
CSE 206 - 파일 처리론 (File Processing)

3 장. File I/O Control

Contents

- **3.1. I/O Control System**
- **3.2. Directory Management**
- **3.3. Device Management**
- **3.4. Buffer Management**
- **3.5. Unix 에서의 I/O**

3.1. I/O Control System

Operating System (OS, 운영체제) : 컴퓨터의 구조

입력장치 (Input Device):

외부의 정보(주위 환경 등) 및 사람의 명령을 컴퓨터에 알려준다.



출력 장치 (Output Device):

연산(계산) 결과를 명령을 사람에게 알려준다.



CPU (Central Processing Unit):

명령에 따라 정보를 처리하여 결과를 생성한다. 정보 처리를 위해 메모리에 저장된 정보를 읽고 생성된 결과를 메모리에 저장한다.



사용자는 이 부분에 대해 몰라도 컴퓨터에 명령하고 결과를 받아 볼 수 있다.

1차 저장장치(Memory)

: 계산에 필요한 정보를 저장해 두는 장치. 빠르게 접근할 수 있으나 용량이 상대적으로 작다. 전원이 꺼지면 정보는 모두 사라진다.

(책상위의 메모지, 연습장)



2차 저장장치(HDD)

: 계산에 필요한 정보를 저장해 두는 장치. 용량이 상대적으로 크나 접근 속도가 느리다. 전원이 꺼져도 정보는 계속 저장된다. (책장에 꽂아 둔, 책 혹은 잘 정리된 노트)

1 + 1 = ?

2

Operating System (OS, 운영체제) : 사용자 (응용 프로그램)와 기계 사이의 중개자.

입력장치 (Input Device):

외부의 정보(주위 환경 등) 및 사람의 명령을 컴퓨터에 알려준다.



1 + 1=?

OS

CPU (Central Processing Unit):

명령에 따라 정보를 처리하여 결과를 생성한다. 정보 처리를 위해 메모리에 저장된 정보를 읽고 생성된 결과를 메모리에 저장한다.



사용자는 이 부분에 대해 몰라도 컴퓨터에 명령하고 결과를 받아 볼 수 있다.

1차 저장장치(Memory)

: 계산에 필요한 정보를 저장해 두는 장치. 빠르게 접근할 수 있으나 용량이 상대적으로 작다. 전원이 꺼지면 정보는 모두 사라진다.

(책상위의 메모지, 연습장)



2차 저장장치(HDD)

: 계산에 필요한 정보를 저장해 두는 장치. 용량이 상대적으로 크나 접근 속도가 느리다. 전원이 꺼져도 정보는 계속 저장된다.
(책장에 꽂아 둔, 책 혹은 잘 정리된 노트)

출력 장치 (Output Device):

연산(계산) 결과를 명령을 사람에게 알려준다.



I/O Control System (입출력 제어 시스템)

- **os의 기능**

- **Main memory management**

- 각 프로세스의 요청에 대해 메모리의 어디를 얼마만큼 할당하고, 다 사용한 메모리를 회수하거나 하는 일.

- **Process management**

- 프로세스 (프로그램) 띄우고 프로세스 실행하는데 필요한 정보 (프로그램 카운터 등) 관리
 - 프로세스가 다른 프로세스의 메모리에 접근하려 하면 죽여 버리기 (e.g., Segmentation fault)

- **Job schedule**

- 어떤 프로세스한테 CPU 할당했다가 빼앗아서 다른 프로세스 주었다가 다시 빼앗았다가.
 - 여러 작업들을 효과적으로 수행하기 위해 실행순서를 정해 실행시키기 등.

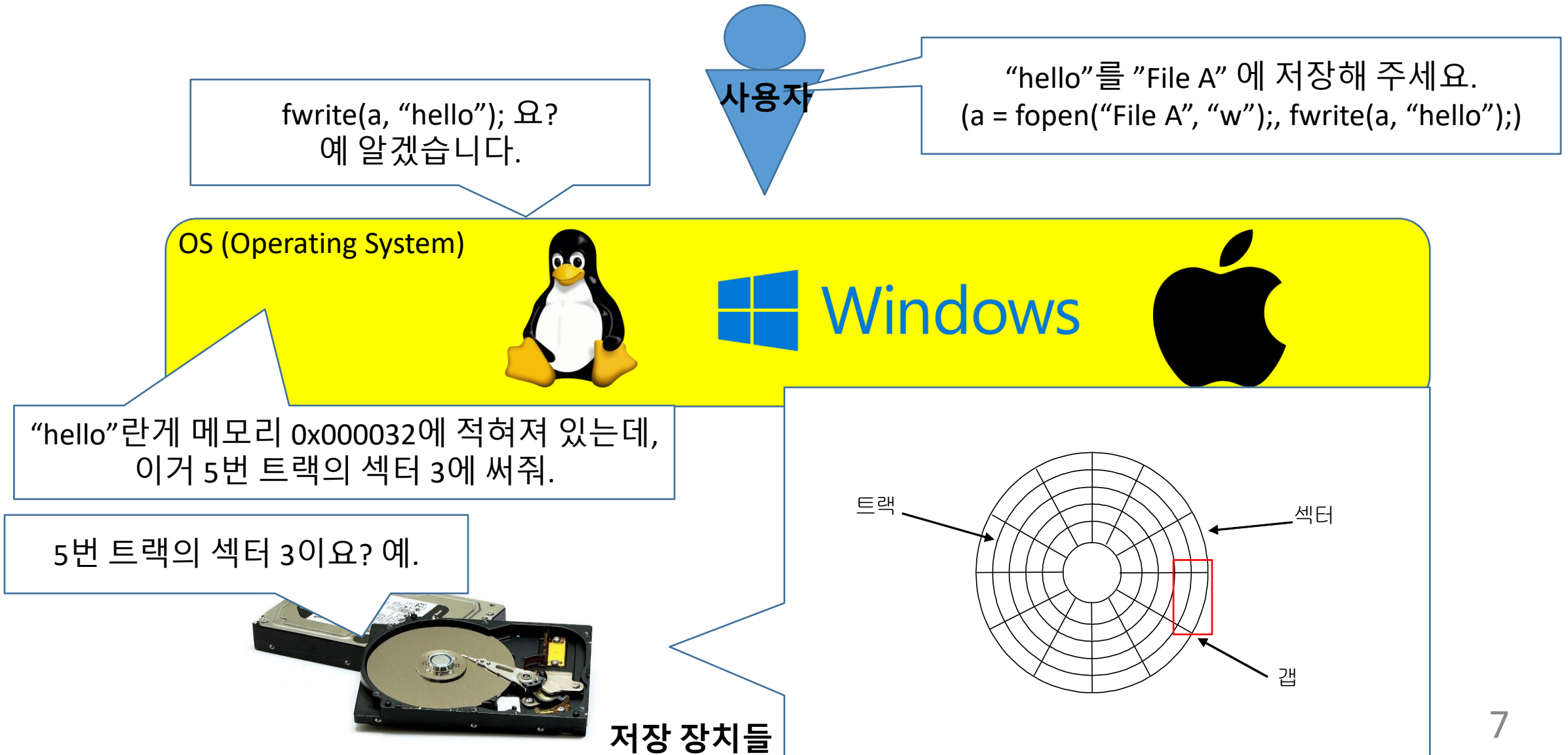
- **File management (파일 관리)**

- **Device management (장치 관리)**

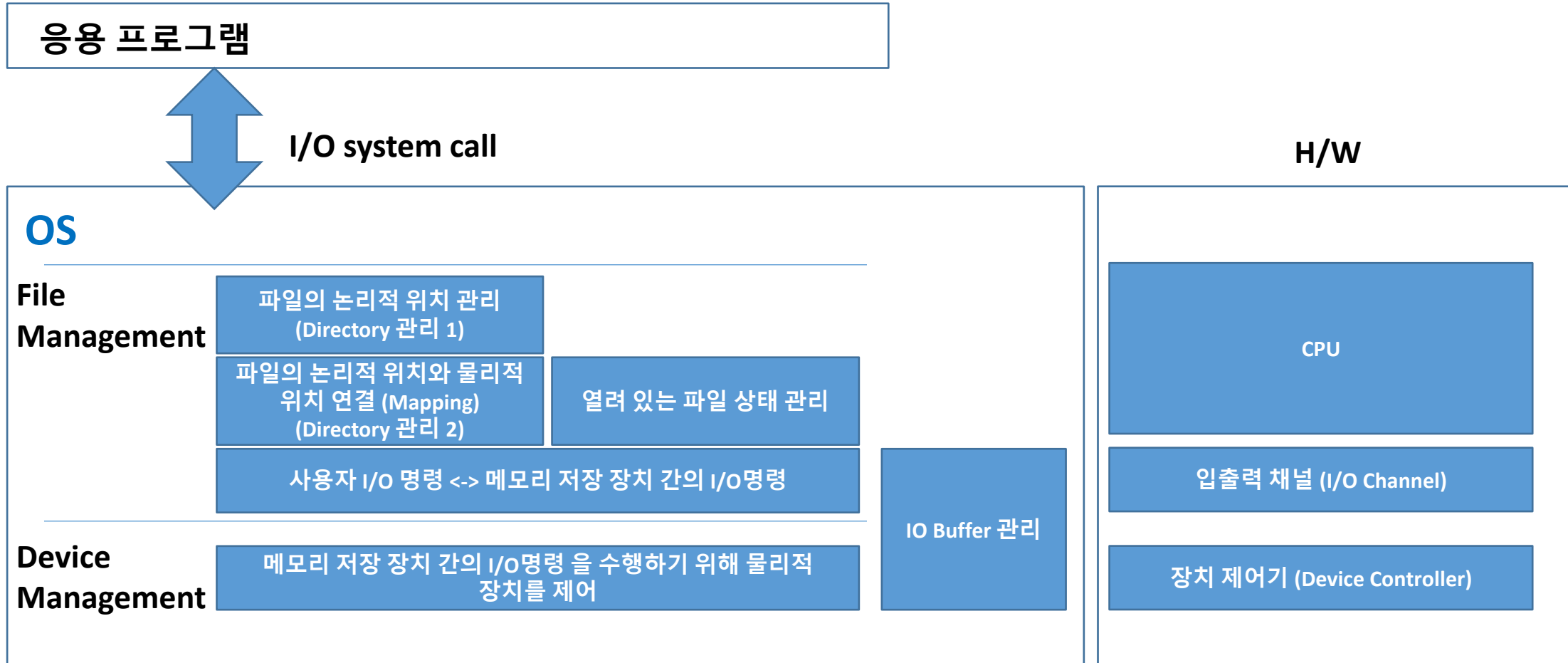
File I/O Control

입출력 제어 시스템

File I/O Control: 논리적 파일 연산을 물리적 장치 작동으로 변환



File I/O Control



I/O Control System – File Management

- **File management (파일 관리)**
 - **Directory Management (디렉터리 관리)**
 - 파일의 논리적 위치 관리
 - 사용자가 파일 찾기 위한 구조 관리
 - 파일의 논리적 위치와 물리적 위치 연결관리
 - 파일이 Disk의 어디에 저장되어 있는가?
 - **열린 파일 관리**
 - 현재 열려 있는 파일들의 상태 관리
 - Memory (Buffer) 확보, File Cursor 관리 등.
 - **I/O Abstraction**
 - 논리적 I/O 명령 (READ/WRITE)을
 - 메모리<->저장 장치 간의 기계적 데이터 전송 명령으로 변환

File Management – Directory 관리

- File을 열어서 데이터를 읽어올 경우를 생각해 보자?
 - 사람이 열 File을 찾아야 한다.
 - 어떻게 찾을까?
 - 열 File을 찾았다면 파일을 열고 파일에 저장된 데이터를 읽어와야 한다.
 - 파일에 저장된 데이터는 어디에 저장되어 있을까?
 - HDD? 좀더 구체적으로는 HDD의 어디?



Directory Management (디렉터리 관리)

파일의 논리적 위치 관리

사용자가 파일 찾기 위한 구조 관리

파일의 논리적 위치와 물리적 위치 연결관리

파일이 Disk의 어디에 저장되어 있는가?

File Management – 열린 파일 관리

- 파일을 성공적으로 열었다고 가정하자.
 - 파일을 Open한 다음에 올 file operation은?
 - CPU가 처리에 사용할 데이터는 어디에 저장되나?
 - Hint : C의 malloc()
 - Application program이 fseek()를 사용하면?



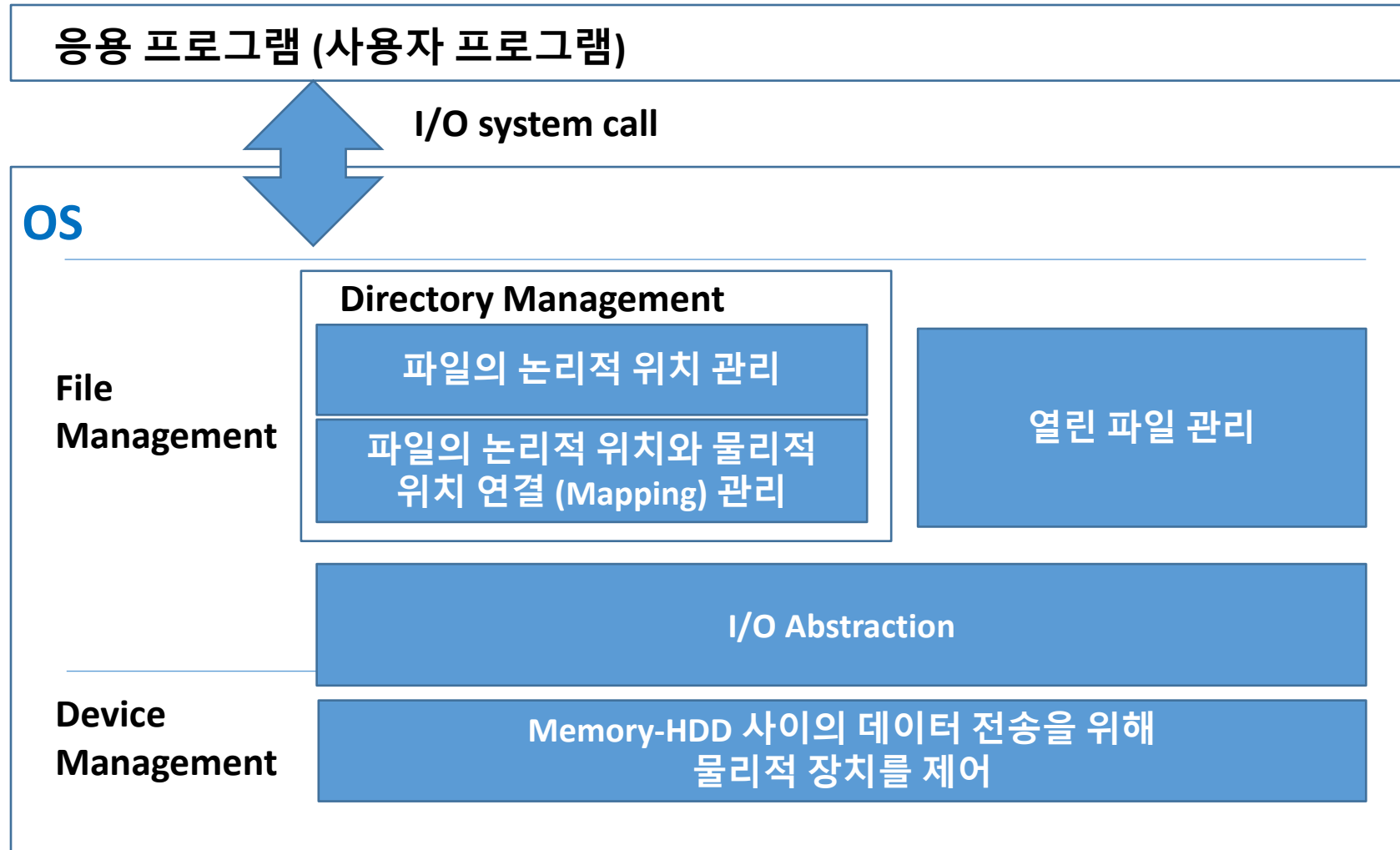
열린 파일 관리

현재 열려 있는 파일들의 상태 관리
Memory (Buffer) 확보, File Cursor 관리 등.

Device Management – I/O Abstraction

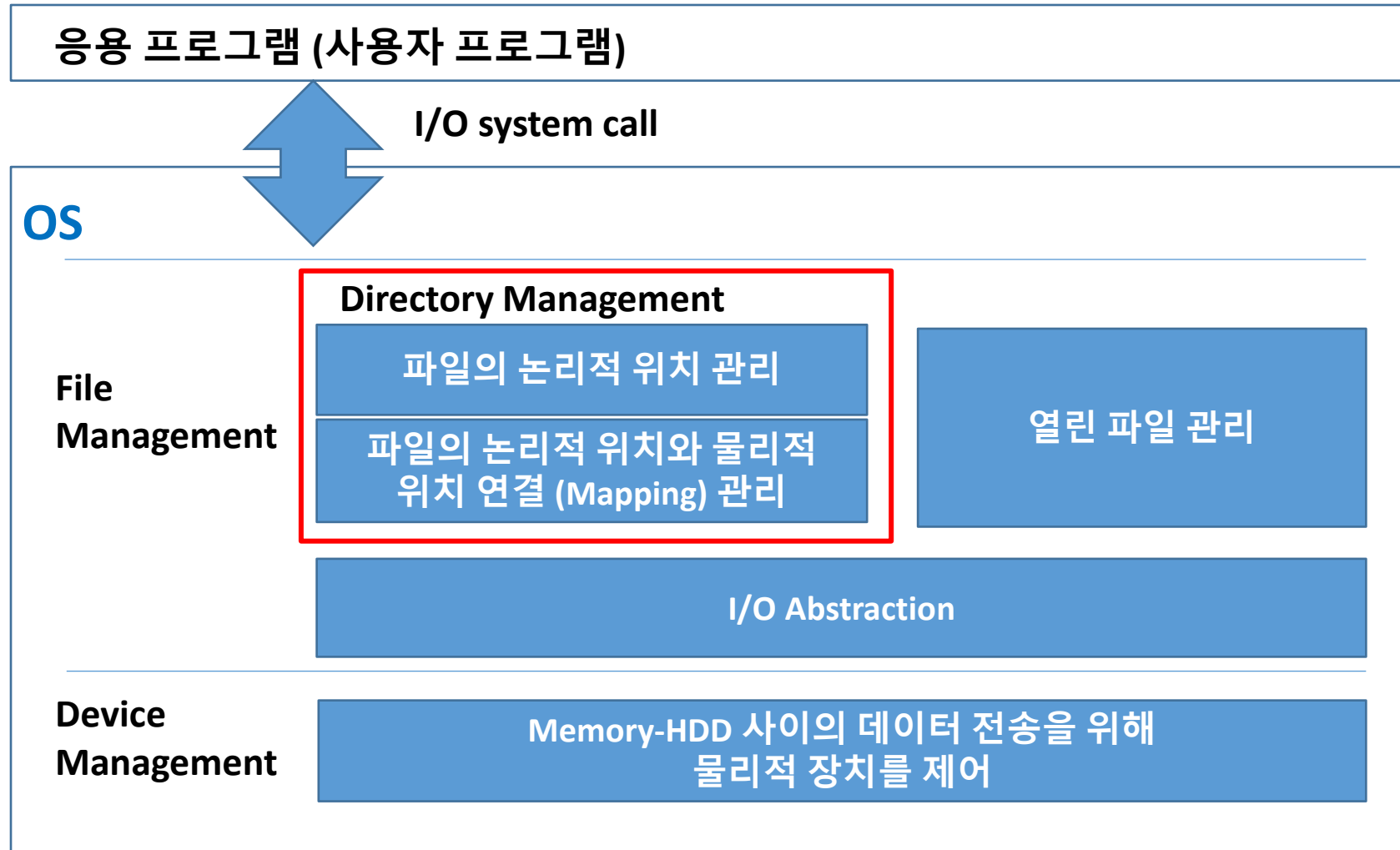
- **Device management (장치 관리)**
 - 물리적 저장장치에 대한 접근을 제공
 - 사용자의 논리적 관점에서의 I/O를 물리적 관점으로 변환하여 입출력 **투명성 (transparency)**을 제공

Summary - I/O Control System



3.2. Directory Management

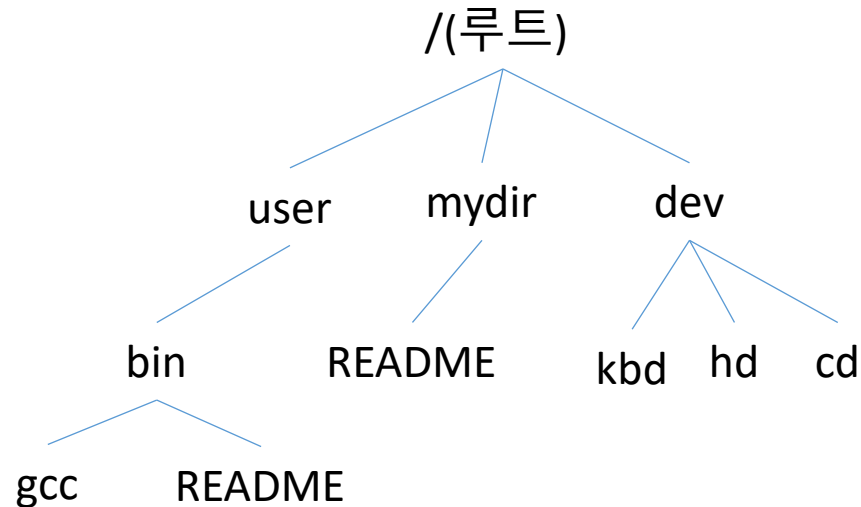
Summary - I/O Control System



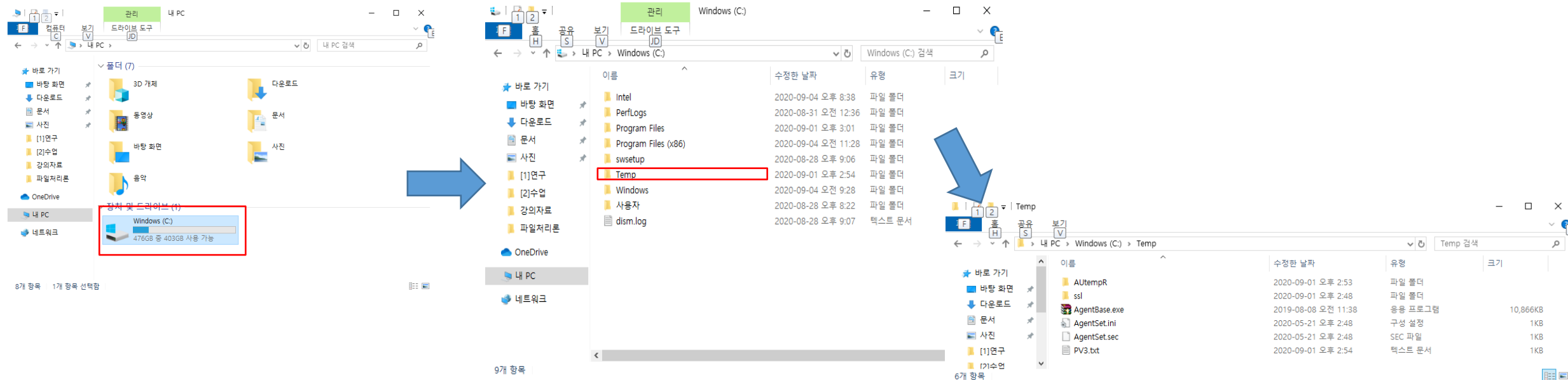
File Directory

- 시스템이 저장하고 있는 수 많은 File에 대한 편리한 접근을 위해
- File들을 Hierarchical Structure (계층 구조)로 관리하고
- 각각의 File에 대한 (접근에 필요한) 정보를 저장하는 File 조직 방법.

Hierarchical Structure (계층 구조)

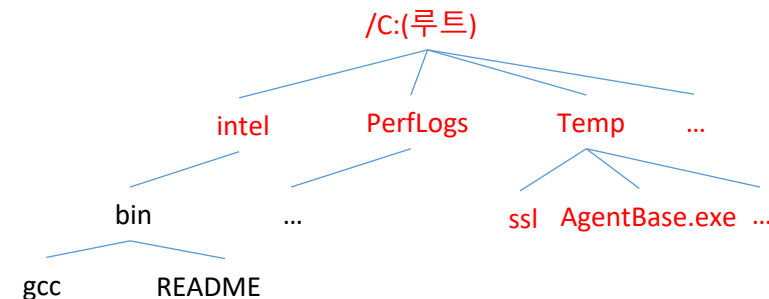


File Directory (Cont'd)



File Management System은 File Directory를 이용하여 시스템의 모든 File을 관리

- File의 이름, **저장 위치**, 크기, 타입 등의 정보를 File directory에 저장하여 관리
- **저장 위치**는 **논리적 위치** 뿐만 아니라 **실제 물리적 저장 위치도 포함** 함.
 - 이때문에 디렉터리가 파일과 파일 저장 장치를 연결시켜 준다 할 수 있음.



File Directory (Cont'd)

- **Directory structure**

- Symbol table 유지
 - Symbol name : 유저가 읽고 있는 이름.
 - OS 가 관리할 때는 Symbol name 이 아닌 다른 이름을 사용.
 - Symbol table : Symbol name <-> OS 관리 이름 간의 연결 (Mapping)
- Subdirectory도 포함
 - Directory 아래는 File이 아니고 Directory일 수 있다
- 한 레벨이나 여러 레벨의 계층 구조
 - File 식별을 위해 경로이름(pathname)을 사용
 - 루트에서부터 원하는 File에 이르는 디렉터리를 명세

File Directory (Cont'd)

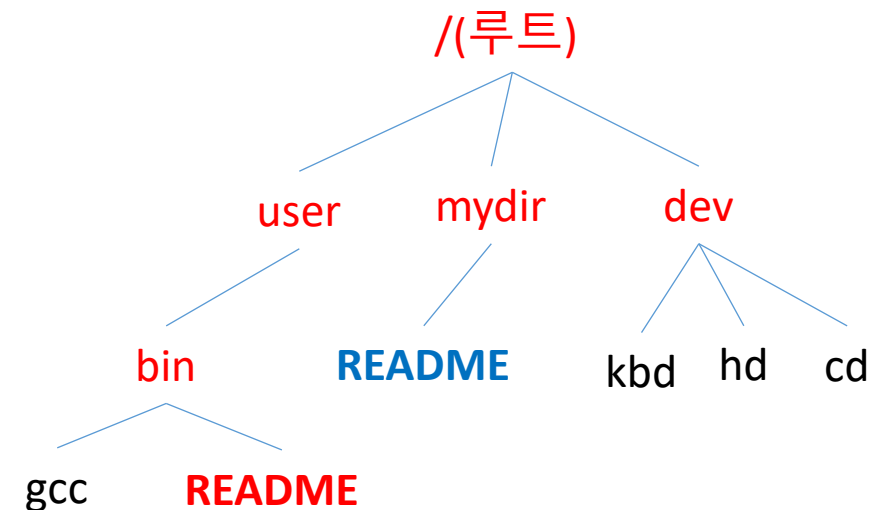
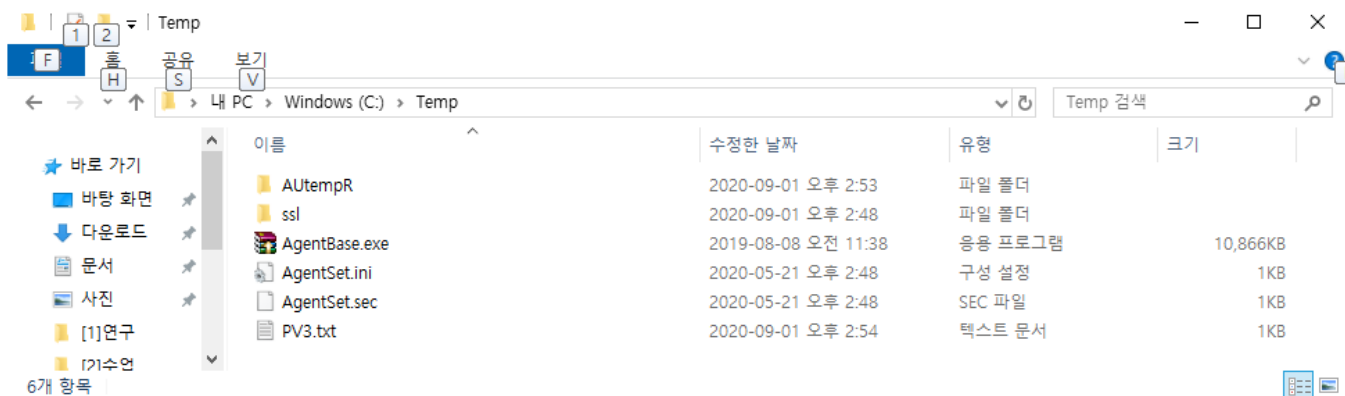
- **Directory structure**

- Symbol table 유지
 - Symbol name : 유저가 읽고 있는 이름.
 - OS 가 관리할 때는 Symbol name 이 아닌 다른 이름을 사용.
 - Symbol table : Symbol name <-> OS 관리 이름 간의 연결 (Mapping)
- Subdirectory도 포함
 - Directory 아래는 File이 아니고 Directory일 수 있다
- 한 레벨이나 여러 레벨의 계층 구조
 - File 식별을 위해 경로이름(pathname)을 사용
 - 루트에서부터 원하는 File에 이르는 디렉터리를 명세

File Directory (Cont'd)

• Directory structure

- Subdirectory도 포함
 - Directory 아래는 File이 아니고 Directory일 수 있다
- 한 레벨이나 여러 레벨의 계층 구조
 - File 식별을 위해 경로이름(pathname)을 사용
 - 루트에서부터 원하는 File에 이르는 디렉터리를 명세



Path to blue README : /mydir/README

Path to red README : /user/bin/README

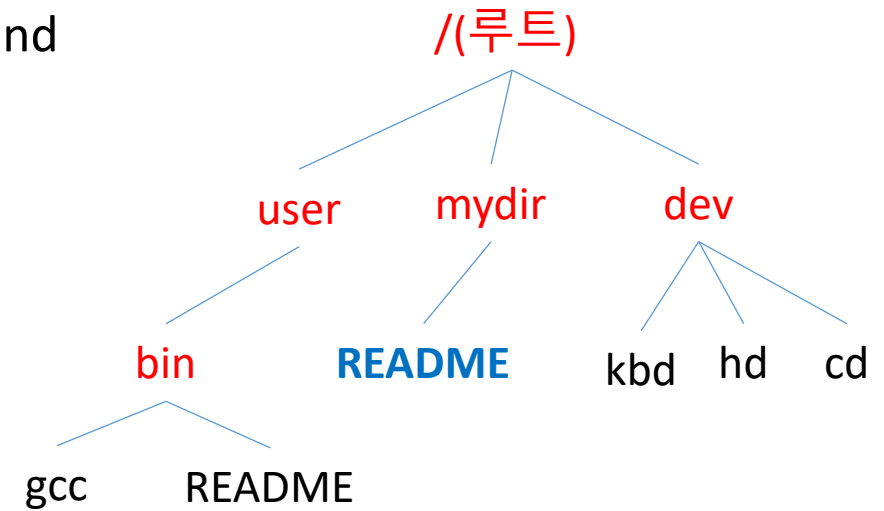
File Directory (Cont'd) - Directory를 이용한 연산

- **탐색 (search)**

- 특정 이름(symbolic name)의 파일을 찾기 위해 디렉터리 탐색
- cd (change directory) command + ls (list directory) command
- Ex) cd /user/bin, or cd user; ls, cd bin;

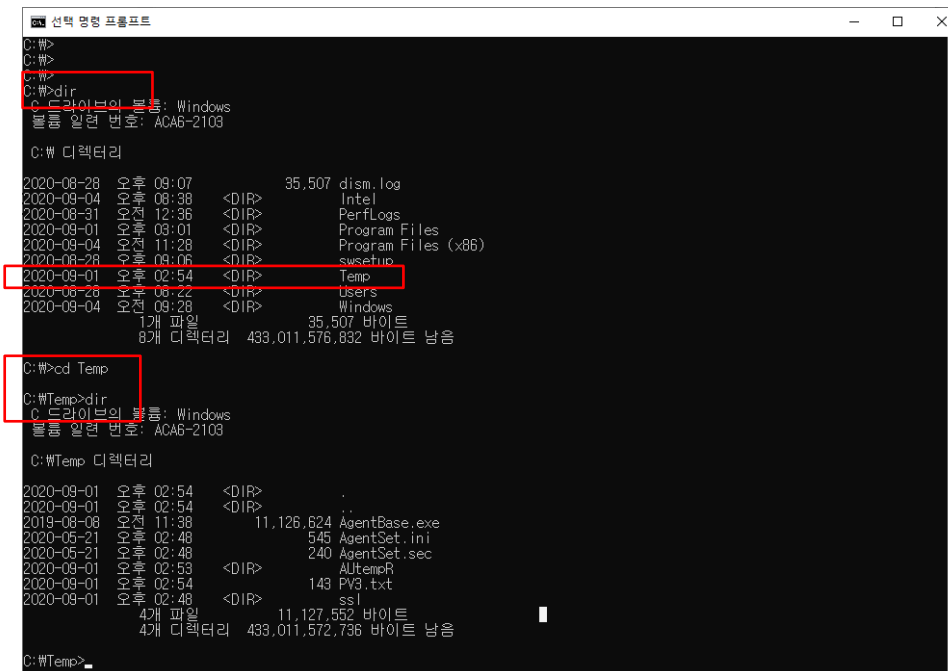
- **파일 생성(create file)**

- 파일을 디렉터리에 추가



File Directory (Cont'd) - Directory를 이용한 연산

- **Delete file (파일 삭제)**
 - Directory로부터 File 삭제
 - Ex) rm /user/bin/README
- **List directory (리스트 디렉터리)**
 - Directory 내용과 File에 대한 Directory Entry의 값을 표시
 - Ex) ls, ls -al, ls /user/bin.
 - Ex) dir



```
C:\>
C:\>
C:\> dir
C 드라이브의 볼륨: Windows
볼륨 일련 번호: AC46-2103

C:\> 디렉터리

2020-08-28 오후 09:07      35,507  <DIR>  dism.log
2020-09-04 오후 08:38      <DIR>  Intel
2020-08-31 오후 12:36      <DIR>  PerfLogs
2020-09-01 오후 03:01      <DIR>  Program Files
2020-09-04 오후 11:28      <DIR>  Program Files (x86)
2020-08-28 오후 09:06      <DIR>  swsetup
2020-09-01 오후 02:54      <DIR>  Temp
2020-08-26 오후 06:22      <DIR>  Users
2020-09-04 오전 09:28      <DIR>  Windows

1개 파일              35,507 바이트
8개 디렉터리      433,011,576,832 바이트 남음

C:\> cd Temp
C:\Temp> dir
C 드라이브의 볼륨: Windows
볼륨 일련 번호: AC46-2103

C:\Temp> 디렉터리

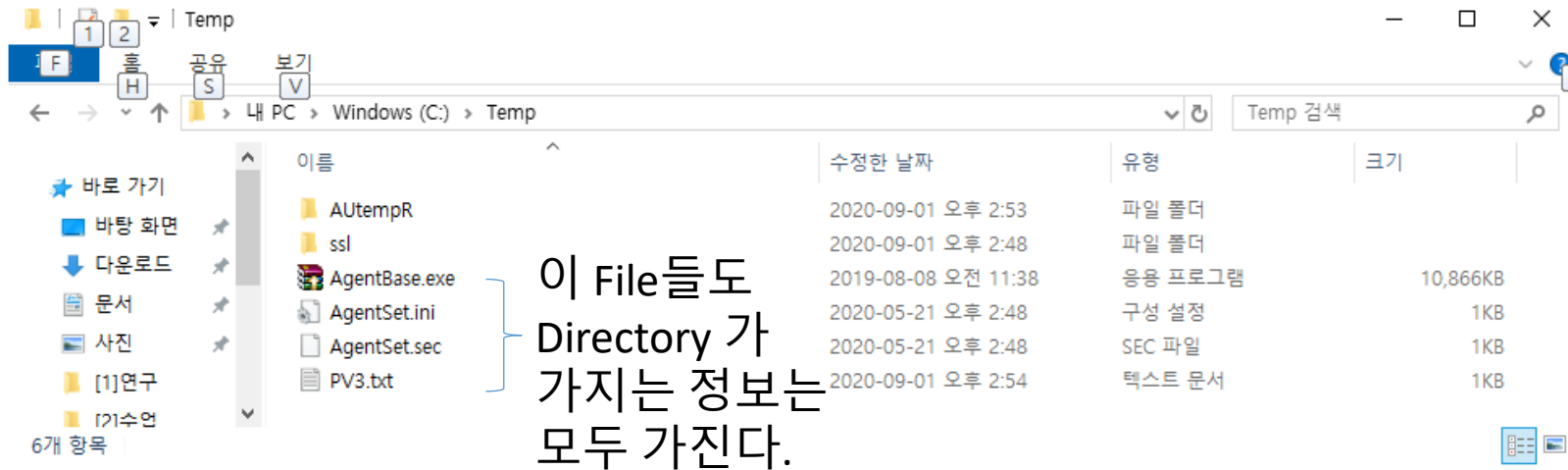
2020-09-01 오후 02:54      <DIR>  .
2020-09-01 오후 02:54      <DIR>  ..
2019-08-08 오전 11:38    11,126,624 AgentBase.exe
2020-05-21 오후 02:48      545 AgentSet.ini
2020-05-21 오후 02:48     240 AgentSet.sec
2020-09-01 오후 02:53      <DIR>  AutempR
2020-09-01 오후 02:54     143 Prg.txt
2020-09-01 오후 02:48      <DIR>  ssl

4개 파일              11,127,552 바이트
4개 디렉터리      433,011,572,736 바이트 남음

C:\Temp>
```

파일의 논리적 위치와 물리적 위치 연결 (Mapping)

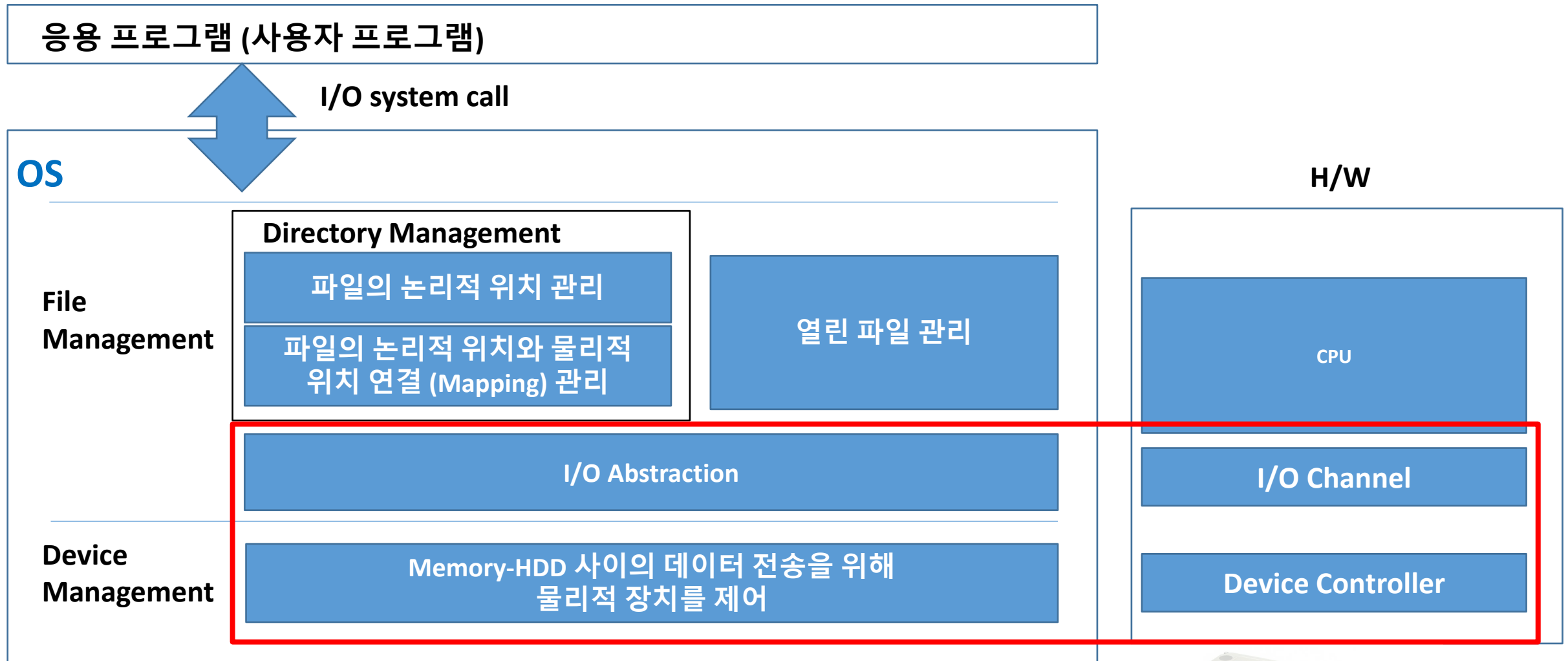
- Directory 에서 보이는 File도 Directory의 일부이다.



- 유저 눈에는 안보이지만 모든 Directory(File)은 실제 **물리적 장치의 어디에 데이터가 저장되어 있는지에 대한 정보도 저장**하고 있다.

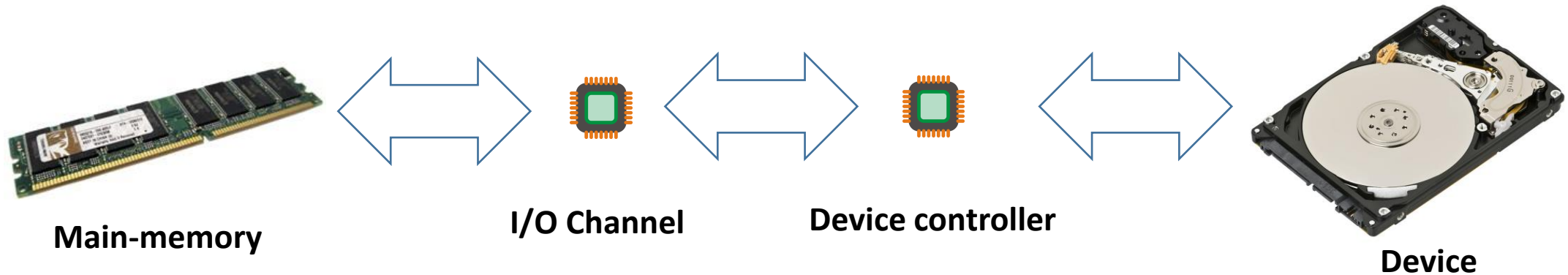
3.3. Device Management

I/O Device Control

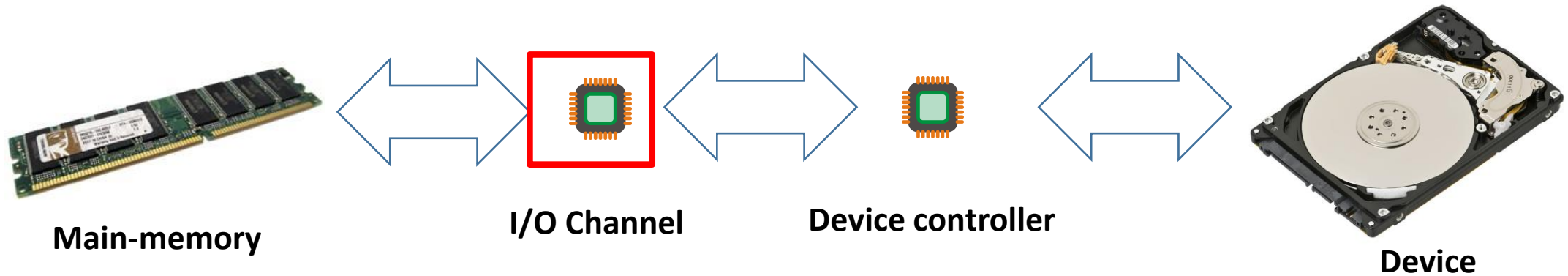


(물리적) I/O 경로

- 경로 :
 - Main Memory와 HDD (저장장치) 사이에 데이터를 전송하기 위해 (해 오려면) Main Memory 와 File을 저장하고 있는 HDD(저장장치) 사이에 거쳐야 할 장치들



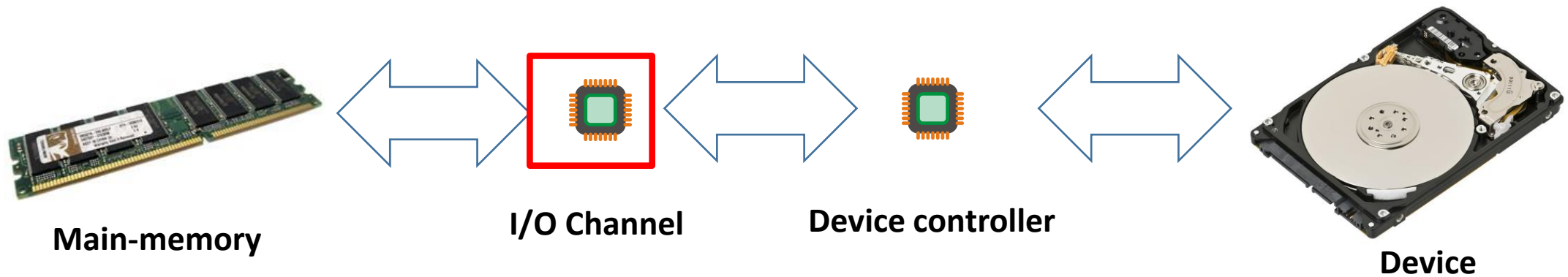
I/O channel



• I/O Channel

- CPU 명령(채널 프로그램)으로 작동하는 **입출력 처리기(I/O processor)**
 - I/O 장치를 제어하는 일종의 컴퓨터
 - 장치의 접근이나 데이터 경로 제어에 필요한 연산들을 수행
 - 어떤 연산을 수행할 지는 **채널 프로그램**에 정의됨.
 - 채널이 수행하는 프로그램
 - 장치의 접근이나 데이터 경로 제어에 필요한 연산들을 명세
 - OS에 포함되어 있음

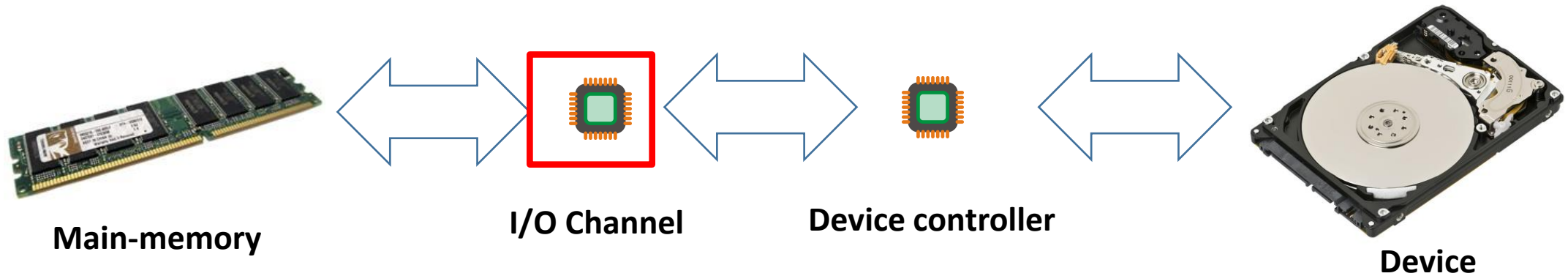
I/O channel (Cont'd)



• 채널 프로그램 예:

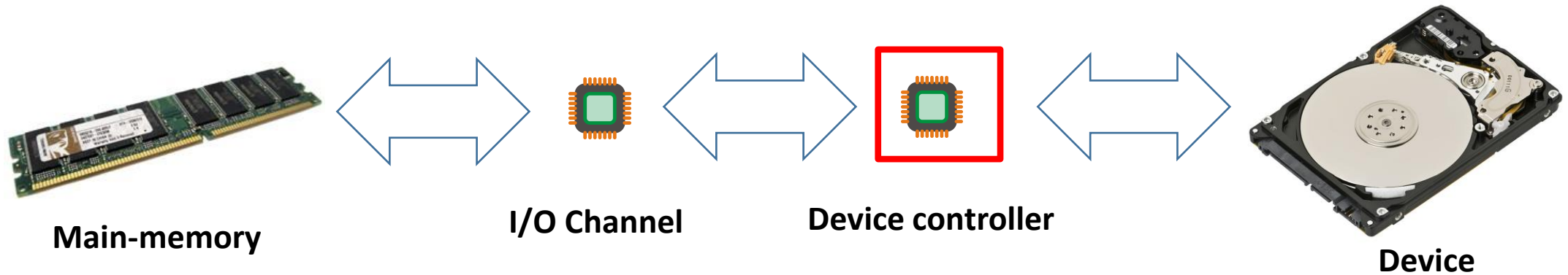
- 디스크에 대한 채널 프로그램 구성 명령어
 - Search : 요구하는 데이터를 디스크에서 탐색
 - Read : 레코드를 읽어서 메인 메모리 버퍼로 전송
 - Write : 메인 메모리 버퍼로부터 데이터를 디스크로 전송
 - Wait : 앞의 연산이 끝날 때까지 다음 read/write 명령어의 실행을 지연

I/O channel (Cont'd)



- 채널은 작업이 완료되면 인터럽트(interrupt)를 통해 CPU에 통보 (I/O interrupt)
 - 다른 작업 하고 있는 CPU를 잠시 불러 세워 긴급히 발생한 상황에 대한 처리를 시키는 루틴
 - I/O 인터럽트가 발생하면 OS는 I/O 처리 루틴으로 CPU 제어를 전달, I/O 작업이 완료 되었으므로, I/O작업에 대한 후속 작업을 처리하고 하던 일을 계속.

Device Controller (장치 제어기)



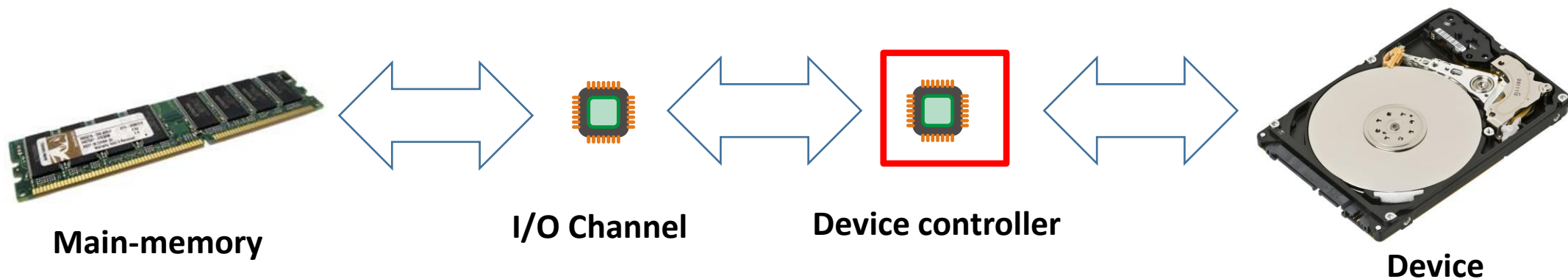
채널 명령어(search, read, write 등)를 해당 저장 장치에 적합한 연산으로 번역해서 입출력 실행을 제어 하는 장치

- I/O 채널이나 파일 관리자에게 상태 정보를 제공
장치 준비 여부, 데이터 전송 완료 등

- 호스트 컴퓨터와 장치 사이의 데이터 변환
호스트 : 비트들의 병렬 전송
예 : 8 byte를 한번에 보낸다. (10010001)
I/O 장치 : 비트들을 직렬 전송
예 : 1 bit 씩 보낸다. (1,0,0,1,0,0,0,1)

- 데이터 전송 시 에러 검사와 교정

Device Controller (장치 제어기)



채널 명령어(search, read, write 등)를 해당 저장 장치에 적합한 연산으로 번역해서 입출력 실행을 제어 하는 장치

- 데이터 전송 시 에러 검사와 교정

패리티 체크(parity check)와 복원

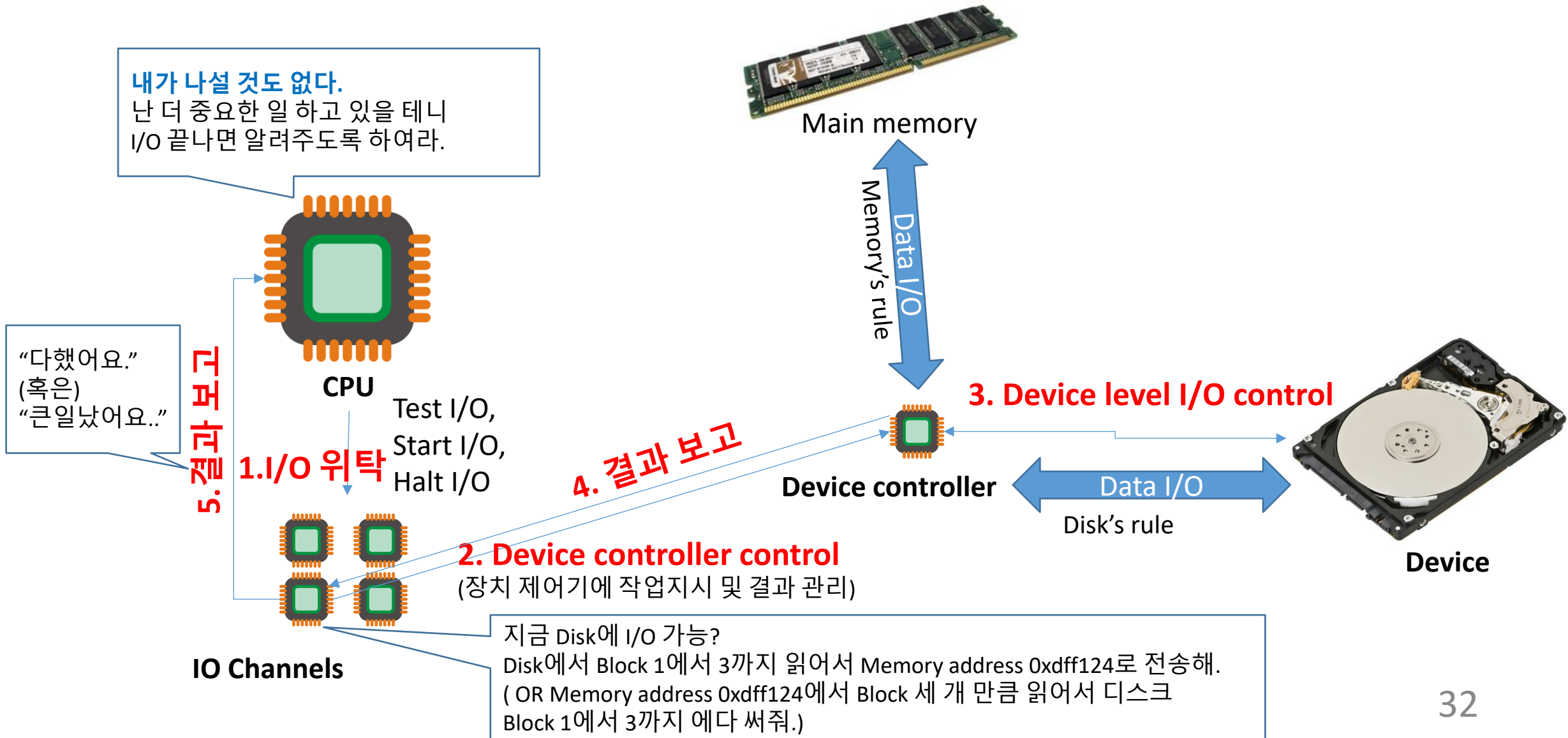
100?0001 에서 parity 1 (odd parity) => 100**1**0001

100?00?1 에서 parity 1 (odd parity) => 복원 불가

에러 교정을 위한 코드

CC(cyclic check characters), CRC(cyclic redundancy check characters),
ECC(error correction code) 등의 검사, 제거, 복원

Summary: I/O Channel and Device Controller (Disk Controller)

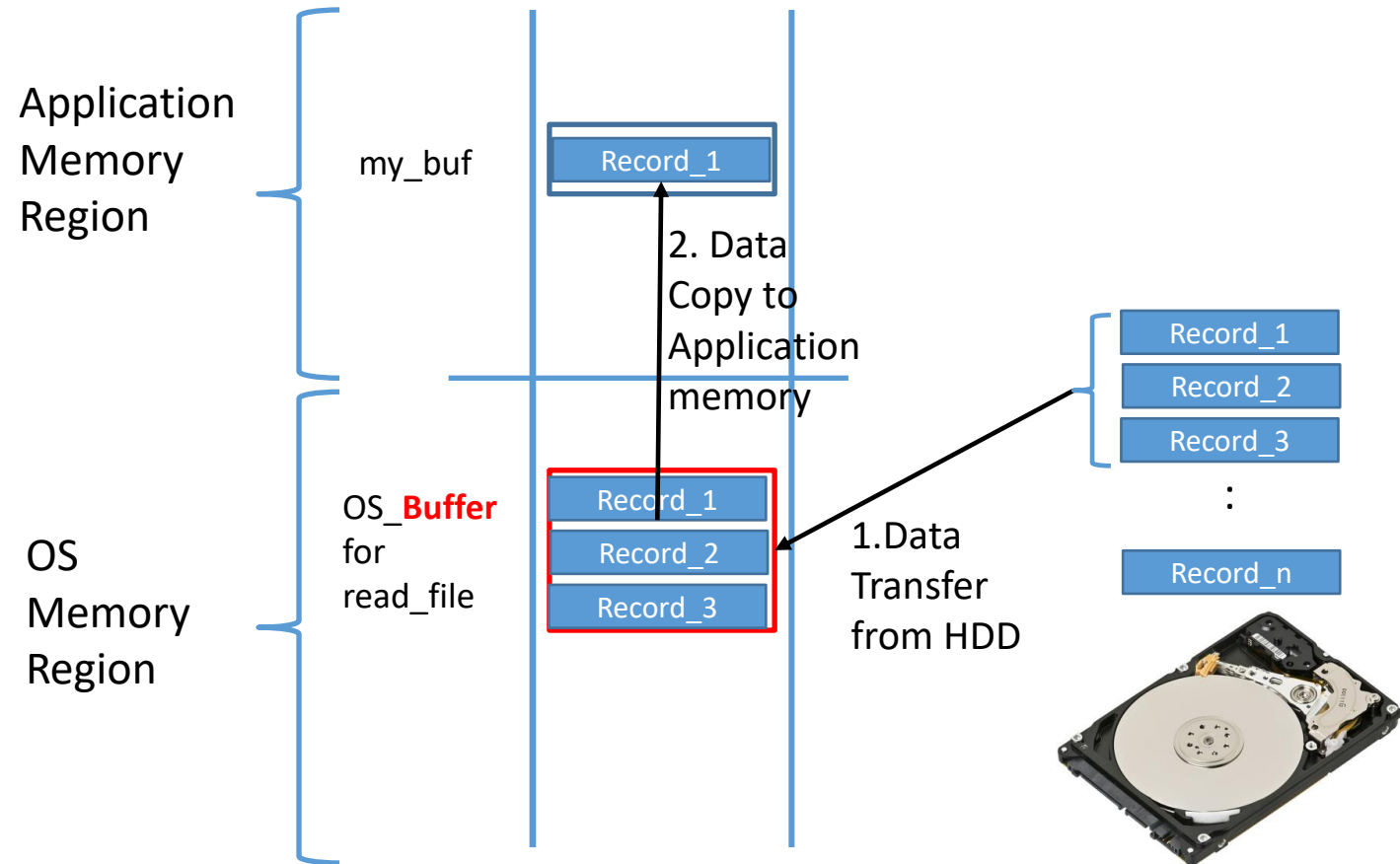


3.4. Buffer Management

Buffer : Disk에서 읽어온 Data를 임시 저장하기 위해 확보한 OS Memory.

Program A

```
int main() {
    ..
    ..
    ..
    char my_buf[RECORD_SIZE];
    read_file = fopen("input_a.txt", "r");
    fread(read_file, my_buf, RECORD_SIZE);
    buffer_print(my_buf)
    ...
}
```



Buffer (Cont'd)

Buffer

- Disk에 저장되어 있는 File에서 데이터를 읽어 들여 저장하는 Main memory의 일정 구역
 - Disk와 Memory의 Data 전송 속도 차이를 흡수하기 위한 공간.
 - 전송 속도 차이 흡수를 위해서 사용하는 공간은 모두 Buffer라 할 수 있으나, Disk(2차 저장장치) I/O와 관련 있는 OS의 Buffer만을 Buffer라 하자.
- Buffer 관리의 목적
 - 느린 I/O 저장장치를 최소한만 호출하여 I/O 처리 속도를 증가시키자.
 - 하나의 Buffer에 Record 여러 개를 저장 할 수 있도록, Buffer는 충분한 크기를 가져야 한다.

Buffer : Blocked I/O와의 차이점

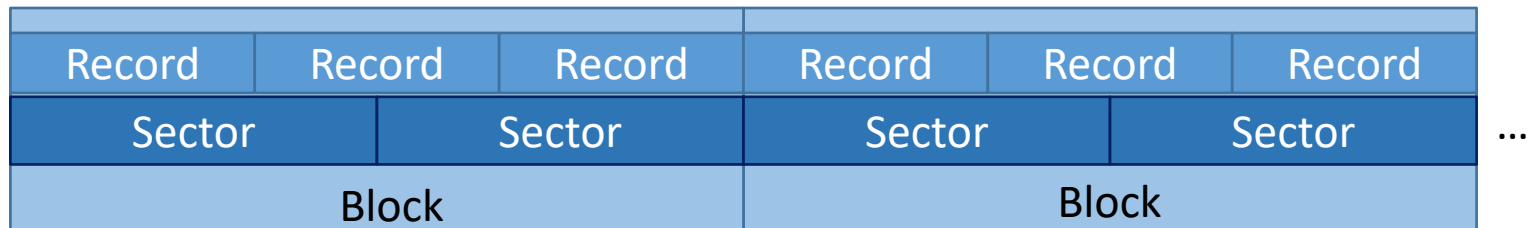
• 블록(block)

• 디스크와 메인 메모리 사이의 데이터 전송의 단위

- I/O 속도 향상을 위해 항상 “연속된 n개의 sector” 단위를 최소 I/O 단위로 한번에 읽고 쓴다.
- N은 OS에 따라 다르나, OS 운용 도중에 변하는 경우는 거의 없다.
 - 예) Data를 읽을 때는 $n=3$, 쓸 때는 $n=5$ 와 같이 사용하지 않고, 읽을 때 $n=3$ 이라면 쓸 때도 $n=3$ 이다.
- I/O의 최소 단위가 1개의 Block이다.

- Block은 연속된 record로 구성되어 있다고 볼 수 있음.

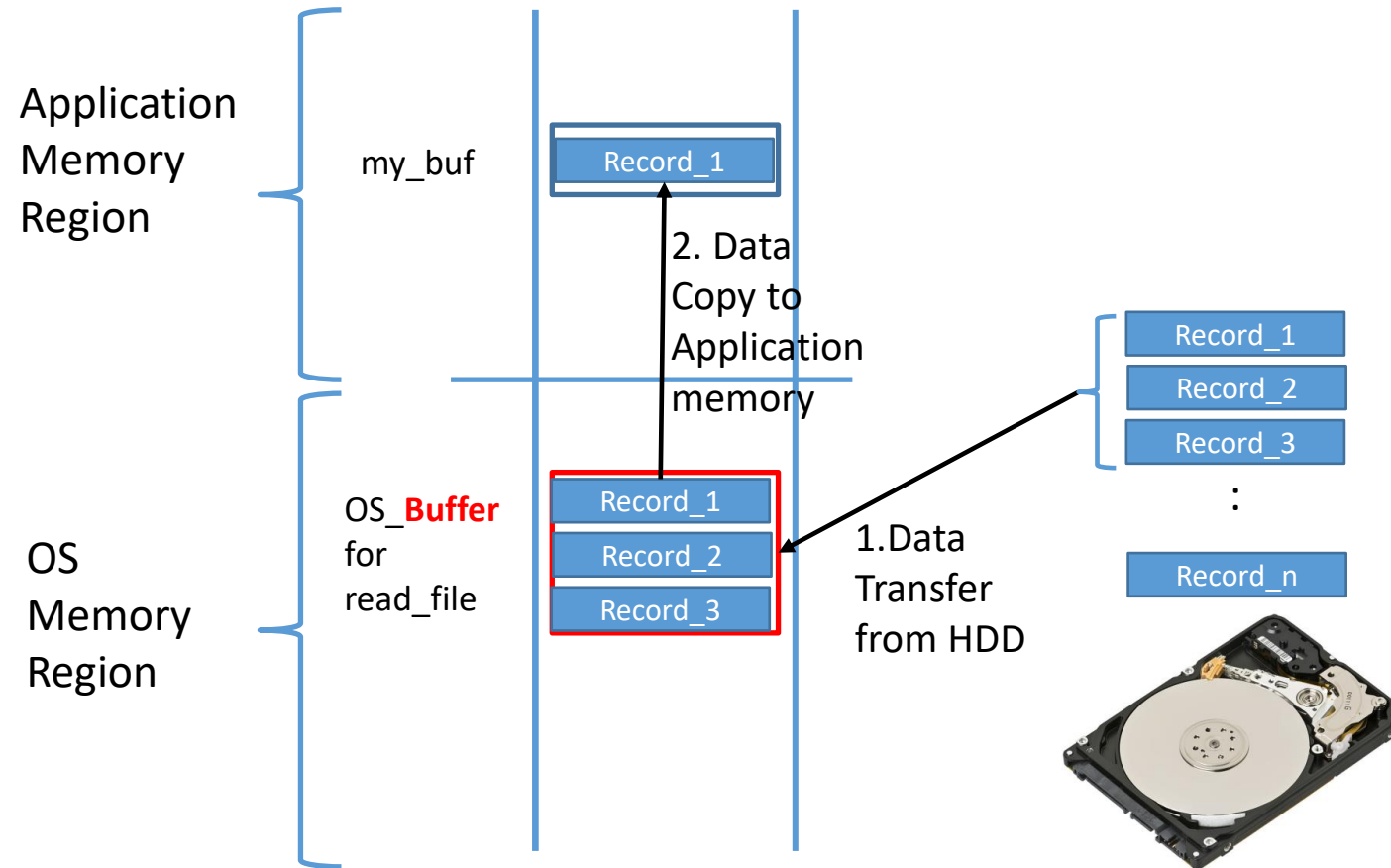
- 파일에 structured data가 저장됨.
- 일반적으로, File에 저장되는 record 하나하나가 순차적으로 Disk sector에 저장됨.
- Block이 연속된 sector로 구성되어 있다.



Buffer : Blocked I/O와의 차이점

- **Block I/O** : 데이터 전송의 형태를 정의
 - Block의 크기가 Main memory ⇔ Disk 사이의 최소 데이터 전송량 이다.
- **Buffer** : 전송해 온 데이터를 저장하는 Memory 공간을 의미
 - Block I/O로 Main memory 에 전송되어 온 Block 크기의 데이터가 Buffer에 저장된다.
 - Buffer의 최소 크기는 Block의 크기이다.
 - 일반적으로 Buffer의 크기는 Block의 크기.

Buffer : Blocked I/O와의 차이점 (Cont'd)



Buffer의 목적

- 목적: Disk와 Memory의 Data 전송 속도 차이를 흡수
- 어떻게 속도 차이가 흡수되는가?
 - 한번에 여러 개의 Record를 읽어와서 저장하기 때문에,
 - **미래에 읽을 데이터가 미리 Memory에 올라와 있다.**

Program A

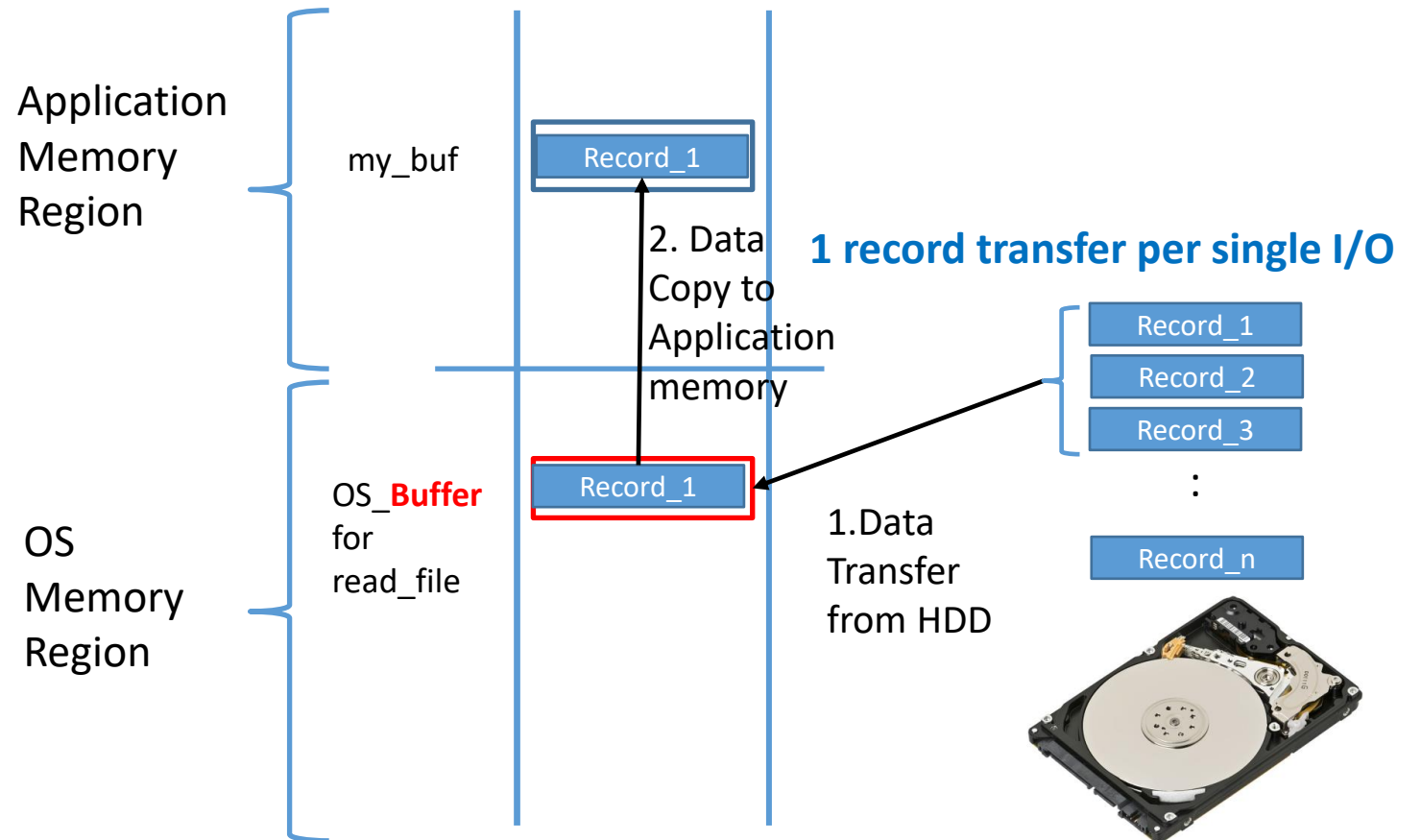
```
int main() {  
    ..  
    ..  
    ..  
    char my_buf[RECORD_SIZE];  
    read_file = fopen("input_a.txt", "r");  
    fread(read_file, my_buf, RECORD_SIZE);  
    fread(read_file, my_buf, RECORD_SIZE);  
    fread(read_file, my_buf, RECORD_SIZE);  
    fread(read_file, my_buf, RECORD_SIZE);  
    buffer_print("%s", my_buf)  
    ...  
}
```

Buffer의 목적 (Cont'd) : 예

- **Buffer size = Block size, Blocking factor = 1**에서 왼쪽 아래 Code 실행 => 4 Disk IO

Program A

```
int main() {
    ..
    ..
    ..
    char my_buf[RECORD_SIZE];
    read_file = fopen("input_a.txt", "r");
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    buffer_print("%s", my_buf)
    ...
}
```

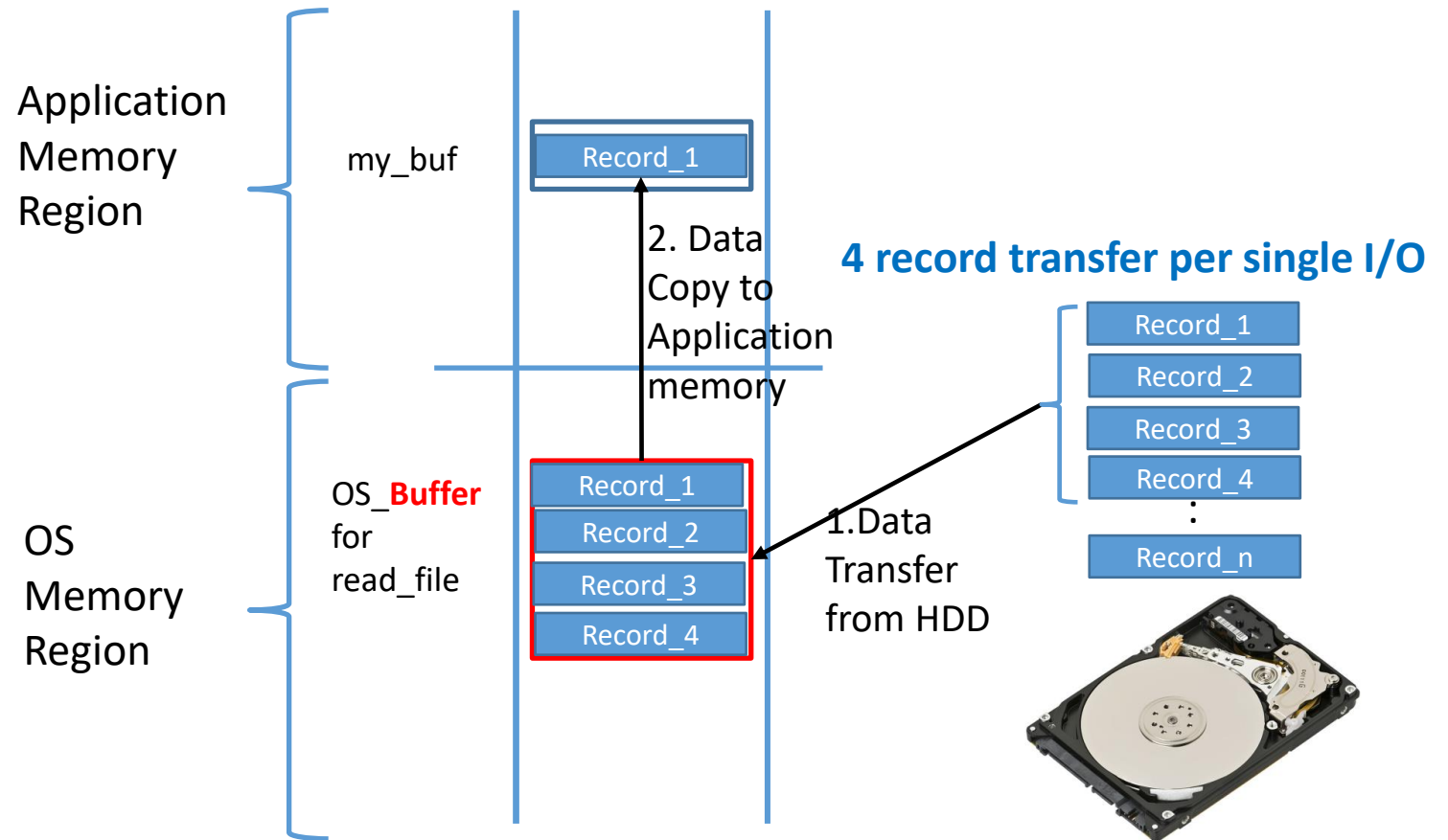


Buffer의 목적 (Cont'd) : 예

- **Buffer size = Block size, Blocking factor = 4**에서 왼쪽 아래 Code 실행 => 1 Disk IO

Program A

```
int main() {
    ..
    ..
    ..
    char my_buf[RECORD_SIZE];
    read_file = fopen("input_a.txt", "r");
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    fread(read_file, my_buf, RECORD_SIZE);
    buffer_print("%s", my_buf)
    ...
}
```



Buffer Management

- **Buffer Manager (버퍼 관리자)**
 - OS 기능의 일부
 - 제한된 크기인 Main memory에서 Buffer를 위한 공간을 최적으로 분배
 - 응용 프로그램의 요구에 따라 Buffer 공간 할당
 - 할당된 버퍼 중에서 사용하지 않는 Buffer 공간을 관리
 - Buffer 요구량이 할당 가능 공간을 초과시
 - 응용 프로그램을 지연
 - 우선 순위가 낮은(또는 사용도가 낮은) 프로그램에 할당된 버퍼 공간을 회수

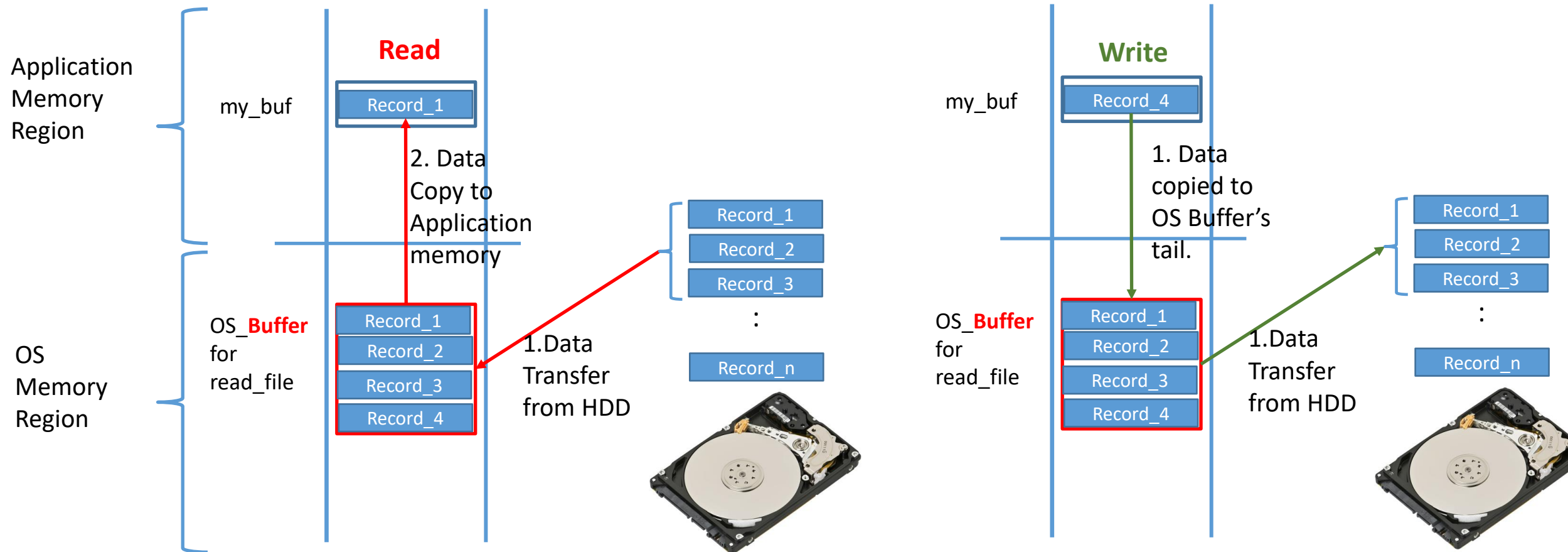
Simple Buffer System (단순 버퍼 시스템)

- **Simple Buffer(단순 버퍼)**

- 하나의 파일 I/O를 위해 Memory에 하나의 Buffer만 할당
- 응용 프로그램이 File 데이터를 읽을 때 Buffer 로 데이터 Block을 읽어 들임
- 응용 프로그램이 File에 데이터를 쓸 때 Buffer에 저장된 Block을 Disk에 저장.

Buffer 채우기와 비우기

- 읽기와 쓰기에 따라 채우는 쪽과 비우는 쪽이 달라진다.
 - 버퍼를 채우는 쪽을 생산자(producer), 비우는 쪽을 소비자(consumer)로 부른다.



Anticipatory Buffering (예상 버퍼링)

- Buffer가 채워질 때까지 Application이 유휴 상태(idle)로 되는 문제가 발생
- OS (혹은 File Manager)가 프로그램이 필요로 할 것으로 예측되는 데이터로 미리 Buffer를 가득 채워 놓음 (prefetching)
 - 프로그램이 필요로 할 것으로 예측되는 데이터가 무엇인지 알아내는 방법은 수업 범위 밖.
 - 일단 그런 방법이 있고 사용할 수 있다고 가정하자.
- 적용하기 적합한 Application Program Procedure

Program A

```
int main() {  
    ..  
    char my_buf[RECORD_SIZE];  
    read_file = fopen("input_a.txt", "r");  
    많은 CPU 연산 수행  
    for(int i = 0; i < 10; i++) {  
        fread(read_file, my_buf, RECORD_SIZE); fread(read_file, my_buf, RECORD_SIZE);  
        fread(read_file, my_buf, RECORD_SIZE); fread(read_file, my_buf, RECORD_SIZE);  
        읽은 데이터로 많은 CPU 연산 수행 (<= 이 동안 I/O Device는 할 일이 없음, 미리 읽을 수 있다면 읽자)  
    }  
    ...  
}
```

Full-Flag: Anticipatory Buffering 을 위해 필요한 구조

- Full-Flag
 - 예상 Buffering을 위한 Buffer 구조.
 - 버퍼가 채워졌는지를 **표시하는 플래그(full-flag)**
 - Full-flag가 1 :
 - Buffer에 아직 사용하지 않은 데이터가 있으므로 Buffer내용을 지우고 새로 데이터를 적을 수 없음.
 - AND
 - Buffer에서 내용을 읽을 수 있음.
 - Full-flag가 0 :
 - Buffer의 Data를 모두 사용하여서 Buffer 내용을 지우고 **새로 데이터를 적을 수 있음.**
 - OR
 - Buffer에 내용을 채워넣는 중이므로 Buffer의 내용을 읽을 수 없음.

Anticipatory Buffering 생산자 프로그램 구조

- 생산자(Producer) 루틴 :

- 버퍼 빌 때까지 기다렸다가 비면 버퍼 채우고 버퍼에 다 찼음 플래그를 (full_flag) 1로 만든다.

loop : **if (full_flag = 1) goto loop; //버퍼가 차 있으면 다 비워질 때 까지 대기**

issue start-I/O command to disk-controller; //디스크 제어기에 I/O시작 명령을 내린다.

wait while buffer is being filled; //버퍼가 채워지는 동안 대기

full_flag = 1; // 버퍼 내가 채워 났다. 읽어도 된다.

goto loop;

※ 초기에 full_flag=0으로하고 I/O 채널이 버퍼를 채우기 시작



Anticipatory Buffering 소비자 프로그램 구조

- 소비자(Consumer) 루틴

- 버퍼 찰 때까지 기다렸다가 차면 버퍼 비우고 버퍼 다 찼음 플래그를 (full_flag) 0으로 만든다.

wait : **if (full_flag = 0) goto wait; //버퍼가 비워져 있으면 찰 때까지 대기**

read buffer into work area; //버퍼에 있는 레코드를 작업 구역으로 이동 (버퍼 비우기)

full_flag = 0; // 버퍼 내가 다 읽어서 비었다.

goto wait;



Double Buffer System : Application I/O Wait Time을 더 줄여 보자.

• Simple Buffer System의 문제점

- 다음 두 pseudo code에 대해 Anticipatory Buffering 으로 Simple Buffer를 채우고 소비하는 경우를 생각해 보자.
- 기다리는가/기다리지 않는가?
- 왜?

```
For() {  
    fread(record 1개 읽기);  
    일 적당히  
    fread(record 1개 읽기);  
    일 적당히  
    fread(record 1개 읽기);  
    일 적당히  
    fread(record 1개 읽기);  
    일 적당히  
}
```

```
For() {  
    fread(record 1개 읽기);  
    fread(record 1개 읽기);  
    fread(record 1개 읽기);  
    fread(record 1개 읽기);  
    일 많이 (“일 적당히” 4 배 이상)  
}
```

* 이 부분의 설명은 Buffer에 4개의 Record가 저장됨을 전제로 한다.

Double buffer system

- 파일당 두 개의 버퍼를 할당하여, 비우는(읽는) Buffer와 채우는(쓰는) Buffer를 서로 다른 Buffer를 사용.
 - 소비자가 하나의 버퍼를 비우는 동안 생산자는 다른 버퍼를 채움
 - 생산 연산과 소비 연산이 순환,반복되면서 병행적으로 수행
- 이중 버퍼 시스템에서 기대 하는 것 : 생산자 및 소비자의 기다리는 시간 최소화

Double buffer system (Cont'd)

Buffer 1

Buffer 2

Step 0. Buffer 1에 Disk에서 읽어온 Data를 채워 넣는다.

Step 1. App. 이 Buffer 1에서 데이터를 읽을 때, Buffer 2에 Disk에서 읽어온 Data를 채워 넣는다.

Step 2. (이상적이라면) App. 이 Buffer 1에서 데이터를 다 읽고 이들을 사용한 연산을 다 끝냈을 즈음에, Buffer 2에 Disk에서 읽어온 Data를 채워 넣는 작업이 끝난다.

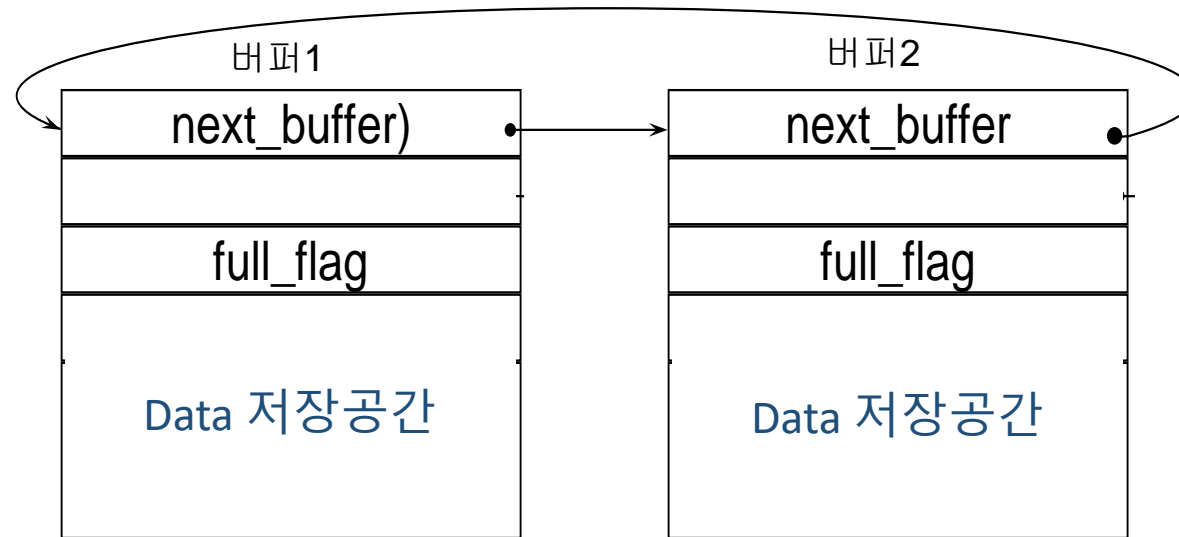
Step 3. 비우는 버퍼 채우는 버퍼의 역할을 바꾸어서,
App. 이 Buffer 2에서 데이터를 읽을 때, Buffer 1에 Disk에서 읽어온 Data를 채워 넣는다.

Step 4. (이상적이라면) App. 이 Buffer 2에서 데이터를 다 읽고 이들을 사용한 연산을 다 끝냈을 즈음에, Buffer 1에 Disk에서 읽어온 Data를 채워 넣는 작업이 끝난다.

Step 5. Step 1로 돌아가 반복

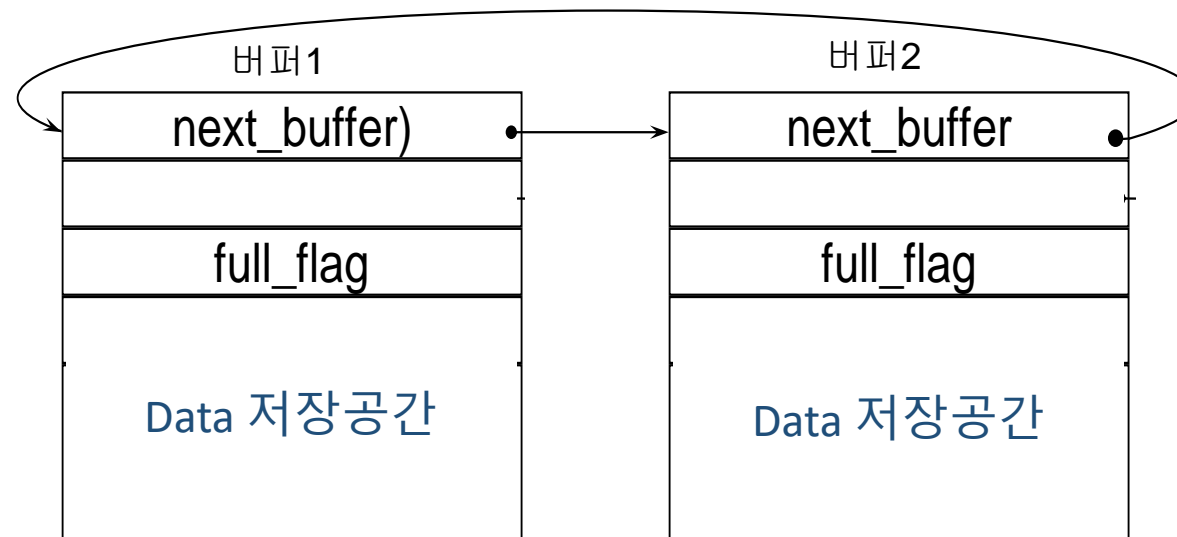
Double buffer system 구조

- Double buffer system 구조



Double buffer system 에서의 생산자/소비자

- 2 개의 포인터를 추가로 사용
 - to_fill : 현재 채워지고 있거나 다음에 채워야 할 버퍼에 대한 포인터. 생산자용.
 - to_empty : 현재 비워지고 있거나 다음에 비워져야 할 버퍼에 대한 포인터. 소비자용.
- 생산자는 항상 to_fill이 가리키는 버퍼를 채움
 - (초기에는 버퍼1을 지시)
- 초기
 - 두 버퍼가 모두 공백 (full_flag는 모두 0)
 - 생산자의 to_fill, 소비자의 to_empty 모두 버퍼 1 을 가리킴.



Double buffer system 에서의 소비자/생산자

- 생산자

- 1. 항상 to_fill이 가리키는 버퍼가 비기를 기다렸다가 비면 데이터를 채움
- 2. 다 채우면 현재 버퍼가 아닌 다른 버퍼를 to_fill 로 설정하고 1부터 반복

```
loop : if (to_fill.full_flag = 1) goto loop; //to_fill.buffer가 비워질 때 까지 대기
        디스크에서 데이터 읽어 to_fill buffer로 전송하는 명령을 내린다.
        wait while to_fill.buffer is being filled; // to_fill.buffer가 채워질 때까지 대기
        to_fill.full_flag = 1;
        to_fill = to_fill.next_buffer; // to_fill은 다음에 채워져야 할 버퍼를 지시
        goto loop;
```

Double buffer system 에서의 소비자/생산자

- 소비자

- 1. 항상 to_empty이 가리키는 버퍼가 차기를 기다렸다가 차면 데이터를 비움
- 2. 다 비우면 현재 버퍼가 아닌 다른 버퍼를 to_empty 로 설정하고 1부터 반복

wait : if (to_empty.full_flag = 0) goto wait; //to_empty.buffer가 채워질 때 까지 대기
Application이 필요로 하는 Record를 Application의 Memory로 복사.
모든 Record를 Application이 사용한 경우.
to_empty.full_flag = 0; // 현재 버퍼 비었다고 표시
to_empty = to_empty.next; //to_empty는 다음에 비워야 할 버퍼를 지시
goto wait;

3.5. Unix 에서의 I/O

Unix 파일 I/O

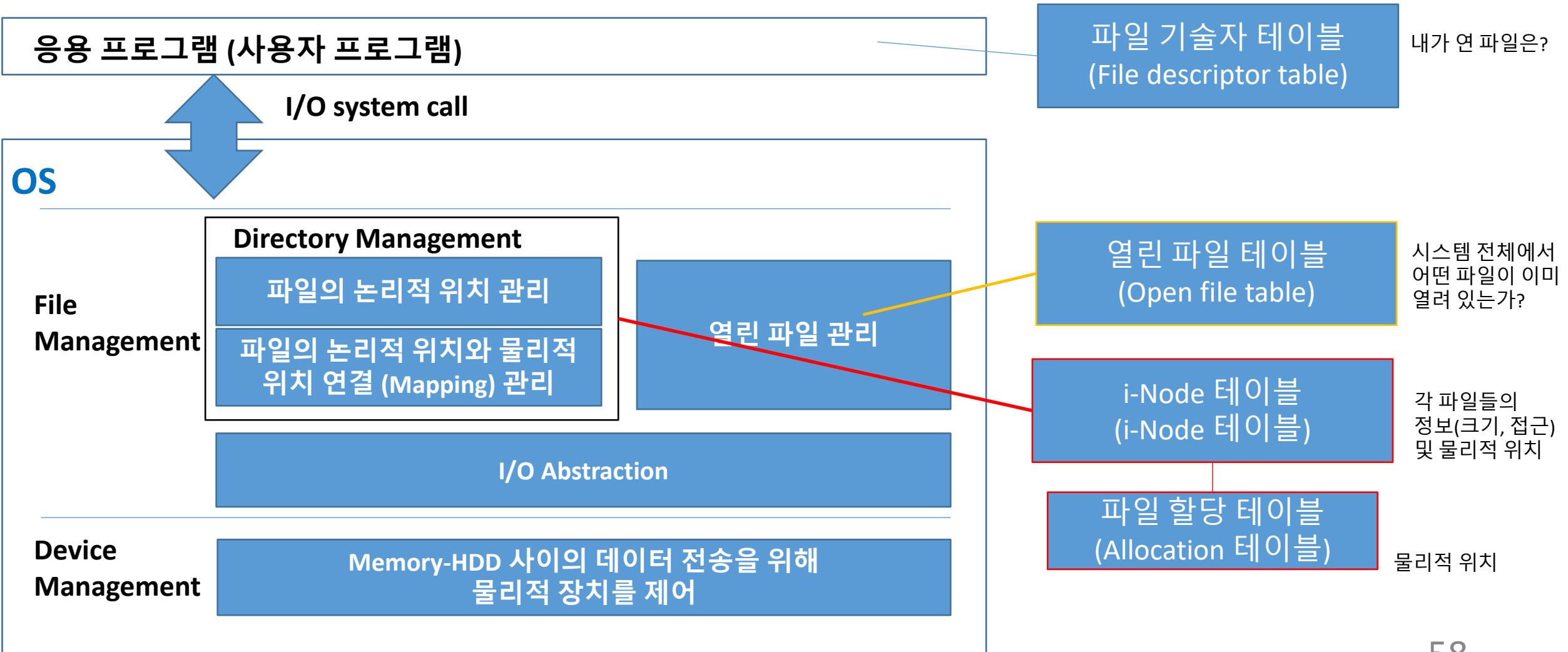
- **UNIX 란?**

- 1970년대 초반 벨 연구소 직원인 켄 톰슨, 데니스 리치 등이 처음 개발.
- 다중 사용자 방식의 멀티 태스킹 운영체제
- 이후 등장한 LINUX, iOS 에 많은 영향을 줌
- 판매하는 UNIX 예: Solaris

- **UNIX 파일 I/O 예제 시나리오 : 데이터 저장할 파일의 물리적 위치 찾기**

- 응용 프로그램이 파일에 데이터를 기록하라는 명령문 `fwrite (fd, "write_this", 10)`을 실행.
- 시스템 호출 인터페이스(system call interface)를 통해 커널이 기동.
 - System call interface: 프로세스가 커널과 직접 통신하도록 해주는 루틴.
 - 쉽게 말하면 커널 불러서 일 부탁하는 함수.
- 이 후 커널은 어떻게 저장된 파일을 찾을까?

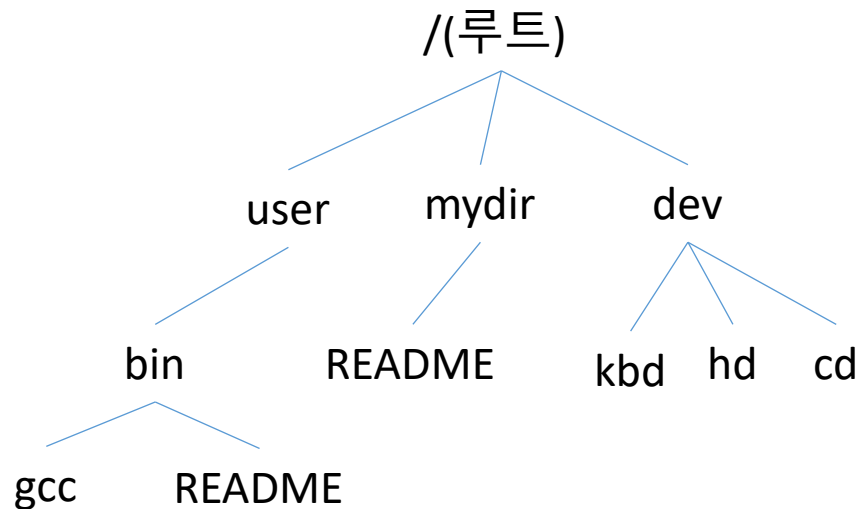
UNIX 파일 I/O : 저장된 물리적 파일의 위치 찾기



파일 이름과 디스크 파일의 연결

• 디렉터리 구조

- 파일 이름과 디스크에 저장 되어있는 그 파일의 inode에 대한 포인터로 구성
- Inode에 대한 포인터는 그 파일 이름으로부터 그 파일에 대한 모든 정보에 대한 직접 참조를 제공
- 파일이 열리면 해당 파일의 inode를 메모리(inode 테이블)로 가져오고 열린 파일 테이블에 해당 엔트리를 추가



파일 이름	inode 포인터
...	...

Unix 디렉터리 파일

Unix I/O를 위한 테이블

1. 파일 기술자 테이블(file descriptor table)

- 각 프로세스가 열어 놓은 파일을 기록하는 테이블
- 프로세스당 하나

2. 열린 파일 테이블 (open file table)

- 현재 시스템이 열어 사용중인 모든 파일에 대한 엔트리로 구성
 - 엔트리 : 읽기/쓰기 형식, 사용 프로세스 수, 다음 읽기/쓰기 연산을 위한 파일 오프셋, 이 파일 작업에 사용할 수 있는 일반 함수들의 포인터
- Unix 시스템 전체에 하나

Unix I/O를 위한 테이블

3. 인덱스 노드 테이블(index node table)

- 현재 사용되고 있는 **파일 당 하나**의 엔트리(inode)로 구성
- 각 엔트리는 파일과 함께 디스크에 저장되어 있는 inode(index node)의 사본
- inode에는 파일의 저장 위치 (파일 할당 테이블), 크기, 소유자 등 파일 접근에 필요한 정보가 저장

4. 파일 할당 테이블(file allocation table)

- 실제로는 인덱스 노드(index node) 구조의 일부
- 파일에 할당된 디스크 블록 리스트를 포함

파일 기술자 테이블과 열린 파일 테이블

파일
기술자 테이블
(프로세스당 하나)

파일 기술자	열린 파일 테이블 엔트리
0(키보드)	
1(화면)	
2(에러)	
3(일반화일)	
4(일반화일)	
...	

열린 파일 테이블
(UNIX 전체에 하나)

R/W 모드	파일사용 프로세스수	다음접근 오프셋	Write 루틴 포인터	...	inode 테이블 엔트리
...
write	1	100
...



Inode 테이블 구조

inode 테이블

소유자 ID
장치
그룹 이름
파일 유형
접근 권한
파일 접근 시간
파일 참조 포인터 수
파일 크기 (블록수)
블록 카운트
파일 할당 테이블

파일 할당 테이블

데이터 블록번호 0
데이터 블록번호 1
...
데이터 블록번호 9