출
- open() close(), read(), write(), seek() 등

파일 시스템의 범위

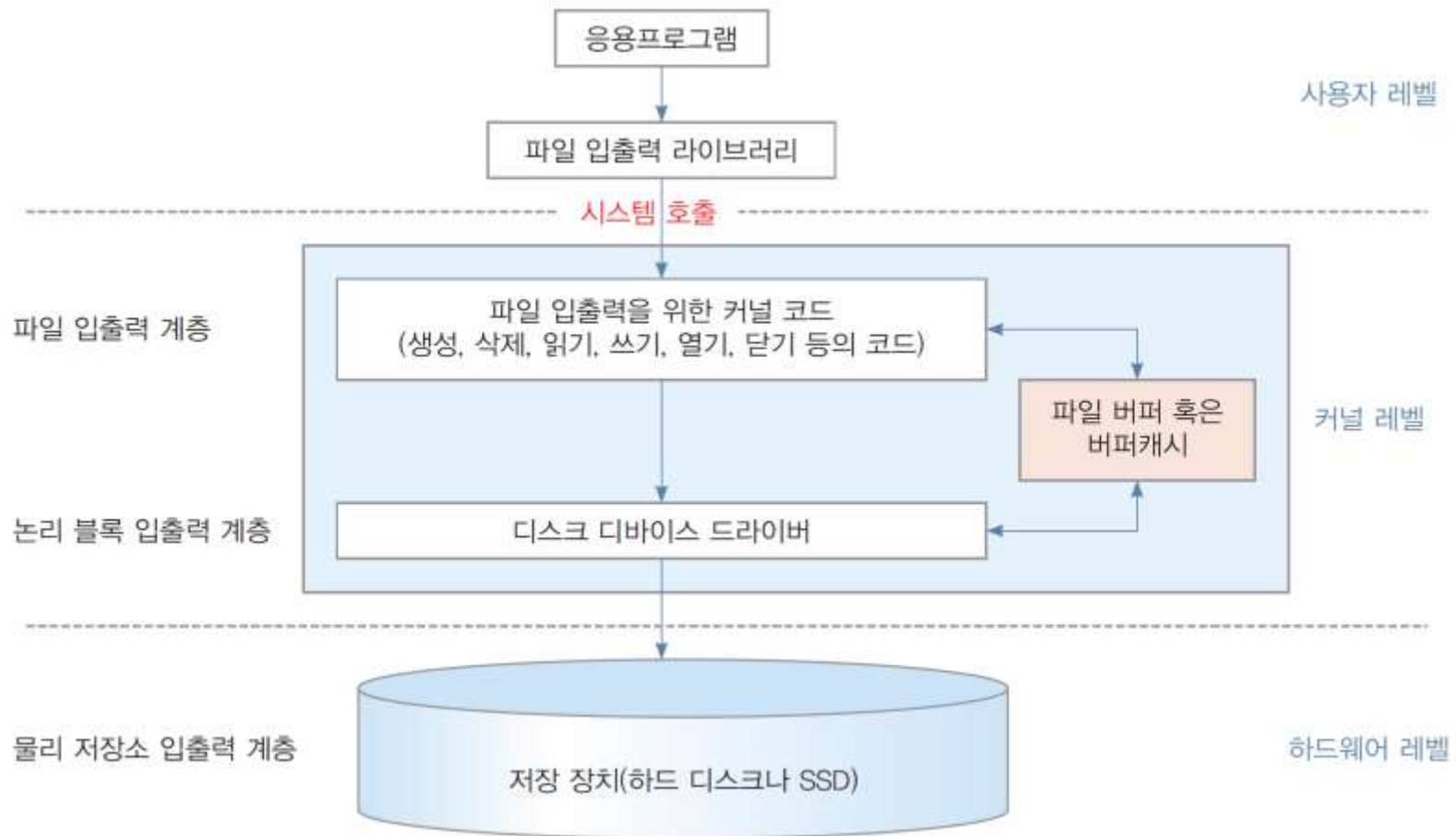
14



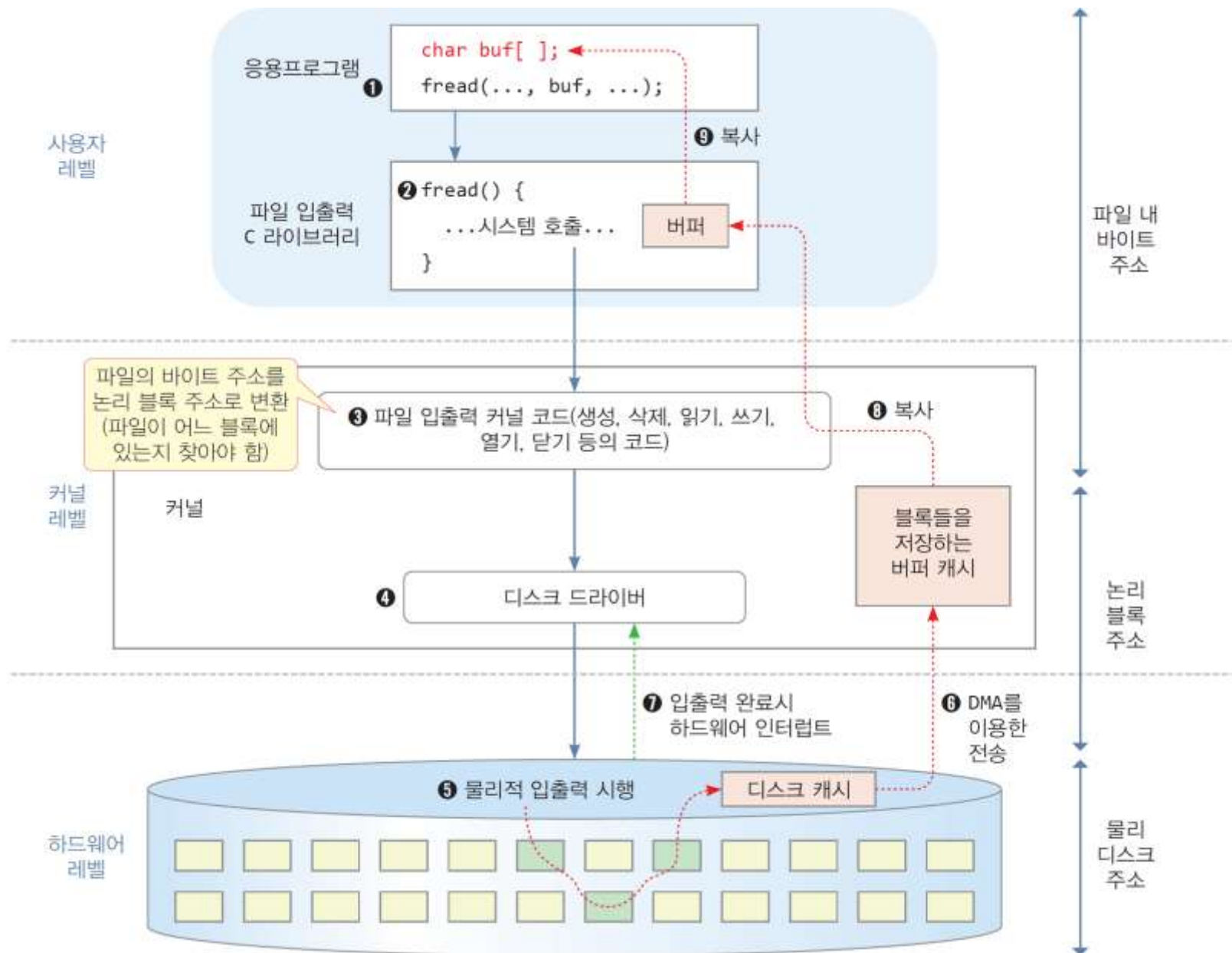
파일 시스템 범위

파일 시스템 입출력 계층

15



파일 읽기를 통한 디스크 입출력 개요



파일 읽기 과정을 통한 주목 사항

17

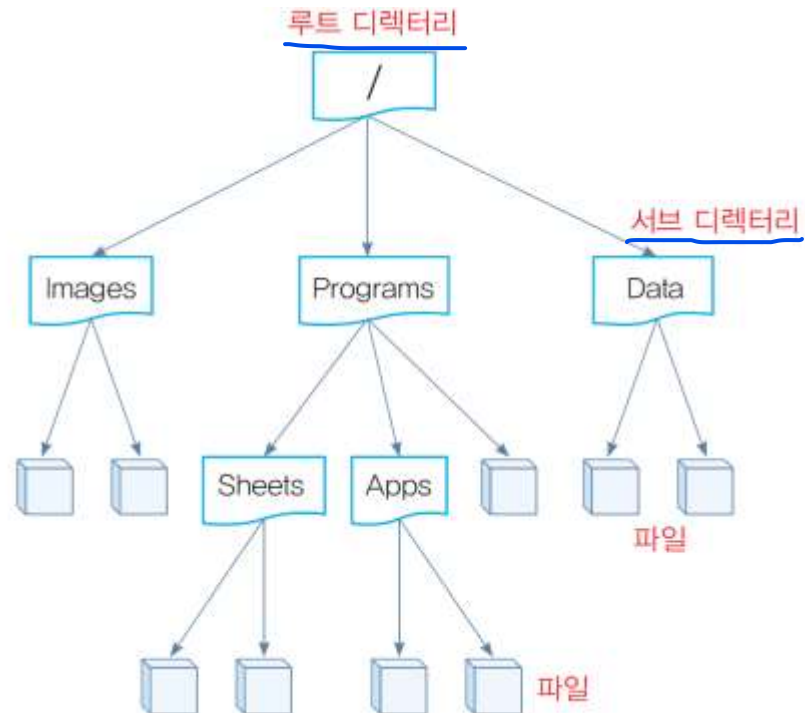
- 파일 읽기에서 각 계층의 역할이 잘 구분됨
- 운영체제는 응용프로그램이 저장 장치의 종류, 구조, 위치 등 물리적인 특성에 무관하게 입출력 지원
- 디스크 디바이스 드라이버가 파일에 대한 논리적 공간과 물리적인 공간 분리
- 파일 데이터는 여러 번의 복사를 거쳐 이동
 - 디스크 플래터->디스크 캐시->커널의 버퍼 캐시->라이브러리의 버퍼 -> 응용프로그램 버퍼
 - 여러 번의 복사로 인해 많은 시간 소요되기도 하지만,
 - 동일한 파일이 여러 번 액세스되는 경우 효과적
 - 시스템 호출 없이, 라이브러리 버퍼에서 응용프로그램 버퍼로 바로 복사
 - 디스크 장치에 요청없이, 버퍼 캐시에서 액세스
 - 여러 응용프로그램들이 동일한 파일을 액세스하는 경우 효과적
 - 버퍼 캐시의 파일 블록이 여러 응용프로그램에 의해 공유
 - 디스크 장치에 요청없이 버퍼캐시에서 읽기
 - 한 프로그램이 순차적으로 파일 데이터를 읽는 경우 효과적
 - 시스템 호출 없이, 라이브러리 버퍼에서 응용 프로그램 버퍼로 바로 복사

2. 파일 시스템의 논리 구조

파일 시스템 구조

19

- 오늘날 대부분 트리 계층 구조의 파일 시스템 구성
 - ▣ 디렉터리와 파일의 트리 구조
 - 루트 디렉터리(root directory) - 계층 구조의 최상위 디렉터리
 - 서브 디렉터리(sub directory) - 하부 디렉터리들
 - 파일
 - 디렉터리도 하나의 파일

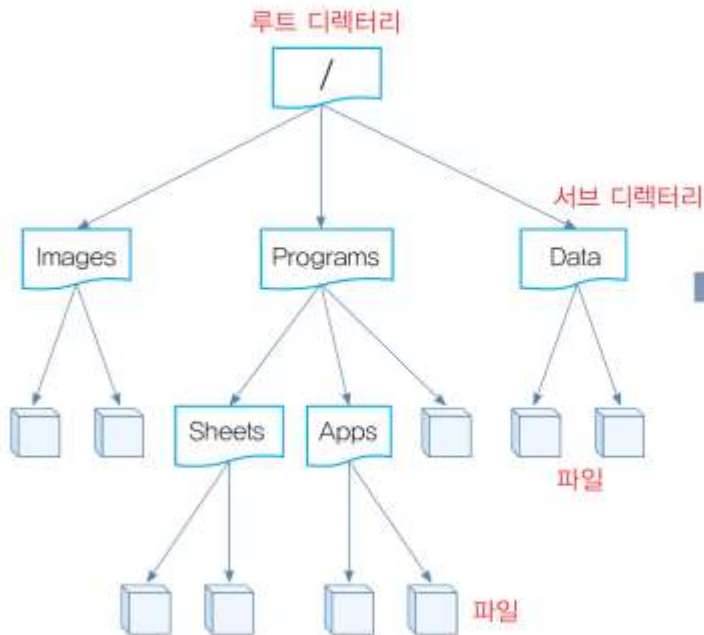


파일 시스템의 논리적 구성과 물리적 구성

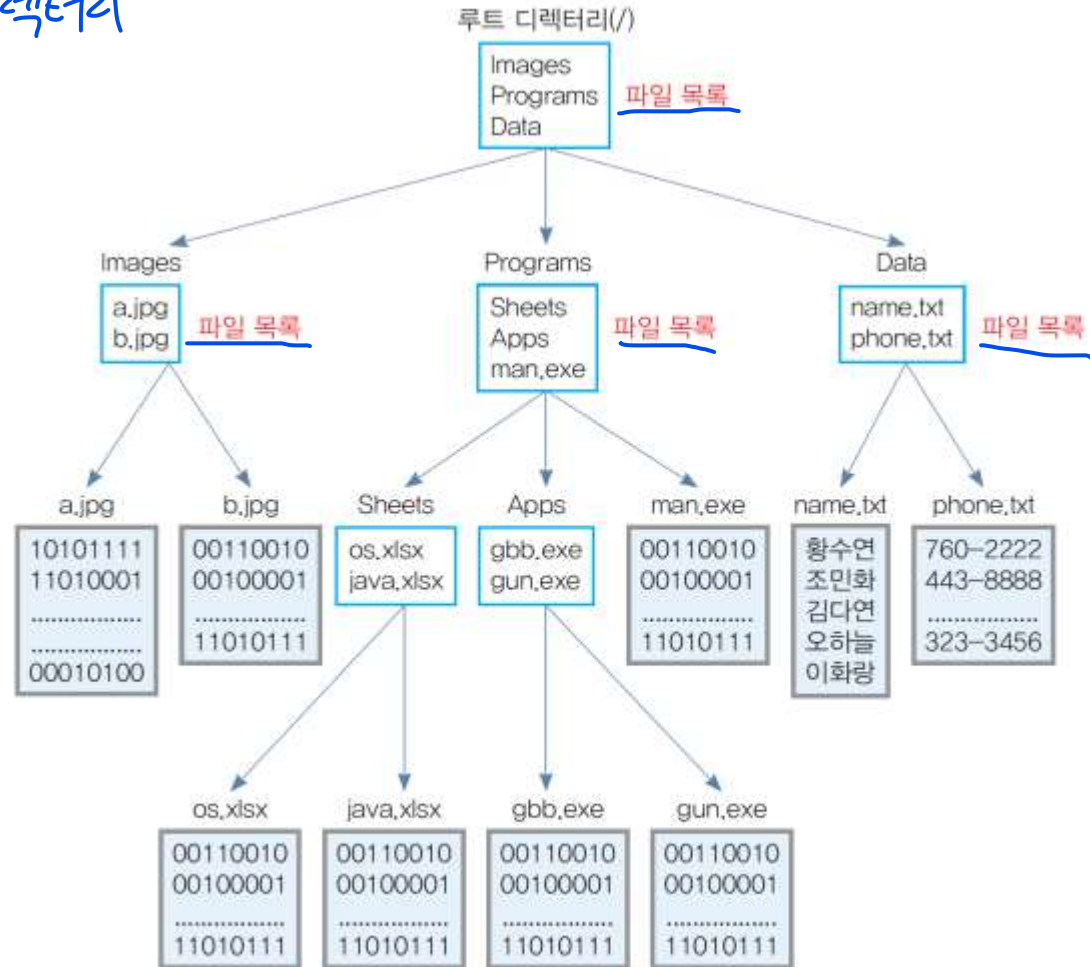
20

* 디렉터리는 서브 디렉터리나 파일들의 목록을 저장한 파일

루트 디렉터리



(a) 파일 시스템의 논리 구성



(b) 파일 시스템의 물리 구성

디렉터리(폴더)와 파일 경로명

21

□ 디렉터리

▣ 논리적인 관점

- 여러 파일 혹은 서브디렉터리를 포함하는 컨테이너
- 파일에 대한 경로 제공

▣ 물리적인 관점

- 디렉터리도 파일로 구현되고 다루어짐
- 디렉터리는 파일이나 서브디렉터리의 이름, 이들에 관한 위치 정보, 혹은 속성 등을 저장하는 특별한 파일

□ 파일 이름과 경로명

▣ 단순 파일 이름

- gun.exe, a.jpg, main.cpp 등

▣ 파일의 경로명(pathname)

- 루트디렉터리에서부터 파일에 이르기까지의 계층의 경로 모두 포함
- 리눅스 - /Programs/Apps/gun.exe
- Windows - C:\Programs\WApps\Wgun.exe

파일 시스템을 다루기 위한 메타 정보

22

- 운영체제에서 파일 시스템을 다루기 위한 **메타 정보** 없으면 파일 못읽음
 - 1. **파일 시스템 메타 정보** – 파일 시스템 전체에 관한 정보
 - 2. **파일 메타 정보** – 파일에 관한 정보

- 파일 시스템 메타 정보

- 파일 시스템 전체 크기와 현재 사용 크기
- 저장 장치에 구축된 파일 시스템의 비어 있는 크기
- 저장 장치의 빈 블록들 리스트 등
- 저장되는 위치
 - 운영체제마다 다름
 - 운영체제가 쉽게 읽고 쓸 수 있도록 저장 매체의 특별한 위치에 저장

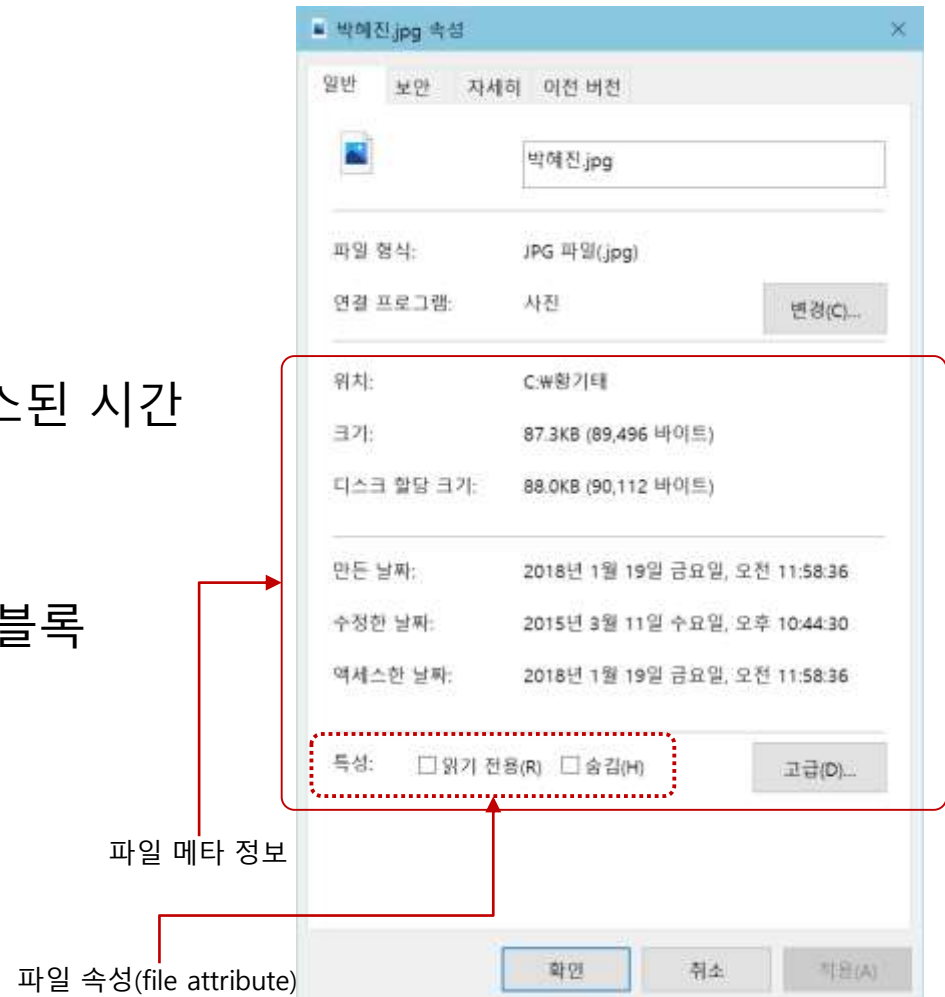
- 파일 메타 정보

- 파일 이름, 파일 크기,
- 파일이 만들어진 시간, 파일이 수정된 시간, 가장 최근에 액세스한 시간,
- 파일을 만든 사용자, 파일 속성(접근 권한),
- 파일이 저장된 위치(파일 블록 배치 정보) 등
- 저장되는 위치
 - 파일 시스템마다 다름
 - 디렉터리 내 혹은 저장 매체의 특별한 위치(예:i-node)에 저장

파일 메타 정보

23

- 파일 메타 정보
 - 파일 이름
 - 파일 크기
 - 파일이 만들어진 시간
 - 파일이 수정된 시간
 - 파일이 가장 최근에 액세스된 시간
 - 파일을 만든 사용자
 - 파일 속성(접근 권한)
 - 파일이 저장된 위치(파일 블록 배치 정보)



Windows에서 C:\Users\User\박혜진.jpg 파일의 메타 정보

Tip. 리눅스의 파일메타정보 중 파일 속성(file attribute)

24

□ 리눅스의 파일 속성

- ▣ r : 읽기 허용
- ▣ w : 파일 쓰거나 수정 허용
- ▣ x : 파일의 실행 허용. 디렉터리 경우 디렉터리 안으로 진입 가능



25

3. 파일 시스템 구축

파일 시스템의 종류와 구현 이슈

26

▣ 파일 시스템 종류

- FAT(File Allocation Table) 파일 시스템
 - MS-DOS에서 사용. 최근에도 사용되고 있음
- UFS(Unix File System) :
 - Unix에서 사용
- ext2, ext3, ext4
 - 리눅스에서 사용
- HFS(Hierarchical File System)
 - Mac 운영체제에서 사용
- NTFS(New Technology File System)
 - Windows3.1부터 지금까지 사용. FAT 개선, 리눅스에서도 지원됨

▣ 파일 시스템 구현 이슈

- 디스크에 파일 시스템 포맷
 - 디스크 장치에 비어 있는 블록들의 리스트를 어떻게 관리할 것인가?
- 파일 블록 할당/배치 관리
 - 파일 블록들을 디스크의 어느 영역에 분산 배치할 것인가?
- 파일 블록 위치 관리
 - 파일 블록들이 저장된 디스크 내 위치들을 어떻게 관리할 것인가?

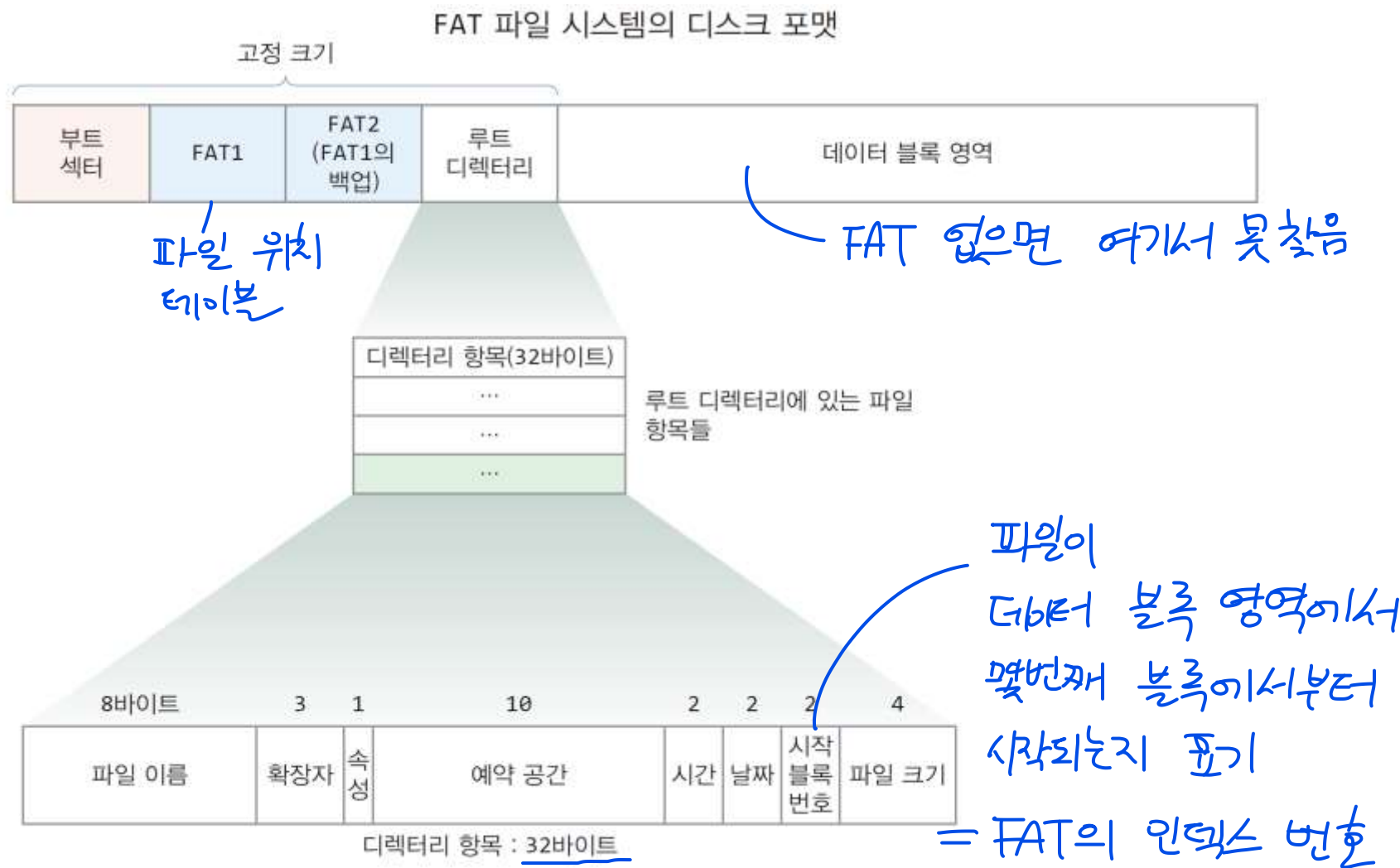
FAT 파일 시스템

27

- 1980년대 PC의 개인용 운영체제인 MS-DOS의 파일 시스템으로 개발
 - ▣ 파일 개수와 크기가 작았던 당시에 적합하도록 설계
 - ▣ 진화된 모습으로 지금도 사용
- 파일 시스템의 구조
 - ▣ 부트 섹터(boot sector)
 - 1섹터 크기, 운영체제인 IO.sys와 msdos.sys를 적재하고 실행시키는 코드
 - ▣ FAT1, FAT2
 - FAT(file allocation table)는 파일 블록들의 할당 테이블
 - FAT2는 복사본
 - ▣ 루트디렉터리
 - 고정 크기이므로 루트디렉터리에 생성되는 파일이나 서브디렉터리의 개수 유한
 - ▣ 데이터 블록들
 - 파일 블록들이 저장되는 곳. 파일은 블록들로 분할되어 분산 저장
- 디렉터리
 - ▣ 파일이나 서브디렉터리의 목록을 담은 특수 파일
 - 파일 이름은 8.3형식 - 이름 최대 8글자, 확장자 3글자
 - ▣ 루트 디렉터리나 서브 디렉터리의 구조 동일
 - ▣ 디렉터리 항목
 - 32바이트 크기로 하나의 파일에 대한 메타 정보 저장

디스크 내 FAT 파일 시스템 구조

28



※ 디렉터리에는 파일 개수와 동일한 수의 항목이 있으며, 각 항목에는 파일 메타 정보가 저장된다.

파일 블록 배치(File Allocation)

29

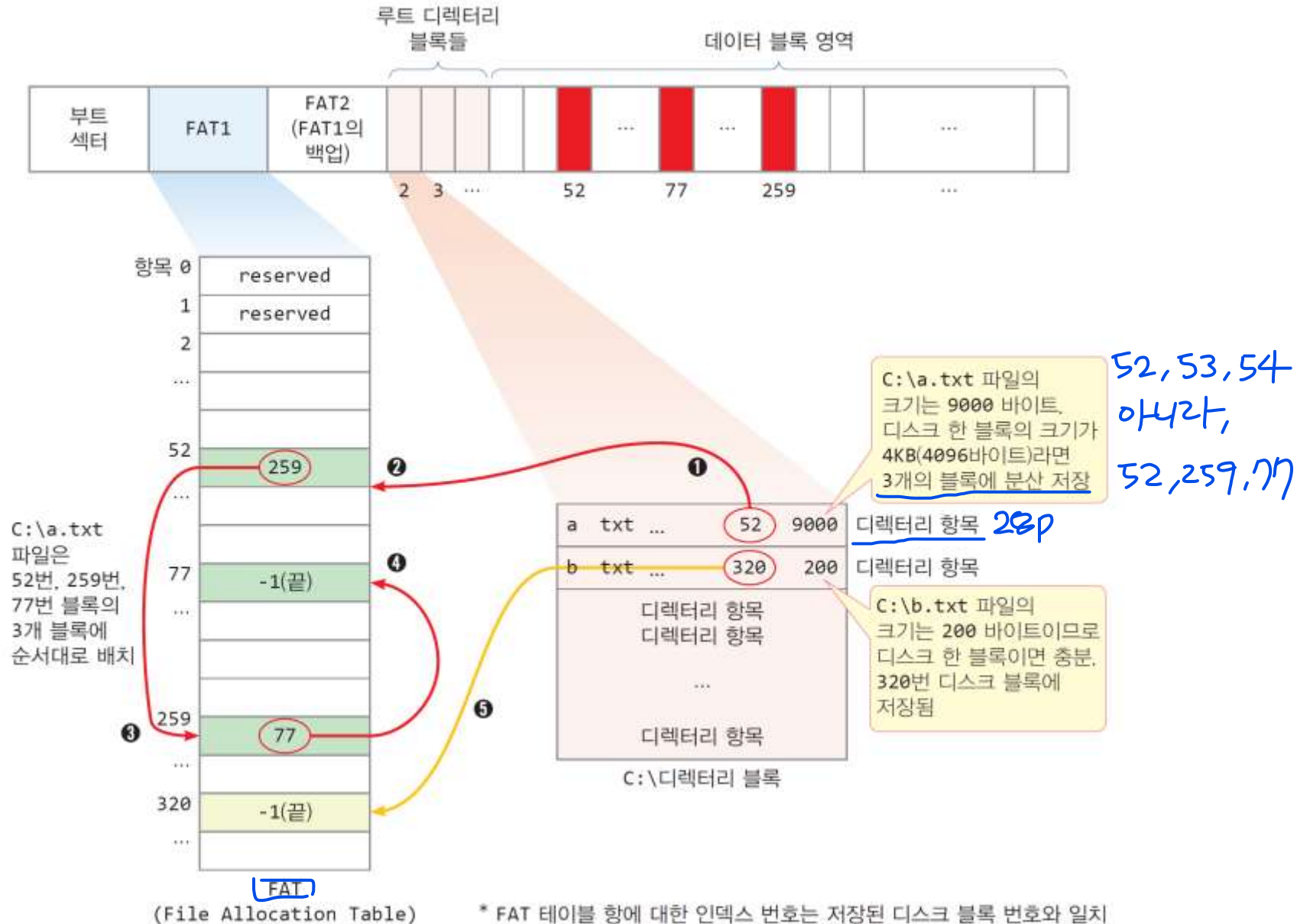
- FAT 파일 시스템의 파일 블록 저장 전략
 - ▣ 파일 데이터를 블록 단위로 디스크에 분산 저장
 - ▣ 파일 메타 데이터는 디렉터리에 저장
 - ▣ 저장된 파일 블록들의 위치는 FAT 테이블에 기록
- FAT 테이블
 - ▣ 파일 시스템에 생성된 모든 파일에 대해, 저장된 파일 블록 번호들이 담겨 있는 테이블
 - ▣ 테이블 항목
 - 디스크 블록 번호는 테이블 항목 번호와 동일
 - 테이블 항목에는 다음 디스크 블록 번호 저장(다음 테이블 항목을 가리킴)
 - 테이블 항목에 저장된 번호가 -1이면 파일 끝을 나타냄
 - 테이블 항목에 저장된 번호가 0이면 빈 블록을 나타냄
 - ▣ FAT 테이블의 항목들이 연결 리스트로 연결
 - ▣ FAT가 손상되면 심각한 문제 - FAT2의 백업으로 해결

파일이 저장된 모든 블록 알아내기

30

- ▣ 파일이 저장된 모든 블록 알아내기
 - 먼저, 파일이 포함된 디렉터리 항목 검색
 - 디렉터리 항목에는 해당 파일이 시작되는 FAT 항목 번호가 저장
 - 디렉터리 항목에는 해당 파일의 크기가 저장되어 있음
 - 디렉터리 항목(파일 크기와 파일 시작 블록 번호)을 이용하여 FAT 테이블을 연결 리스트 방식으로 검색하여 파일이 저장된 블록들을 알아냄(사례는 다음 슬라이드)
- ▣ 하나의 파일을 읽는 데 여러 번 디스크 탐색(seek) 필요
 - 파일이 여러 개의 블록으로 나뉘어 분산 저장되므로
- ▣ FAT 한 항목의 크기가 16비트, 블록이 4KB라면, FAT 파일 시스템이 저장할 수 있는 최대 데이터양
 - 접근 가능한 총 블록 수는 $2^{16}-2$ 개=대략 2^{16} 개
 - $2^{16} \times 4KB = \underline{2^{16} \times 2^{12}}$ 바이트 = 2^{28} 바이트 = 256×2^{20} 바이트 = 256MB

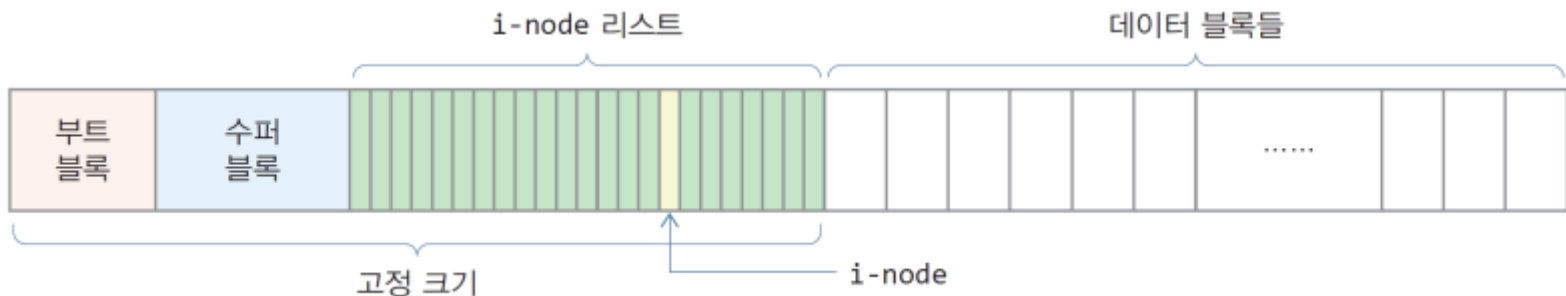
FAT 파일 시스템의 파일 할당 사례



Unix 파일 시스템 구조

32

- 부트 블록(boot block)
 - ▣ 부팅 시 메모리에 적재되어 실행되는 코드로, 운영체제를 적재하는 기능
- 수퍼 블록(super block)
 - ▣ 파일 시스템 메타 정보 저장(매우 중요한 영역)
- i-node와 i-node 리스트
 - ▣ i-node : 파일당 1개의 i-node 필요, 파일 메타 정보 저장
 - ▣ i-node 리스트 : i-node들의 테이블
 - i-node 리스트의 크기는 포맷 시 결정. 포맷 후 i-node개수는 고정
 - ▣ 파일이 생성될 때마다 빈 i-node 할당, 파일 메타 정보 기록
 - ▣ i-node 번호는 0부터 시작. 운영체제마다 첫 i-node 번호가 조금씩 다름
 - ▣ 루트디렉터리의 i-node 번호는 수퍼 블록에 기록. 리눅스의 경우 2, 유닉스의 경우 1
 - 0번 i-node는 오류 처리를 위해 예약
- 데이터 블록들
 - ▣ 파일과 디렉터리가 저장되는 공간



* 디스크에서 전형적인 UNIX 파일 시스템의 구조

수퍼 블록과 i-node

33

□ 수퍼 블록

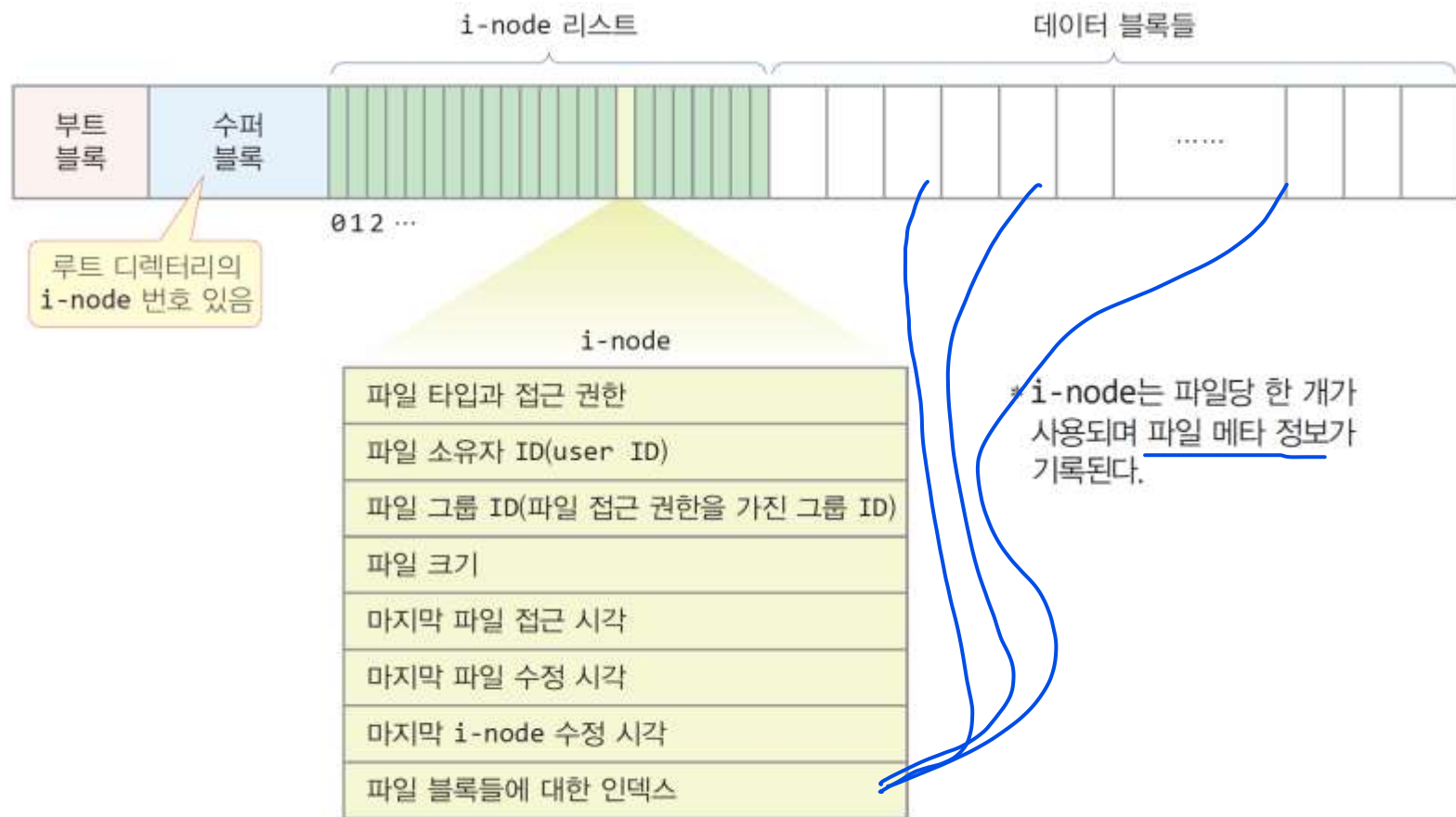
- 파일 시스템의 크기와 상태 정보(수퍼 블록의 수정 여부)
- 파일 시스템 내의 자유 블록 수
- 자유 블록들의 리스트 빈 데이터 블록들
- 자유 블록 리스트에서 요청 시 할당할 다음 블록 인덱스
- 파일 시스템 내의 inode 리스트의 크기
- 파일 시스템 내의 자유 inode 수
- 파일 시스템 내의 자유 inode들의 리스트
- 파일 시스템 내의 자유 inode 리스트에서 요청 시 할당할 다음 자유 inode 인덱스
- 파일 시스템의 논리 블록의 크기
- 루트 디렉터리의 i-node 번호
- 수퍼 블록이 갱신된 최근 시간 자유 inode 때문에 자주 갱신됨

□ i-node

- 파일 타입과 파일 접근 권한
- 파일 소유자
- 파일 그룹
- 파일 크기
- 마지막 파일 접근 시각
- 마지막 파일 수정 시각
- 마지막 i-node 수정 시각
- 파일이 저장된 블록들에 대한 인덱스

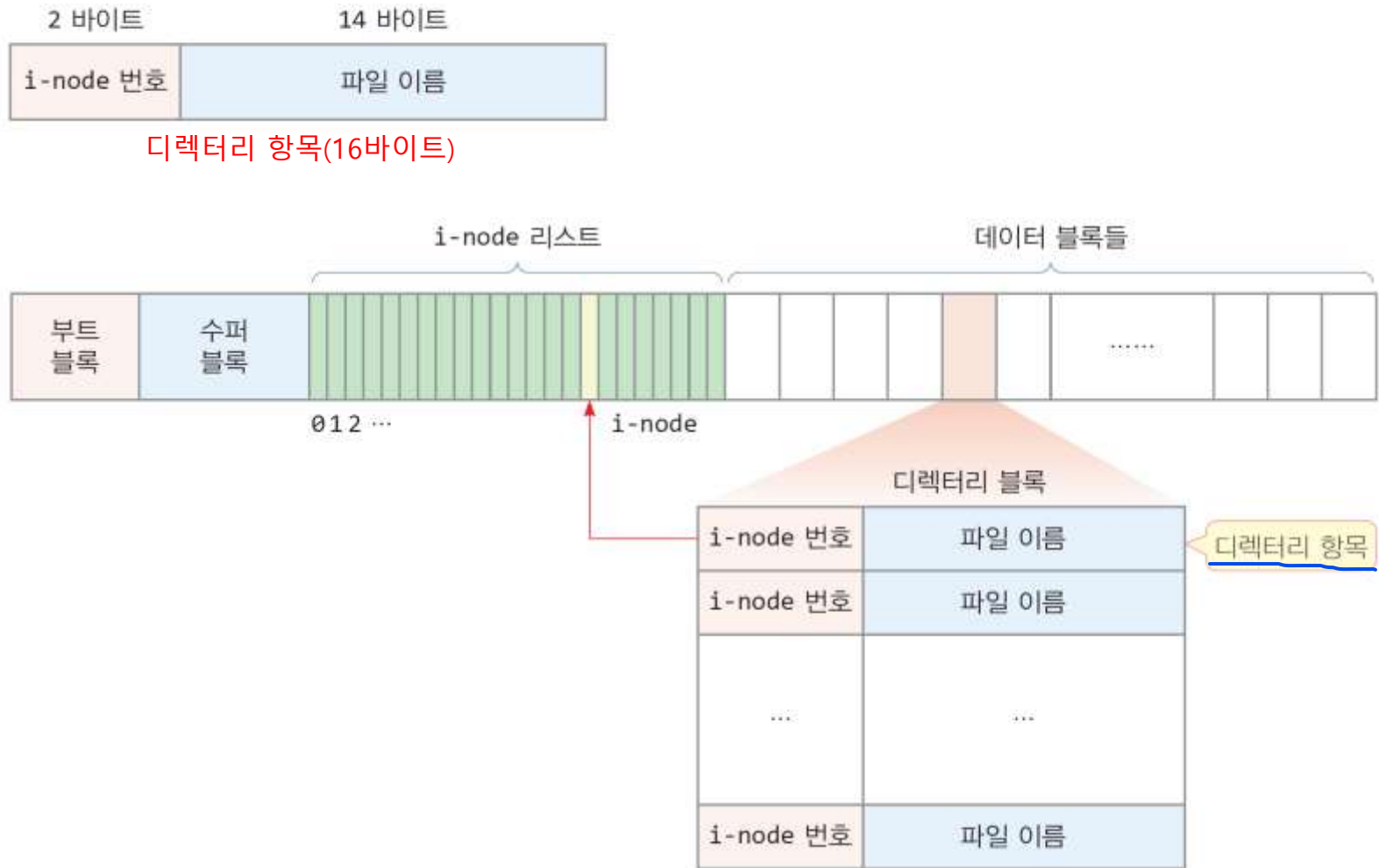
i-node 리스트와 i-node에 저장된 정보

34



디렉터리 블록과 i-node 리스트

35



디렉터리 블록과 디렉터리 항목, 그리고 i-node

Tip. 리눅스에서 파일의 i-node 번호와 메타 데이터 일부 보기

36

- 셸에서 'ls -ial' 명령
 - ▣ 현재 디렉터리에 저장된 파일들의 i-node 번호와 i-node에 들어 있는 파일 메타 정보 출력
 - 파일 이름과 i-node 번호는 디렉터리 항목에 들어 있는 정보가 출력된 것

현재 디렉터리의 파일 리스트 보기 명령

```
$ ls -ial
```

합계 24

9708773	drwxrwxr-x	2	han00	han00	4096	12월 11 16:46	.
9708598	drwxrwxr-x	8	han00	han00	4096	12월 11 16:27	..
9708778	-rwxrwxr-x	1	han00	han00	8640	12월 11 16:30	logical
9708782	-rw-rw-r--	1	han00	han00	90	12월 11 16:30	logicaladdress.c

\$

i-node 번호

i-node에 들어 있는 파일의 메타 데이터 일부

파일 이름

탐구 11-1 Unix 파일 시스템 Q&A

37

Q1. Unix 파일 시스템을 사용할 때 만들 수 있는 파일 개수는 무엇에 의해 달려 있는가?

A. 파일 하나당 하나의 i-node가 필요하므로 i-node 리스트의 크기에 달려 있다. 즉 파일 시스템 전체 i-node 개수에 달려 있다.

Q2. 수퍼 블록이 메모리에 적재된 채 사용되어야 하는 이유는 무엇인가?

A. 파일이 생성될 때마다 자유 i-node를 찾는 등 수퍼 블록은 파일 시스템을 사용하는 동안 계속 필요하므로 커널 코드의 실행을 빨리 하기 위해 메모리에 적재하여 사용하여야 한다.

Q3. Unix 파일 시스템에서 파일 시스템 메타 정보와 파일 메타 정보는 어디에 기록되는가?

A. 파일 시스템 메타 정보는 수퍼 블록에 기록되고, 파일 메타 정보는 파일의 i-node에 기록된다. 파일 이름은 디렉터리의 항목에 기록되어 있다.

탐구 11-1 Unix 파일 시스템 Q&A

38

Q1. Unix 파일 시스템을 사용할 때 만들 수 있는 파일 개수는 무엇에 의해 달려 있는가?

A. 파일 하나당 하나의 i-node가 필요하므로 i-node 리스트의 크기에 달려 있다. 즉 파일 시스템 전체 i-node 개수에 달려 있다.

Q2. 수퍼 블록이 메모리에 적재된 채 사용되어야 하는 이유는 무엇인가?

A. 파일이 생성될 때마다 자유 i-node를 찾는 등 수퍼 블록은 파일 시스템을 사용하는 동안 계속 필요하므로 커널 코드의 실행을 빨리 하기 위해 메모리에 적재하여 사용하여야 한다.

Q3. Unix 파일 시스템에서 파일 시스템 메타 정보와 파일 메타 정보는 어디에 기록되는가?

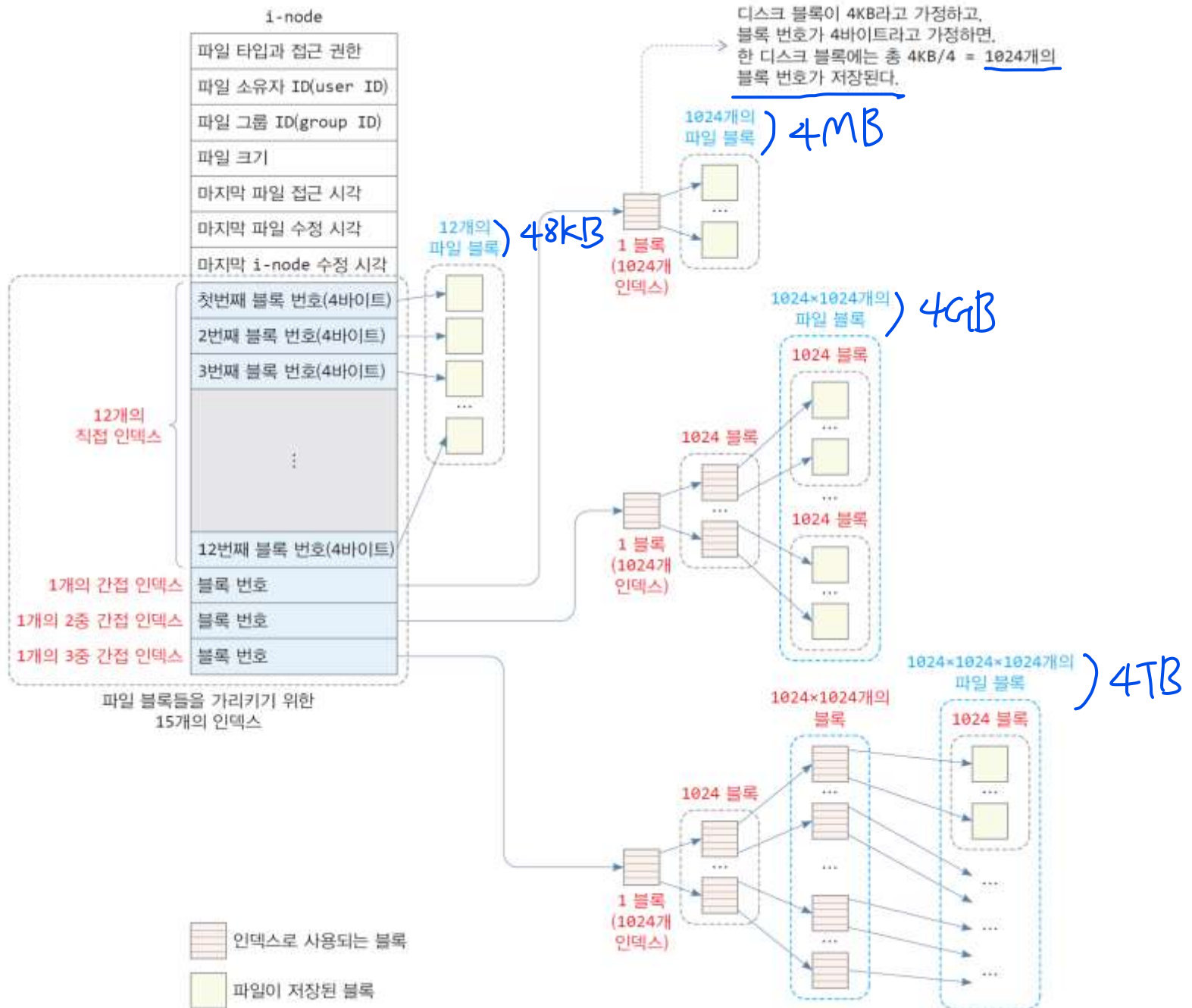
A. 파일 시스템 메타 정보는 수퍼 블록에 기록되고, 파일 메타 정보는 파일의 i-node에 기록된다. 파일 이름은 디렉터리의 항목에 기록되어 있다.

Unix 파일시스템의 파일블록배치(File Allocation)

39

- Unix 파일 시스템은 파일을 블록 단위로 분산 배치
- i-node에 15개 인덱스를 두고 파일 블록들의 위치 정보 저장
 - ▣ 12개의 직접 인덱스
 - 12개의 파일 블록 번호(파일의 앞부분 12개 블록 가리킴)
 - 12개의 직접 인덱스로 가리킬 수 있는 파일 크기 : $12 \times 4\text{KB} = 48\text{KB}$
 - ▣ 1개의 간접 인덱스
 - 파일이 12개의 블록을 넘어갈 때 사용
 - 이 인덱스가 가리키는 한 개의 디스크 블록에 파일 블록 번호들이 들어 있음
 - 한 블록이 4KB이고, 블록 번호가 32비트(4바이트)일 때
 - 간접 인덱스로 가리킬 수 있는 파일 블록 수 : $4\text{KB}/4\text{B} = 1024$ 블록, 파일 크기는 $1024 \times 4\text{KB} = 4 \times 2^{20}\text{바이트} = 4\text{MB}$
 - ▣ 1개의 2중 간접 인덱스
 - 2중 간접 인덱스로 가리킬 수 있는 파일 블록 수 : 1024×1024 블록
 - 2중 간접 인덱스로 가리킬 수 있는 파일 크기 : $1024 \times 1024 \times 4\text{KB} = 4 \times 2^{30}\text{바이트} = 4\text{GB}$
 - ▣ 1개의 3중 간접 인덱스
 - 3중 간접 인덱스로 가리킬 수 있는 파일 블록 수 : $1024 \times 1024 \times 1024$ 블록
 - 3중 간접 인덱스로 가리킬 수 있는 파일 크기 : $1024 \times 1024 \times 1024 \times 4\text{KB} = 4 \times 2^{40}\text{바이트} = 4\text{TB}$
- Unix 파일 시스템에서의 파일 최대 크기
 - ▣ $48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB}$

40




파일의 i-node 찾기

41

on 메모리

- 파일을 읽고 쓰기 위해 파일 블록들의 위치 파악 필요
 - ▣ 파일블록들의 위치는 i-node의 15개 인덱스를 통해 알 수 있음
 - ▣ 그러므로 파일의 i-node를 찾아야 함

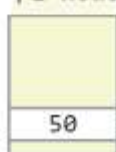
 - /usr/source/main.c 사례로 파일의 i-node 찾는 과정
 1. 루트디렉터리(/)의 i-node 번호 알아내기
 - ▣ 루트디렉터리의 i-node 번호는 수퍼 블록에 있음
 2. 루트디렉터리(/)의 i-node 읽기
 - ▣ 수퍼 블록에 적힌 루트디렉터리의 i-node 번호로부터 i-node 읽기
 3. 루트 디렉터리에서 /usr의 i-node 알아내기
 4. /usr 디렉터리를 읽고 /usr/source 파일의 i-node 번호 알아내기
 5. /usr/source 디렉터리 읽고 /usr/source/main.c 파일의 i-node 번호 알아내기
 6. /usr/source/main.c 파일 읽기
- 



❶ 수퍼 블록에서 /의 i-node 번호 알아내기

❷ /의 i-node 2

직접 인덱스



블록 50

i-node 번호	파일 이름
2	.
2	..
16	etc
25	usr
29	dev
66	lib

루트(/) 디렉터리

/usr의 i-node 25

직접 인덱스



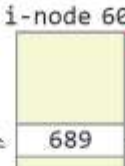
블록 312

25	.
2	..
33	bin
60	source
98	book

/usr 디렉터리

/usr/source의 i-node 60

직접 인덱스



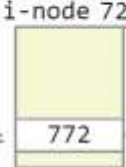
블록 689

60	.
25	..
72	main.c
87	util.c
92	game.c
99	result.c

/usr/source 디렉터리

main.c의 i-node 72

직접 인덱스



블록 772

```
#include <stdio.h>
int main() {
    ....
}
```

/usr/source/main.c 파일

4. 파일 입출력 연산

파일 입출력 연산

44

- 커널의 파일 시스템은 파일 입출력을 위한 시스템 호출 함수 제공
 - ▣ `open()`, `read()`, `write()`, `close()`, `chmod()`, `create()`, `mount()`, `unmount()` 등

- 파일 찾기
 - ▣ 파일의 경로명으로부터 파일의 i-node를 찾는 과정
 - 예) `open("/usr/source/main.c", O_RDONLY)` 가 호출되면 `open()` 은 “/usr/source/main.c” 경로명의 i-node를 찾는다.
 - 앞 절에서 설명하였으므로 생략
 - ▣ i-node에 파일 타입, 접근 권한, 파일 데이터가 담겨 있는 블록 번호 등이 들어 있기 때문
 - ▣ 커널에 의해 수행

파일 열기, open()

45

□ 왜 파일을 열어야 할까?

- 파일이 존재하는지 확인,
- 현재 프로세스가 파일에 접근해도 되는지 접근 권한 확인,
- 파일을 읽고 쓰기 위한 커널 내에 자료 구조 형성

□ 파일 입출력을 위한 커널 내 자료 구조들

■ 메모리 i-node 테이블

- 열린 파일의 디스크 i-node를 읽어 메모리 내에 저장한 테이블
- 파일 블록 위치 등 i-node를 액세스할 때 빠른 처리를 위해 메모리에 적재

■ 오픈 파일 테이블(open file table)

- 시스템에서 열린 모든 파일에 대한 정보 - 파일 오피셋, 파일 액세스 모드, 메모리에 적재된 inode 주소
- 모든 프로세스에 의해 공유

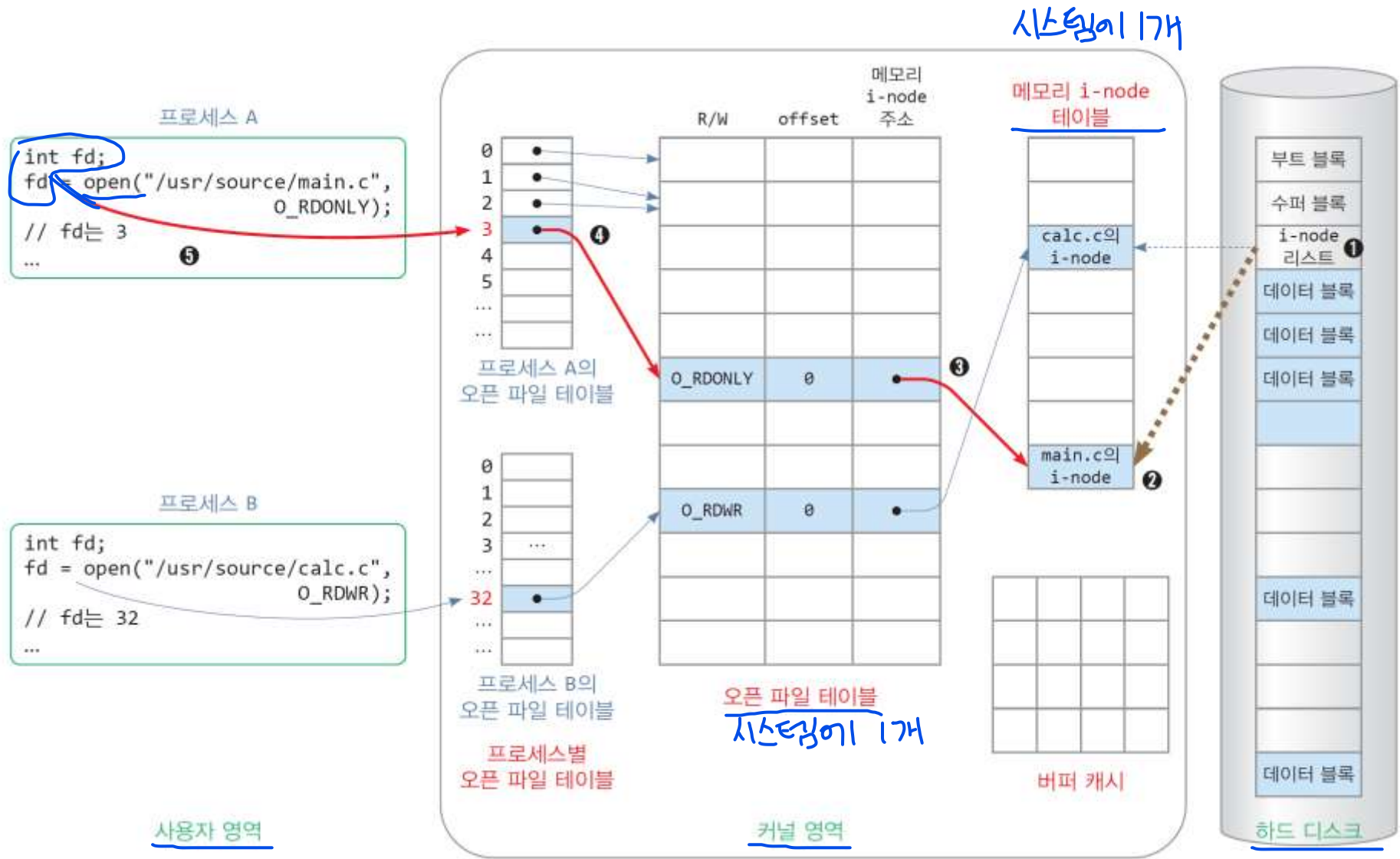
■ 프로세스별 오픈 파일 테이블(per-process open file table)

- 프로세스당 하나씩 있음
- 프로세스가 파일을 열 때마다 한 항목 할당, 오픈 파일 테이블 가리킴
- 항목 번호는 open()의 리턴 값이며 파일 디스크립터로 불림
- 운영체제는 프로세스를 실행시킬 때 표준 입력, 표준 출력, 표준 오류 용으로 3개의 항목을 열어 놓음, 각각 0, 1, 2 항목
- 이 테이블은 PCB에 저장, 프로세스의 모든 스레드에 의해 공유

■ 버퍼 캐시(buffer cache)

- 읽혀지거나 쓰여지는 파일 블록들이 일시적으로 저장되는 메모리
- 디스크 블록 번호로만 관리

파일 열기 후 형성되는 커널 내 구조



파일 열기 과정

47

1. 파일 이름으로 i-node 번호를 알아내기
 - 파일이 존재하지 않거나 접근 권한이 허용되지 않으면 오류 리턴
2. 디스크 i-node를 커널 메모리의 i-node 테이블에 적재
3. 오픈 파일 테이블에 새 항목 만들기
 - i-node 주소 기록
4. 프로세스별 오픈 파일 테이블에 새 항목 만들기
 - 프로세스별 오픈 파일 테이블에 항목 할당
 - 파일 테이블의 주소 기록
5. 프로세스별 오픈 파일 테이블 항목 번호 리턴
 - 프로세스별 오픈 파일 테이블의 항목 번호(정수) 리턴
 - 응용프로그램이 `open()`으로부터 리턴받은 정수는 프로세스별 오픈 파일 테이블 항목 번호임
 - 이 정수를 **파일 디스크립터(file descriptor)**라고 부름 *fd*
 - 응용프로그램은 파일 입출력 시 이 정수를 반드시 사용

파일 읽기 read() 과정

프로세스 A

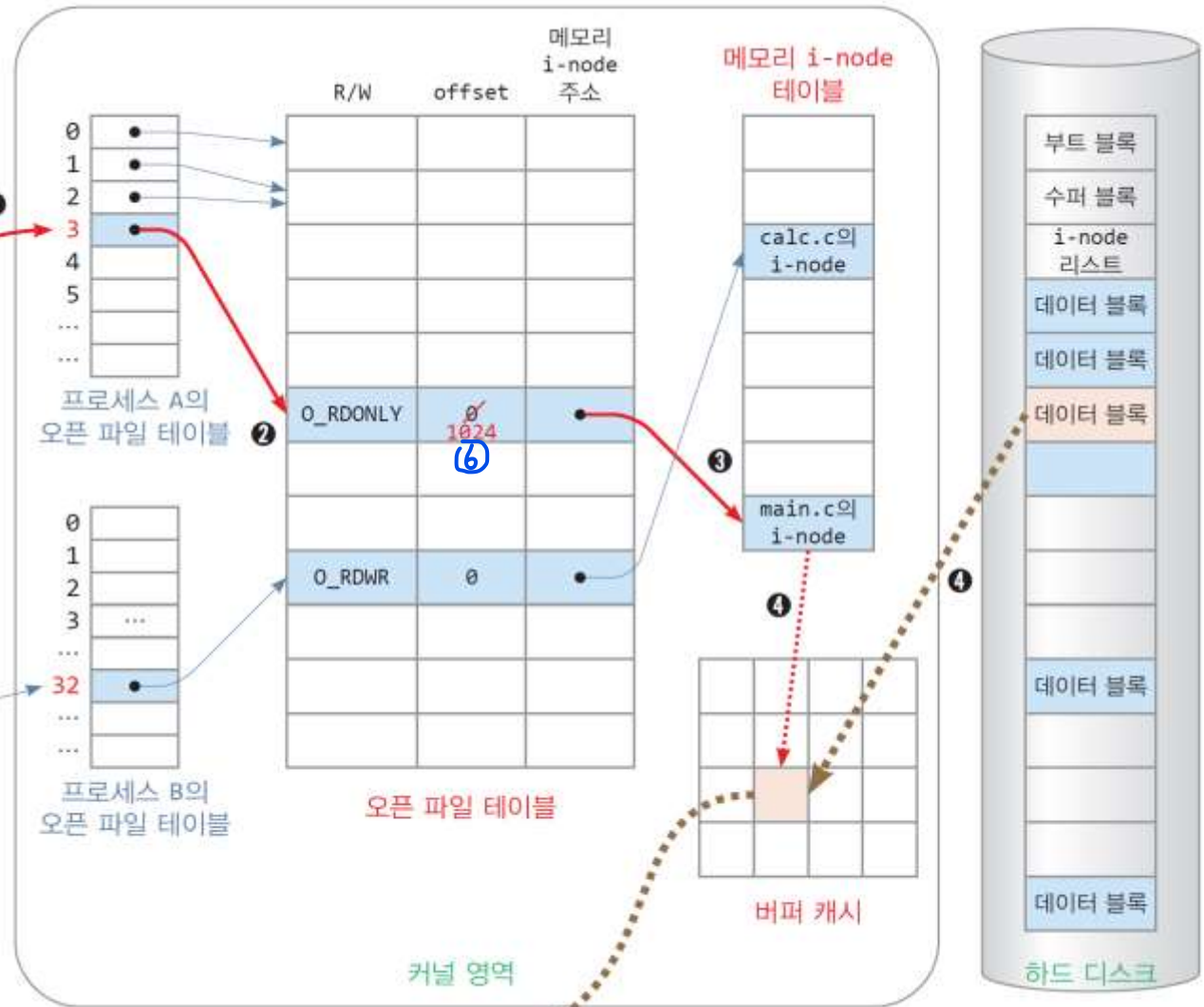
```
char buf[1024];
int fd;
fd = open("/usr/source/main.c",
          O_RDONLY);
// fd는 3
read(fd, buf, 1024);
```

프로세스 B

```
int fd;
fd = open("/usr/source/calc.c",
          O_RDWR);
// fd는 32
...
```

사용자 영역

커널 영역



char buf[1024]

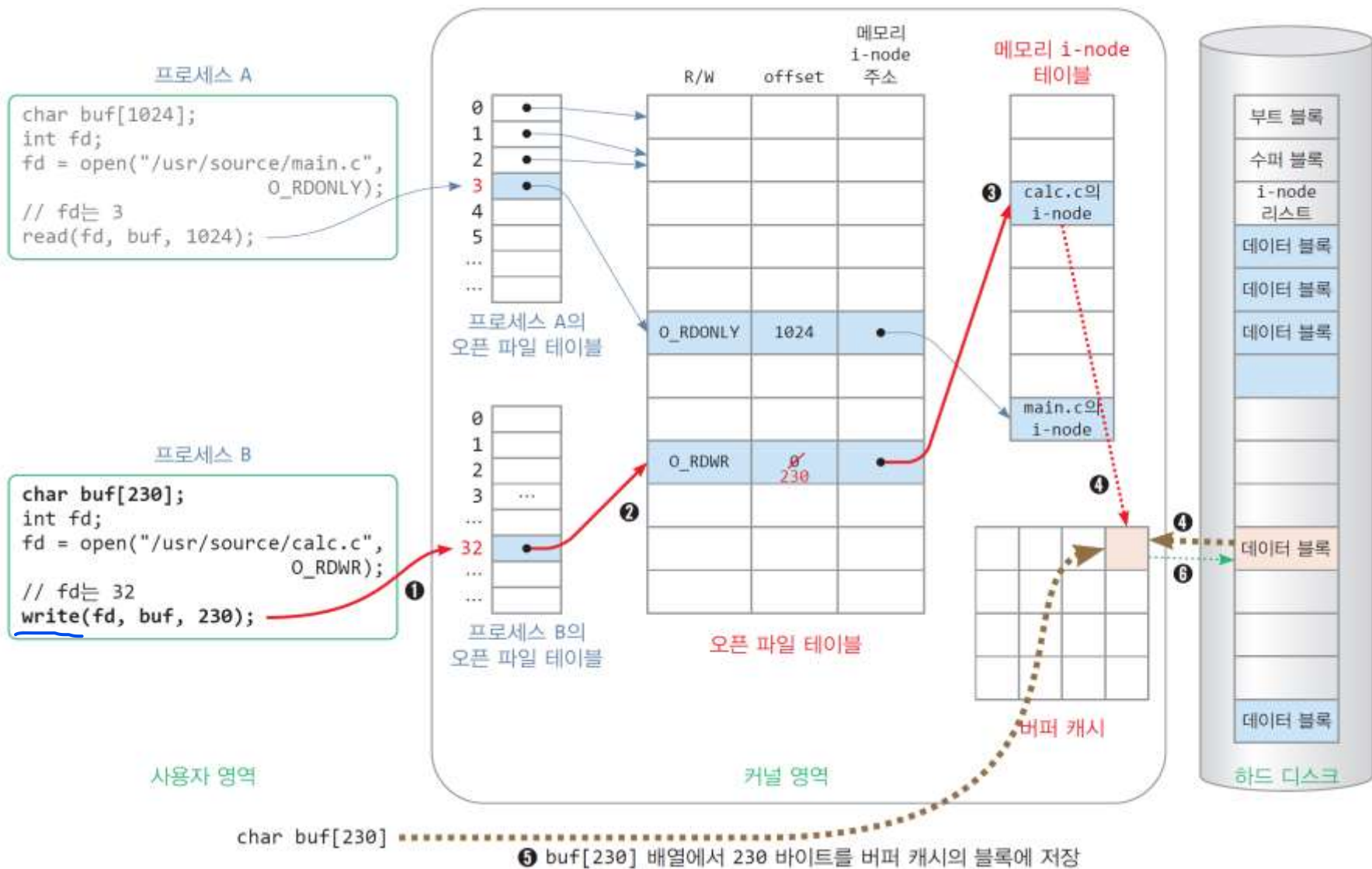
⑥ 버퍼 캐시의 블록에서 1024바이트를 사용자 buf[1024] 배열에 저장

파일 읽기 과정

49

1. read(fd,...)는 fd 번호의 프로세스별 오픈 파일 테이블 참조
2. 파일 테이블 참조
 - R 모드(읽기 허용)가 아닌 경우 오류로 리턴
3. i-node 참조
 - i-node에서 파일 블록들의 리스트 확보
 - 파일 테이블 항목에 적힌 offset 확인
 - offset을 파일 블록 번호로 변환
4. 해당 블록이 버퍼 캐시에 있는지 확인
 - 해당 블록이 버퍼 캐시에 없으면, 버퍼 캐시를 할당받고 디스크에서 버퍼 캐시로 읽기
 - 할당 받은 버퍼 캐시가 dirty이면 버퍼 캐시에 들어 있는 블록을 디스크에 쓰기
5. 버퍼 캐시로부터 사용자 영역으로 블록 복사

파일 쓰기 write() 과정



파일 쓰기 과정

51

1. write(fd,...)는 fd 번호의 프로세스별 파일 테이블을 참조
2. 파일 테이블 참조
 - W 모드(쓰기 허용)가 아니면 오류로 리턴
3. i-node 참조
 - i-node에서 파일 블록들의 리스트 확보
 - 파일 테이블 항목에 적힌 offset 확인
 - offset을 파일 블록 번호로 변환
4. 해당 블록이 버퍼 캐시에 있는 지 확인
 - 해당 블록이 버퍼 캐시에 있으면, 사용자 영역에서 버퍼 캐시에 쓰기
 - 해당 블록이 버퍼 캐시에 없으면, 버퍼 캐시를 할당 받고 디스크 블록을 버퍼 캐시로 읽어 들이기
5. 사용자 공간의 버퍼에서 버퍼 캐시로 쓰기
6. 추후 버퍼 캐시가 교체되거나 플러시 될 때, 버퍼 캐시의 내용이 저장 장치에 기록

파일 닫기 close() 과정

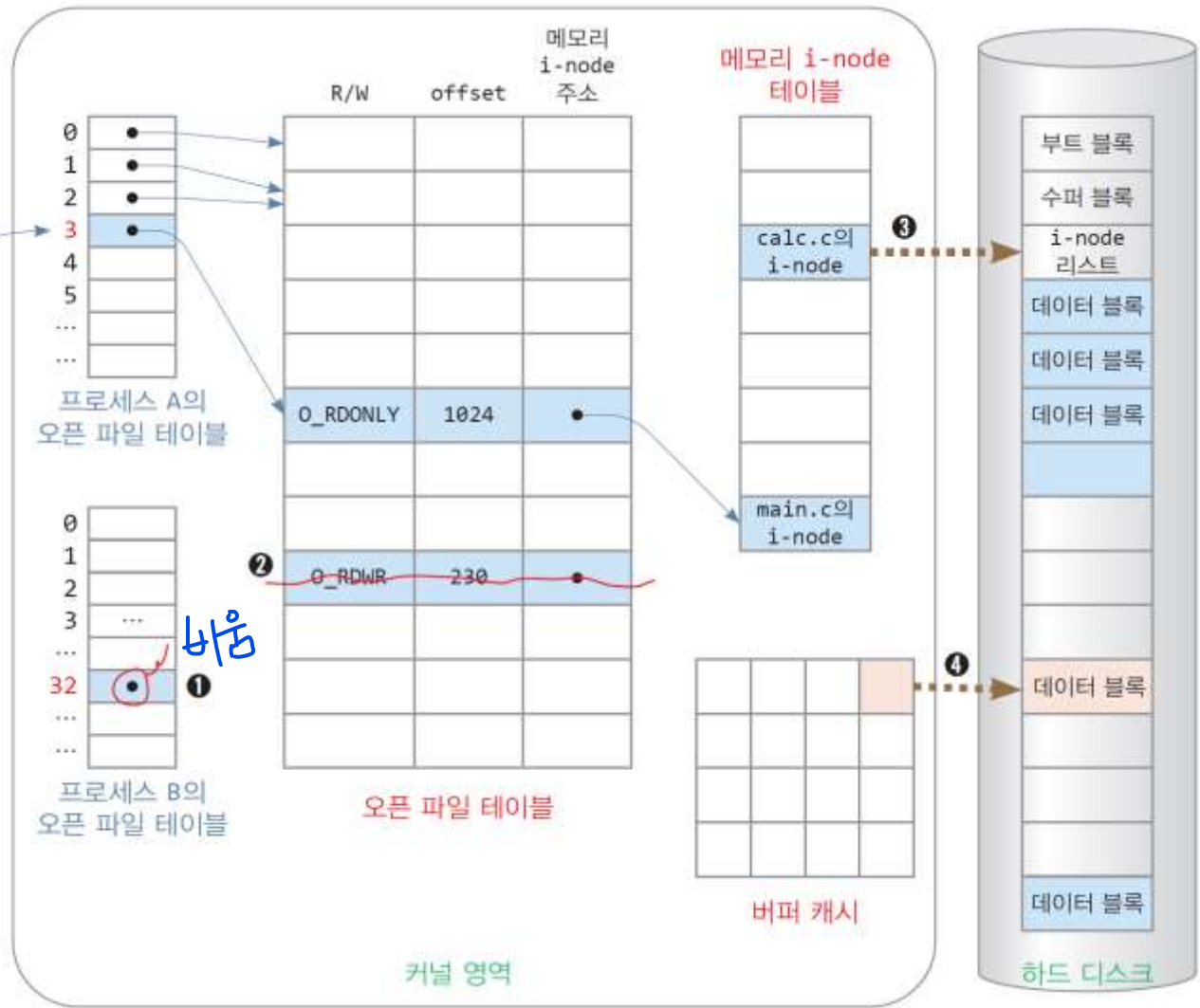
프로세스 A

```
char buf[1024];
int fd;
fd = open("/usr/source/main.c",
          O_RDONLY);
// fd는 3
read(fd, buf, 1024);
```

프로세스 B

```
char buf[230];
int fd;
fd = open("/usr/source/calc.c",
          O_RDWR);
// fd는 32
write(fd, buf, 230);
close(fd);
```

사용자 영역



파일 닫기 과정

53

1. 프로세스의 오픈 파일 테이블 항목에 기록된 내용 지우기
2. 프로세스의 오픈 파일 테이블 항목을 지우기 전, 파일 테이블의 항목을 찾고 (지우고) 반환하기
3. 파일 테이블 항목을 반환하기 전, 메모리 i-node의 사용 해제
4. 버퍼 캐시에 있는 이 파일의 블록들이 수정되었거나 새로 만든 블록인 경우 디스크에 기록

탐구 11-2 동일한 파일이 읽기와 쓰기 모드로 동시에 열린 경우

54

두 프로세스가 phone.txt 파일에 대해 동시에 읽고/쓰는 경우 어떤 상황이 벌어질까?

- 프로세스 A : phone.txt 파일에서 읽기
- 프로세스 B : phone.txt 파일에 쓰기
 - 두 프로세스 중 누가 먼저 실행되느냐에 따라 읽기와 쓰기의 결과가 달라진다
- 프로세스 B가 먼저 실행된 후 프로세스 A가 실행되는 경우
 - 1) 프로세스 A와 B가 각각 열기를 실행함
 - 오픈 파일 테이블에 두 개의 항목 생성
 - 2) 프로세스 B가 phone.txt 파일의 앞부분에 230바이트 기록
 - phone.txt 파일의 앞부분이 수정됨
 - 오픈 파일 테이블에 기록된 offset이 0이므로 파일의 0번 오프셋 위치에 기록
 - 파일 offset은 230으로 변경
 - 3) 프로세스 A가 phone.txt파일에서 1024바이트 읽기
 - 오픈 파일 테이블에 기록된 offset이 0이므로 파일의 0번 오프셋 위치에서 읽기
 - 프로세스 B가 기록한 데이터 230바이트와 그 뒤 이미 저장되어 있는 794바이트 읽음
 - 오픈 파일 테이블의 offset은 1024로 변경

