

8장. File

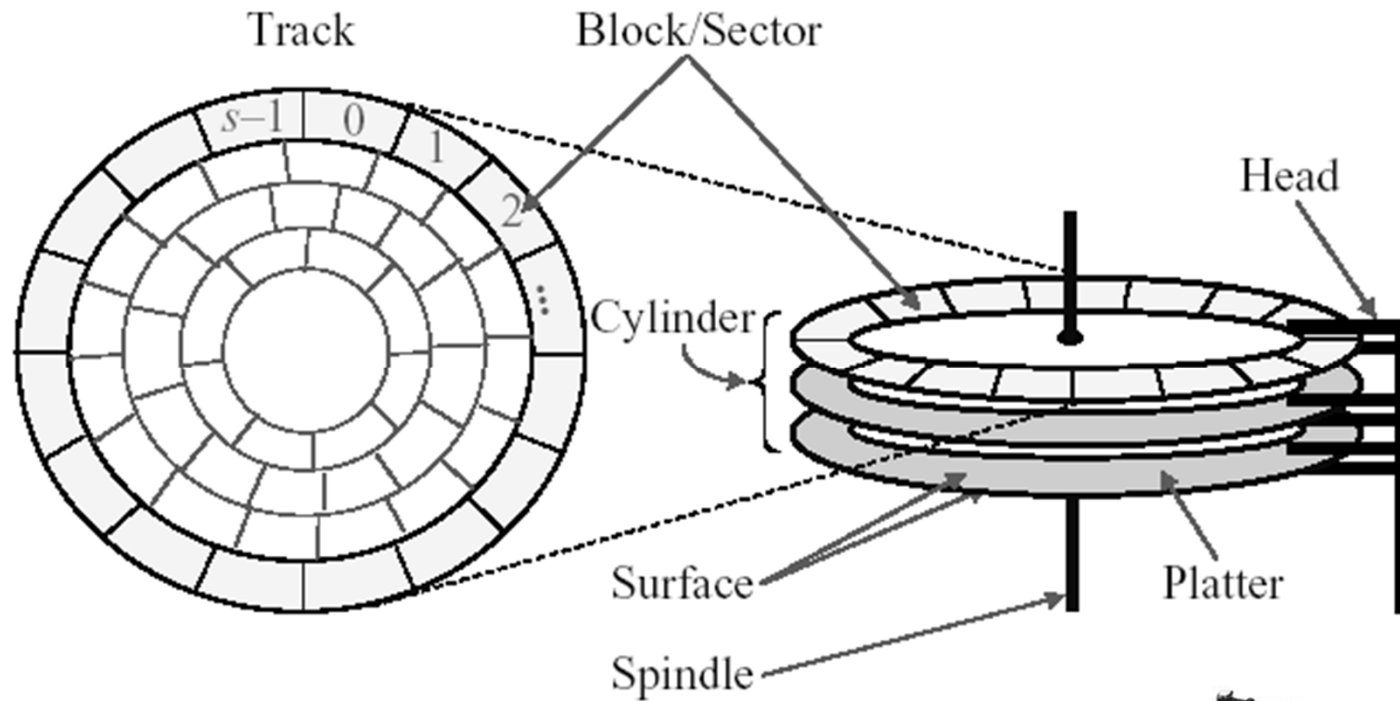
목 차

- Disk 구조
- File system 구조
- inode(index node)
- Virtual File System
- Kernel data structures for open files
- UNIX file access primitives
- Standard I/O Libraries
- File System의 연결(mount)

File 관련 issue

- 파일 관련 issue
 - 사용자가 사용하는 이름과 disk 상의 데이터를 연결하는 객체
 - 파일 속성 제어
 - 파일 접근 제어
 - 파일 계층 구조 지원
 - 장치 파일 지원
- Linux file access primitives
 - open, creat : 파일 열기 or 생성
 - close : 파일 닫기
 - read : 파일에서 정보 추출
 - write : 파일에 정보 기록
 - lseek : 파일 접근 위치 변경
 - unlink, remove : 파일 삭제
 - fcntl : 파일 속성 제어
 - ...

Disk의 물리적 구성 요소

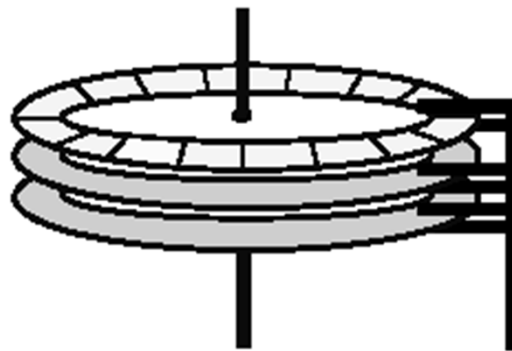


- ✓ Head, ARM
- ✓ Platter, Spindle
- ✓ Surface, Track, Sector, Cylinder



Disk의 물리적 구성 요소

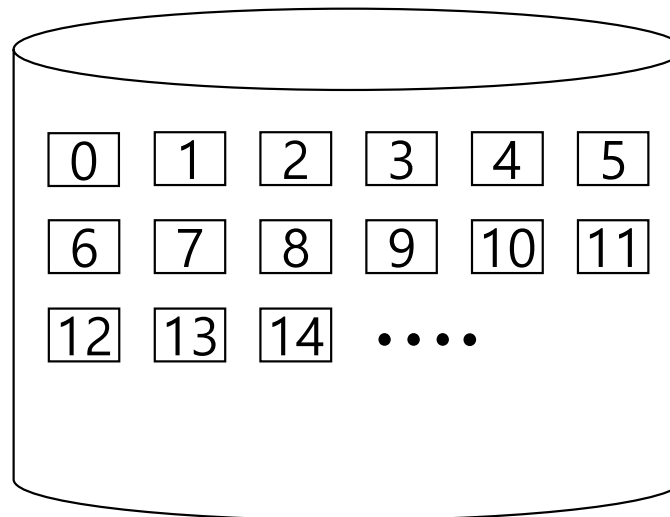
- 데이터 접근 방법
 - Sector addressing : surface(head), track(cylinder), sector
 - Heads move to appropriate track : 탐색 시간 (Seek time)
 - Wait for the sector to appear under the head : 회전 지연 시간 (rotational latency)
 - Read/Write the sector: 데이터 전달 시간 (transmission time)



- Seek time과 Rotational Latency를 줄이는 것이 중요
 - ➔ 다양한 디스크 스케줄링 방법 존재, parallel access 이용

Disk의 논리적 구조

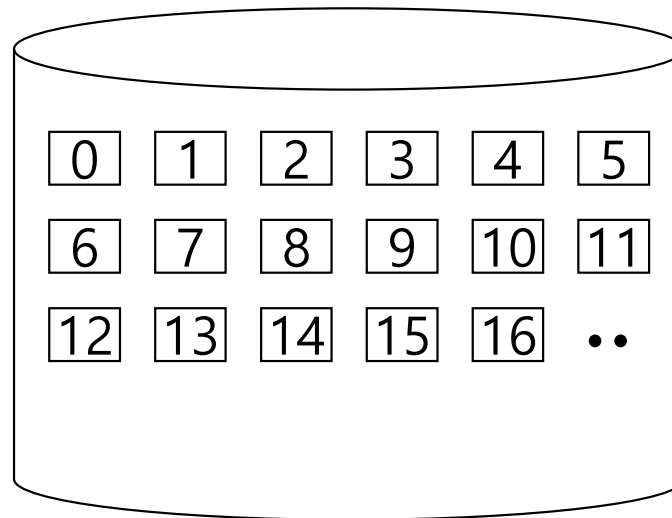
- 논리적 관점 (Linux 입장에서 디스크 구조)
 - 디스크는 디스크 블록들의 집합 (disk is a collection of disk blocks)
 - 디스크 블록의 크기는 보통 페이지 프레임의 크기와 같다 (4K, 8K)
 - 블록 번호를 Sector Address로 변환하는 작업은 컨트롤러나 디스크 드라이버에서 수행



Disk 블록 관리

- 시나리오

- 새로운 14 K 크기의 파일을 디스크에게 쓰려고 함
- 디스크 블록의 크기는 4K로 가정 (따라서 4개의 디스크 블록이 필요)
- 아래 그림에서 빗금이 있는 디스크 블록들은 이미 사용 중인 (파일의 데이터터를 갖고 있는) 블록이라고 가정
- 어떤 디스크 블록들을 할당할 것인가? (disk block allocation problem)

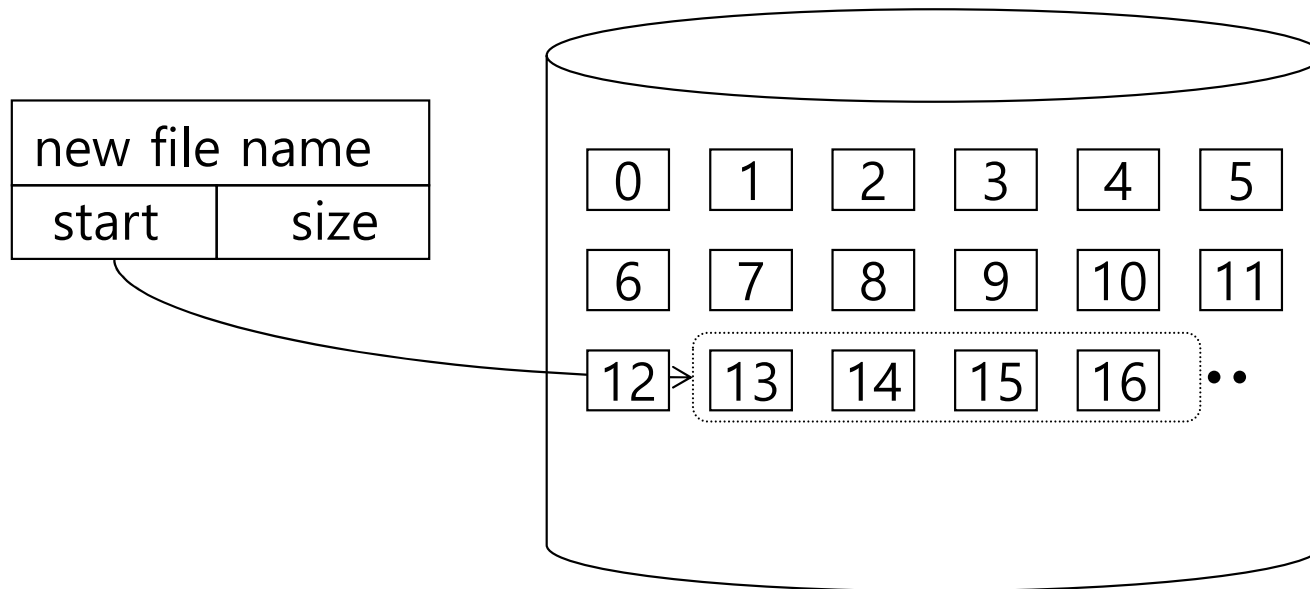


- 디스크 블록 할당 방법

- 연속 할당 방법 (sequential allocation)
- 불연속 할당 방법 (non sequential allocation)

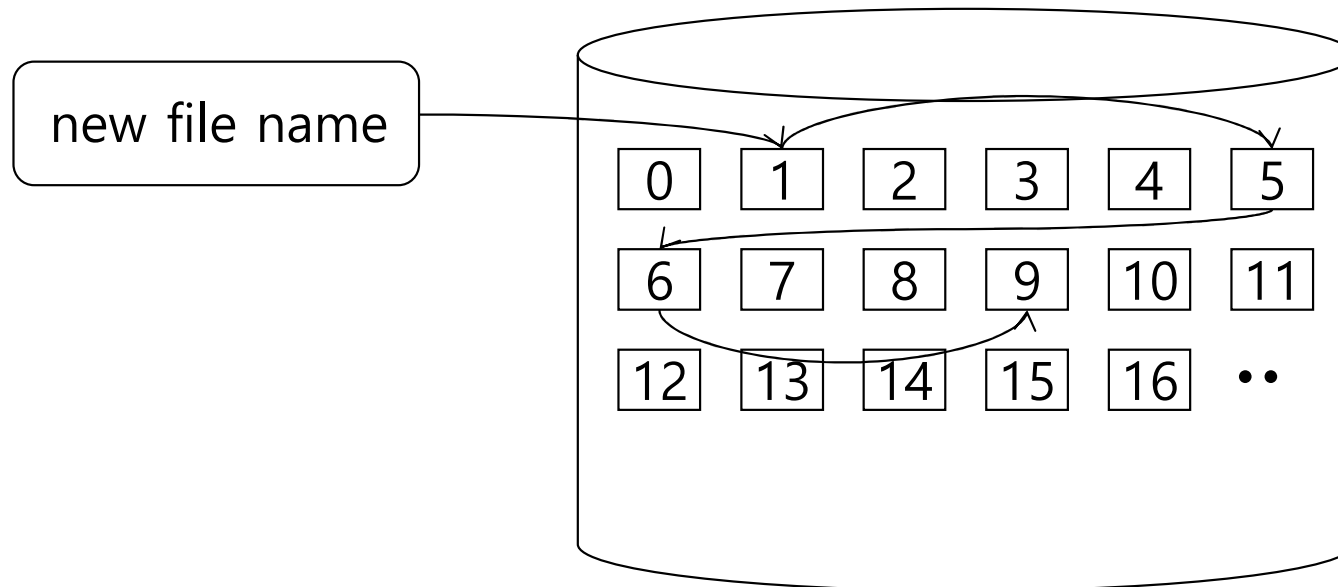
Disk 블록 관리

- 연속 할당 방법



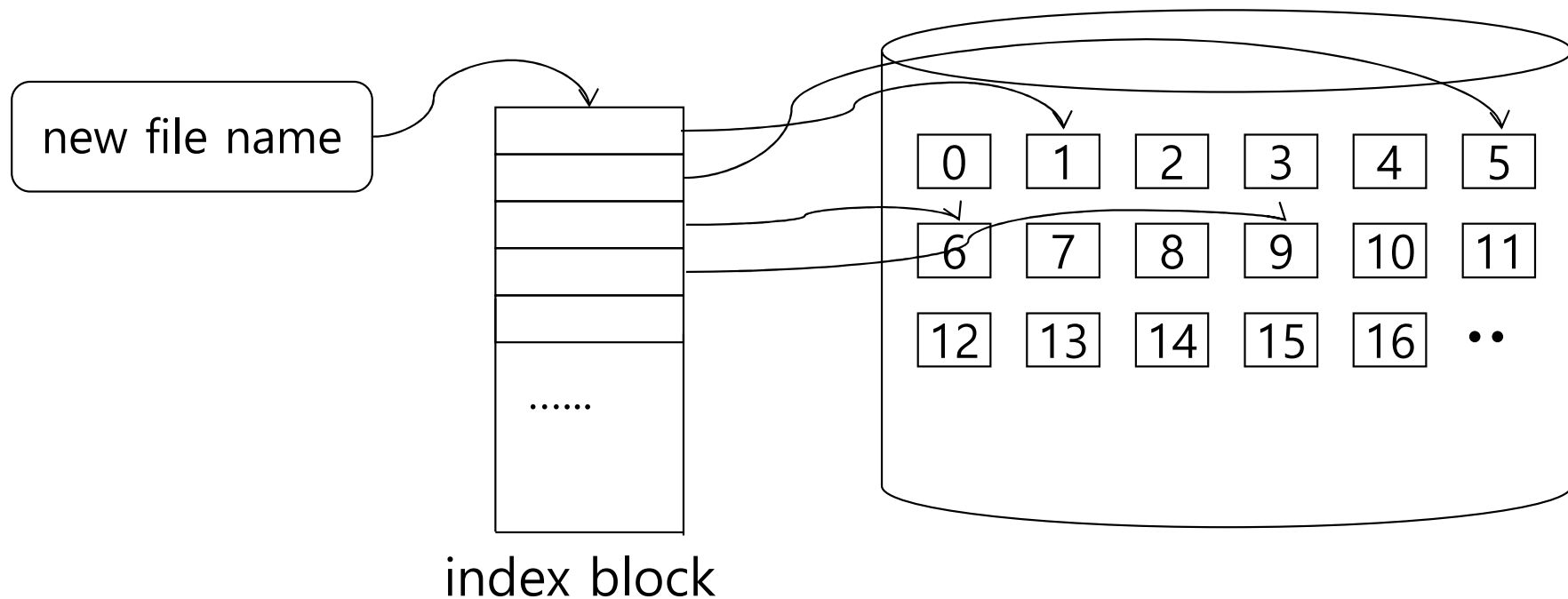
Disk 블록 관리

- 불연속 할당 방법 : 블록 체인 (block chain) 방법
 - 현재 시나리오에서는 1, 5, 6, 9 블록을 할당
 - 할당된 블록을 링크로 연결



Disk 블록 관리

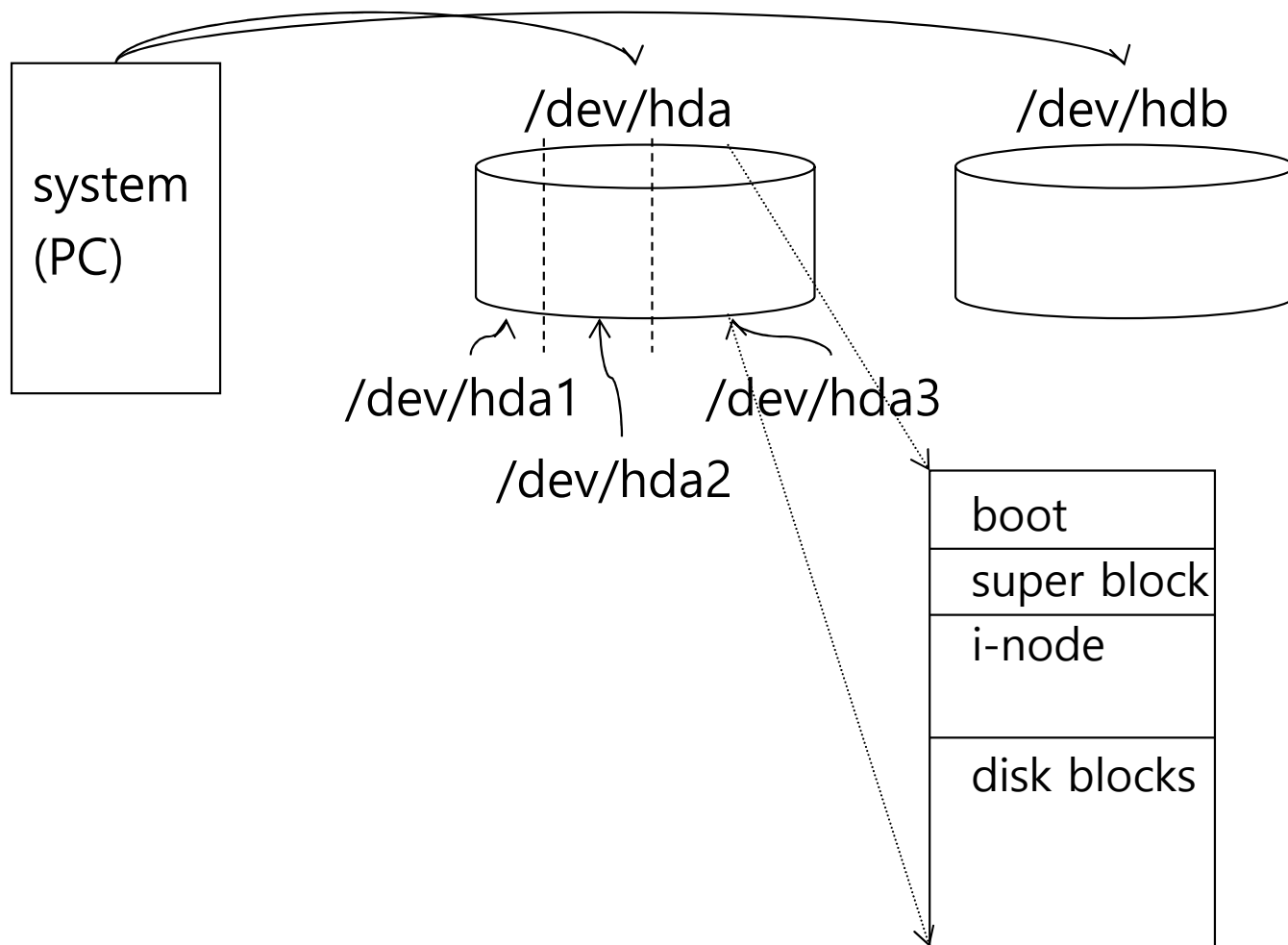
- 불연속 할당 방법 : 인덱스 블록 (index block) 기법



File System 구조

- 파일 시스템

- 디스크의 각 파티션마다 존재
- 4 부분으로 구성 : boot, super block, inode, data blocks
- mkfs, newfs 등의 명령으로 생성

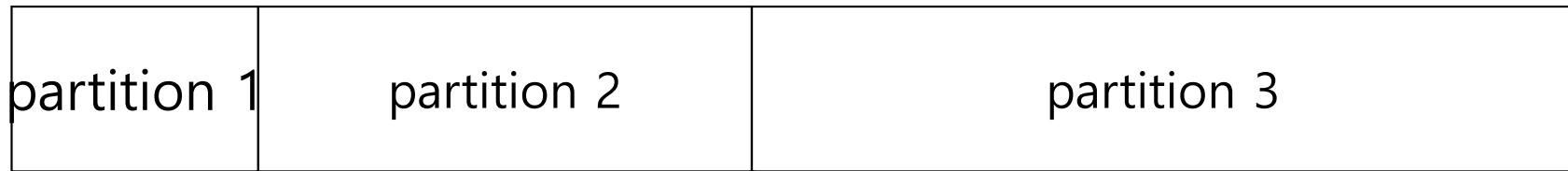


File System 구조

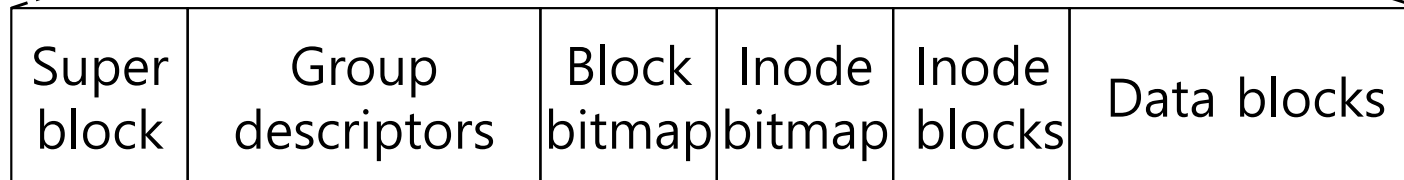
- 수퍼 블록 (super block)
 - 파일 시스템의 정보 관리하는 객체 (파일의 정보를 inode가 관리하는 것과 유사)
 - 파일 시스템 유형 : ext2/4, nfs, msdos, coda, proc, ntfs, ...
 - Ext2/4의 경우
 - free inode와 free block들을 bitmap으로 관리
 - block group 정의 (ffs의 cylinder group와 유사한 개념)
 - 결함 허용 기능 제공
 - nfs (Network File System by Sun)의 경우
 - 각 파일 인터페이스를 서버에 대한 RPC로 변환
 - XDR 사용
 - 약한 일관성 (weak consistency) 보장

Structure of the Ext2 File System

hard disk



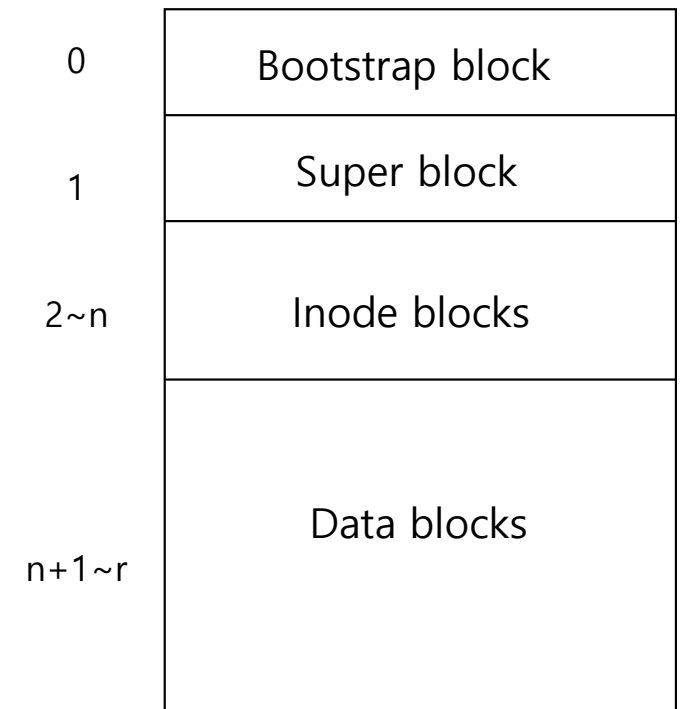
Ext2 file system



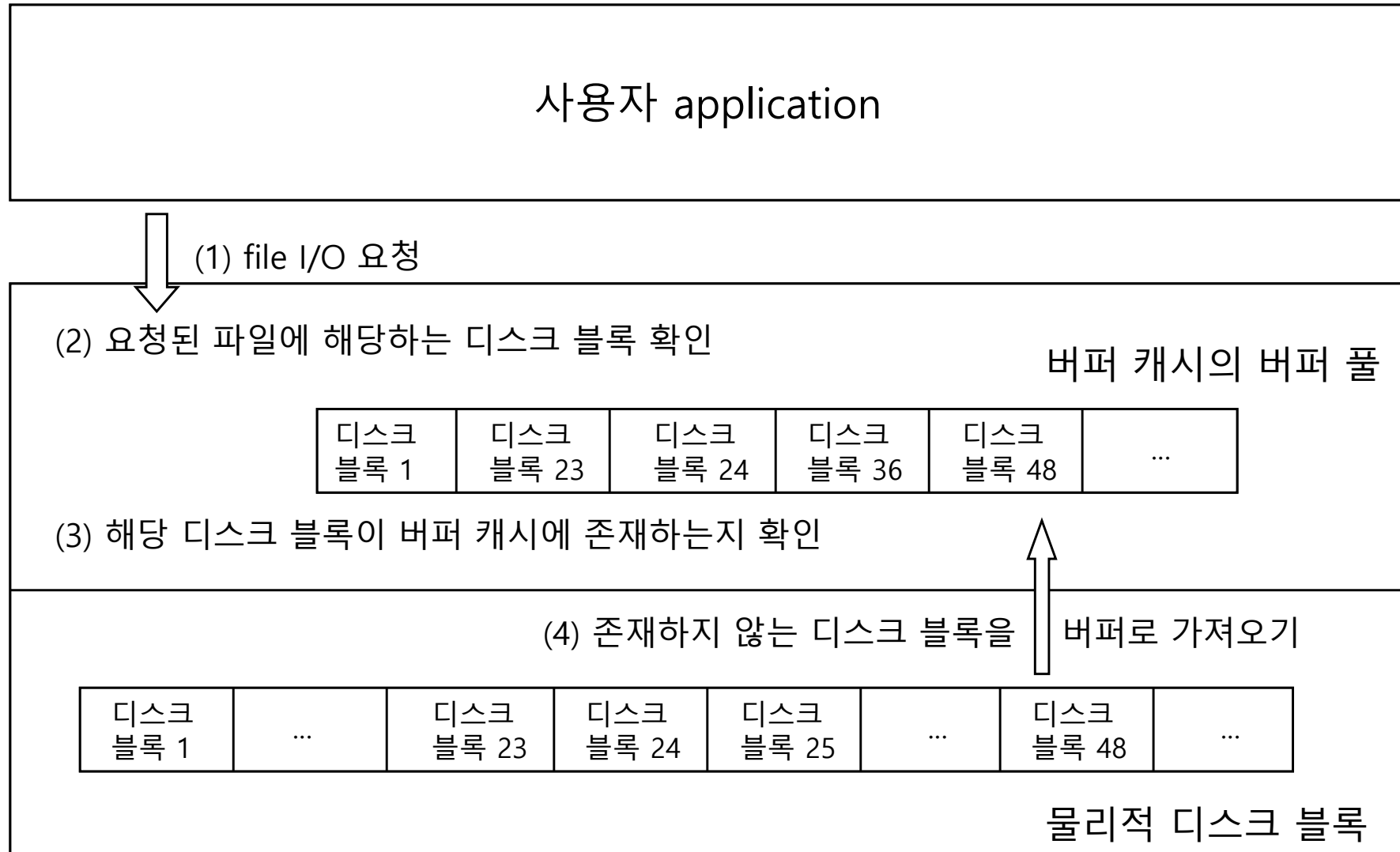
block group

UNIX File System

- File system layout
 - Bootstrap block(0)- kernel loading program
 - Super block(1) - file system information usually kept in memory
 - # of blocks for file system (r)
 - # of blocks for inode blocks ($n-1$)
 - # of blocks for data blocks ($r-n$)
 - File system last update time
 - A list of free data block numbers
 - A list of free inode numbers
 - Inode blocks(2 ~ n)
 - A list of inode structures
 - Data blocks($n+1$ ~ r)
 - A list of data blocks



Caching: Buffer cache



Caching: sync and fsync

```
#include <unistd.h>
```

```
void sync(void);
```

```
int fsync(int fildes);
```

- sync – flush all the main memory buffers containing information about file systems to disk
- fsync – flush out all data and attributes associated with a particular file
- Return of fsync:
 - Success: 0
 - Fail: -1

File System 정보

```
#include <sys/statvfs.h>
```

```
int statvfs(const char *path, struct statvfs *buf);
```

```
int fstatvfs(int fd, struct statvfs *buf);
```

- path 또는 fd 에 의해 참조되는 파일 시스템에 대한 정보를 return

File System 정보 예: fsys

```
/* fsys -- 파일 시스템 정보를 프린트한다. */  
/* 파일 시스템 이름이 인수로 전달된다. */
```

```
#include <sys/statvfs.h>  
#include <stdlib.h>  
#include <stdio.h>
```

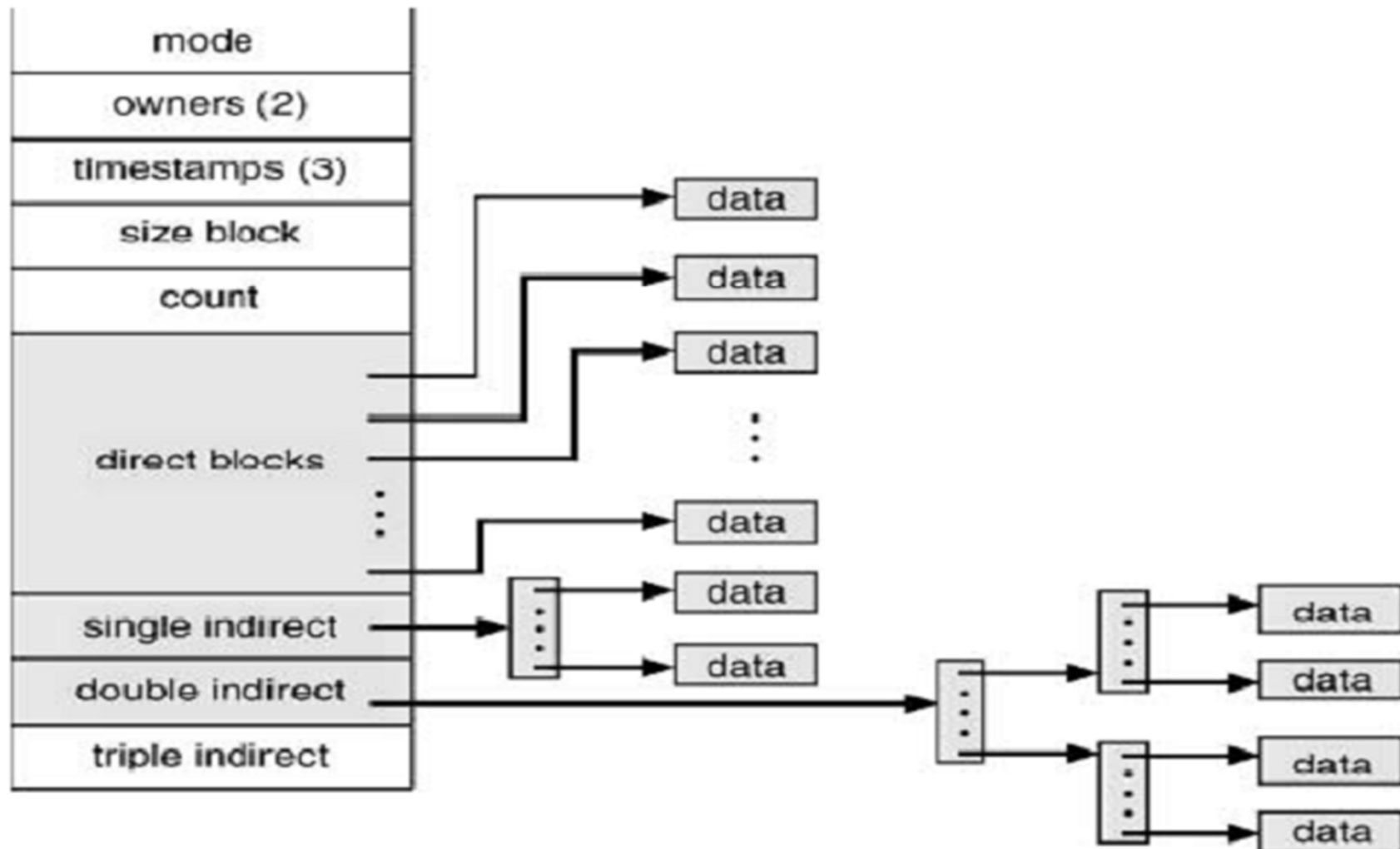
```
main (int argc, char **argv)  
{  
    struct statvfs buf;  
    if (argc != 2) {  
        fprintf (stderr, "usage: fsys filename\n");  
        exit (1);  
    }  
    if (statvfs (argv[1], &buf) != 0) {  
        fprintf (stderr, "statvfs error\n");  
        exit (2);  
    }  
    printf ("%s: %d free blocks %d free inodes %d\n",  
            argv[1], buf.f_bfree, buf.f_ffree);  
    exit (0);  
}
```

inode (index node)

- inode 정의
 - Linux에서 파일을 관리하기 위한 객체
 - 파일이 새로 생성되면 만들어진다.
 - internal representation of a file
 - every file has one inode
 - 파일의 모든 정보를 관리
 - 파일에 속한 블록 위치 (index block 방법과 유사)
 - 파일 소유자 및 접근 권한
 - 파일 시간 정보
 - 파일 유형 : 커널은 정규 파일 뿐만 아니라 디렉토리, 디바이스, 파이프, 소켓 등도 파일이라는 추상화 객체로 관리
 - Disk에 정적으로 존재

inode (index node)

- inode structure



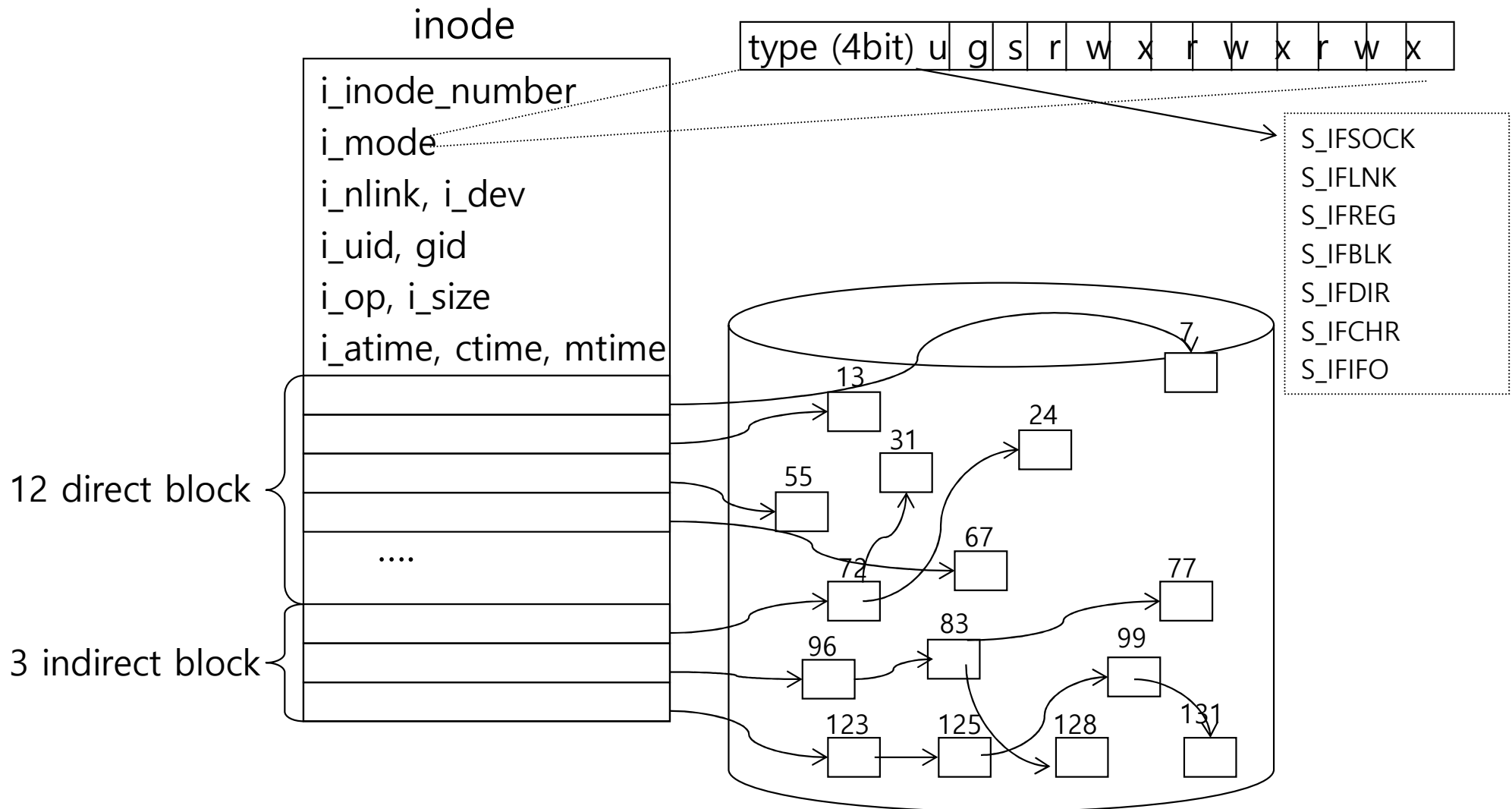
from "Operating System Concepts" written by Siberschatz

inode (index node)

- inode에서 디스크 블록 관리
 - direct blocks: data block을 가리킴
 - indirect blocks
 - single indirect block
 - index block을 가리킴.
 - 이 index block은 실제 data block을 가리키는 포인터들로 구성된다.
 - double indirect block
 - index block이 two layers로 구성.
 - 첫번째 층의 index block은 다른 index block을 가리키는 포인터들로 구성
 - 두번째 층의 index block은 실제 data block을 가리키는 포인터들로 구성
 - triple indirect block
 - index block이 three layers로 구성.
 - 첫번째/두번째 층의 index block은 다른 index block을 가리키는 포인터들로 구성
 - 세번째 층의 index block은 실제 data block을 가리키는 포인터들로 구성
 - 예제:
 - 하나의 디스크블록 8Kbyte, 포인터 64bit(8byte)인 경우, 즉 하나의 블록이 1024개 포인터 가짐)

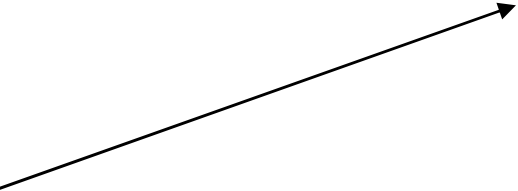
inode (index node)

- inode details in Linux



inode (index node)

- 파일 속성

- 파일 이름(name)
- 파일 유형(type) 
- 파일 크기(size)
- 시간 정보(time, date)
- 파일 소유자(user identification)
 - protection, security, and usage monitoring
- 접근 제어(access control)
 - reading, writing, executing
- number of blocks, link counter
- access time, modification time, change time
- list of addresses of data blocks
- open 이후에는 현재 접근 위치(offset)

- Regular file
- Directory
- Symbolic link
- Block-oriented device file
- Character-oriented device file
- Pipe and named pipe (also called FIFO)
- Socket

inode (index node)

접근 제어

3 class

user			group			other		
read	write	execute	read	write	execute	read	write	execute
400	200	100	40	20	10	4	2	1

access control

Permissions (3 for owner, 3 for group, 3 for other)			Owner	Group	Date and time of last modification			
-	rw-r--r--	1	mdw	users	2321	Mar 15	1994	Fontmap
-	rw-r--r--	1	mdw	users	139836	Aug 11	09:11	Index.whole
d	rxr-xr-x	2	mdw	users	1024	Jan 25	1994	Xfonts
d	rxr-xr-x	3	mdw	users	1024	Sep 20	07:40	bin
-	rw-r--r--	1	mdw	users	124408	Nov 2	10:53	bitgif.tar.gz
d	rxr-xr-x	2	mdw	users	2048	Jan 21	1994	bitmaps

Type of file ("d" means "directory")

Number of hard links

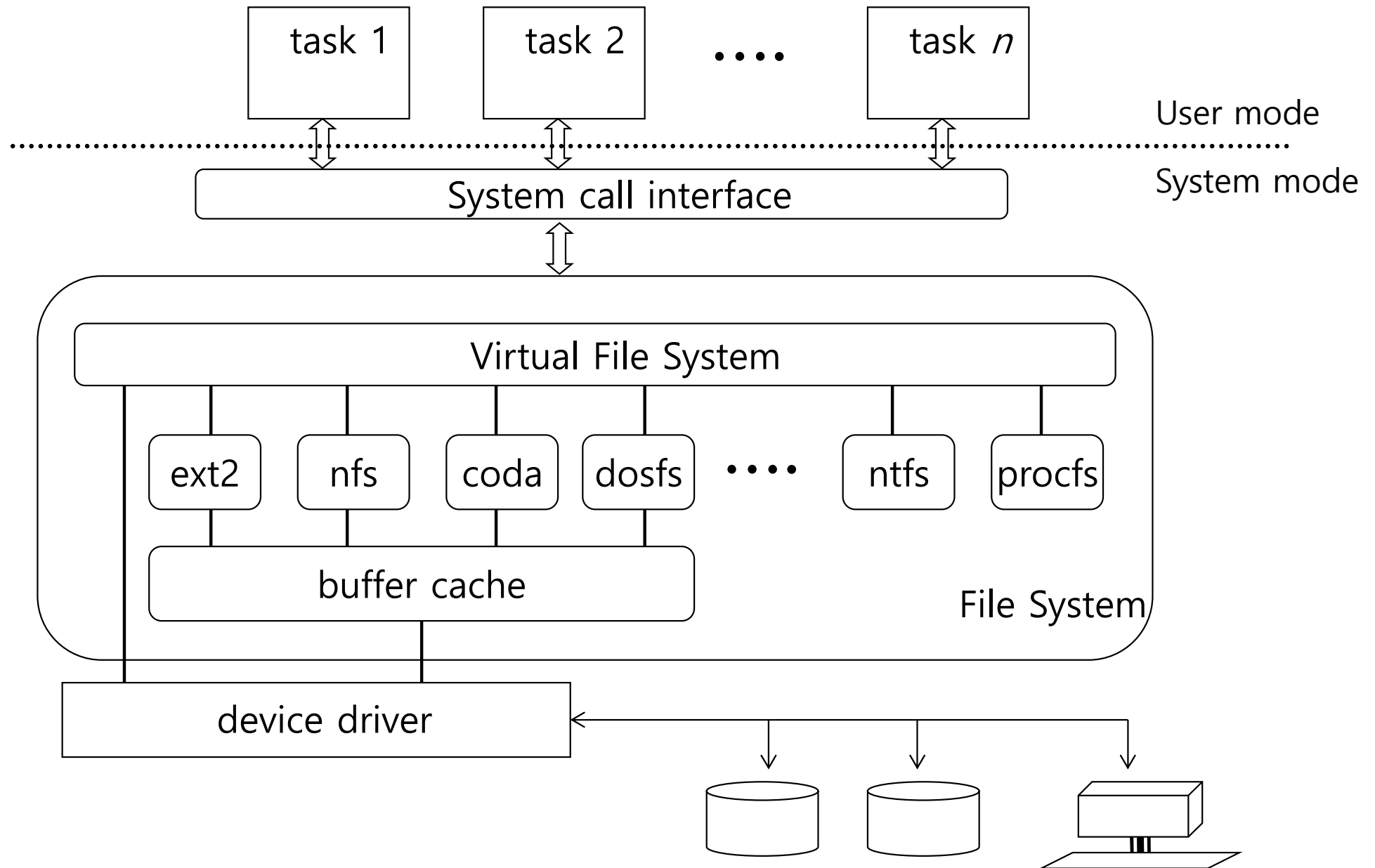
Size in bytes
(for a directory, bytes used to store directory information)

Name

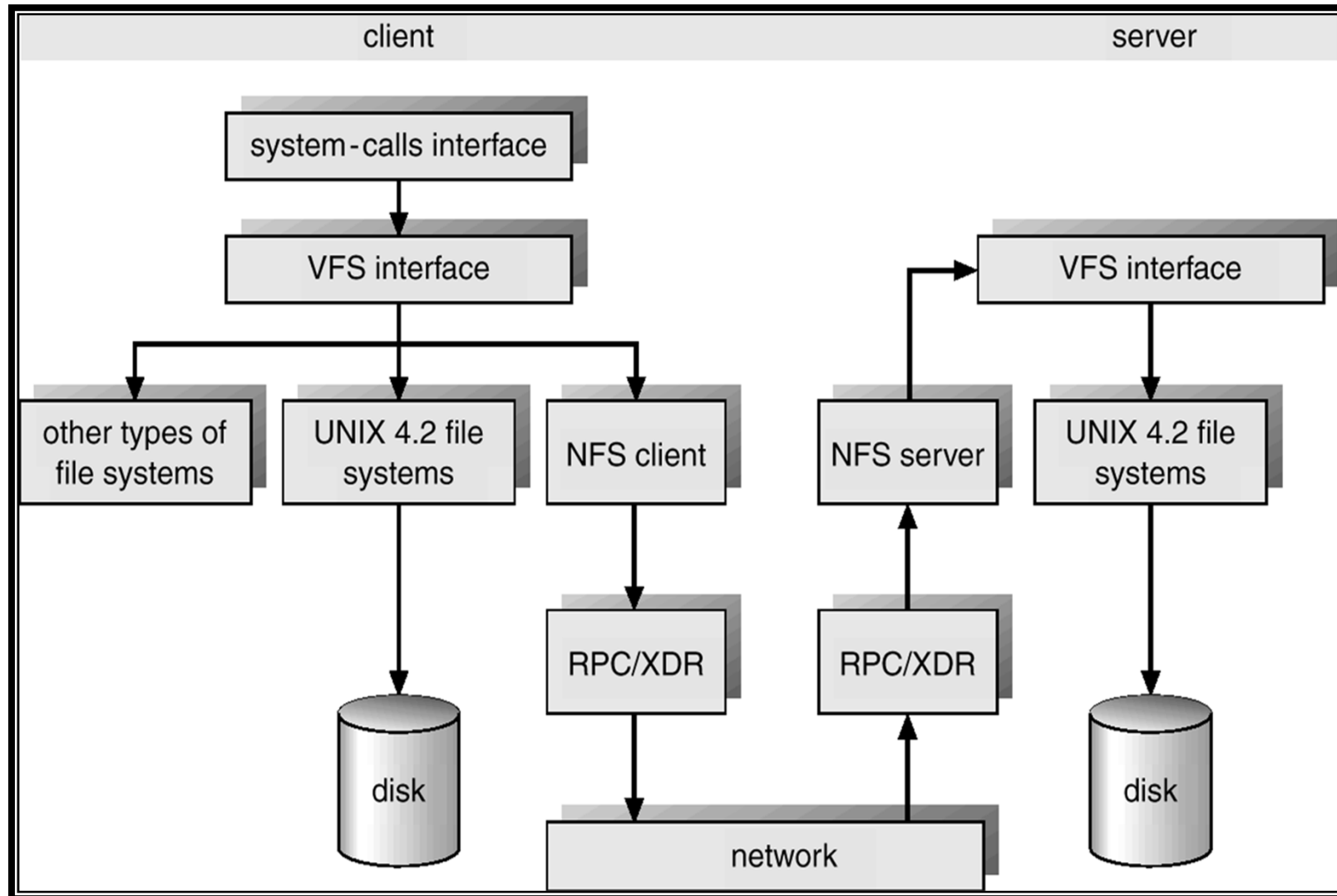
inode (index node)

- 장치 파일의 inode 구조
 - 파이프 (pipe)
 - 간접 블록(indirect block) 포인터를 사용하지 않음
 - readers, writers, read pointer, write pointer 필드 존재
 - 장치 파일
 - 문자 장치 파일, 블록 장치 파일
 - 직접 블록(direct block)과 간접 블록(indirect block) 포인터를 모두 사용하지 않음
 - i_rdev : 장치 번호(device number)
 - 장치 번호는 주 번호(major number)와 부 번호(minor number)로 구성
 - 주 번호 : 장치 유형(device type)에 따라 서로 다른 번호 설정
 - 부 번호 : 장치 단위(device unit)에 따라 서로 다른 번호 설정
 - 소켓(socket)
 - 프로토콜 패밀리와 관련 함수 유지

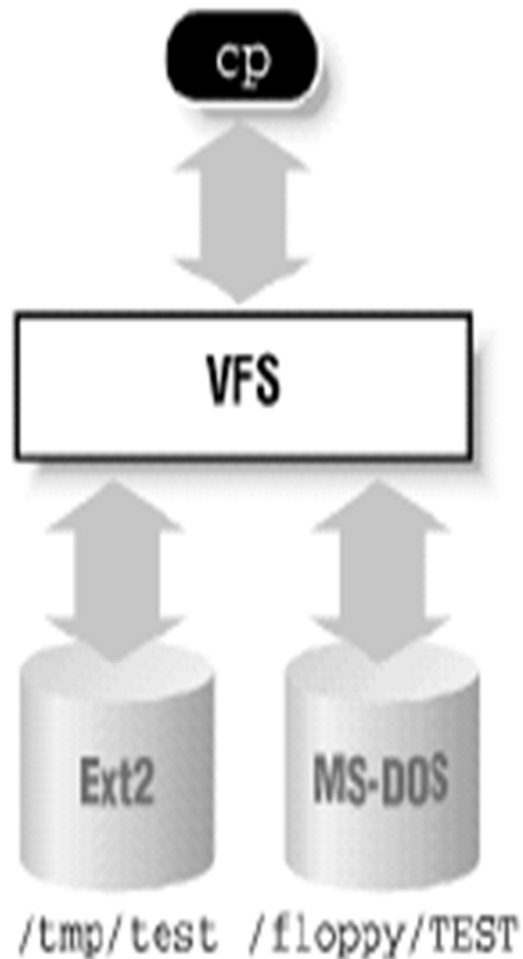
File System 전체 구조



Virtual File System



Virtual File System



```
inf = open("/floppy/TEST", O_RDONLY, 0);  
outf = open("/tmp/test",  
            O_WRONLY|O_CREATE|O_TRUNC, 0600);  
do {  
    l = read(inf, buf, 4096);  
    write(outf, buf, l);  
} while (l);  
close(outf);  
close(inf);
```

VFS는 일관된 인터페이스로 다양한 파일 시스템 지원 가능

File Interface

- 파일 생성
 - creat(), open() with create option, mkfifo(), mknod()
 - inode와 데이터 블록들을 할당
- 파일 접근
 - open(), close(), read(), write()
 - inode와 task_struct 연결
- 파일 제어
 - stat()
 - lseek(), dup(), link()
 - mkdir(), readdir()
- 파일 시스템 제어
 - mount()
 - sync(), fsck()

File Interface

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

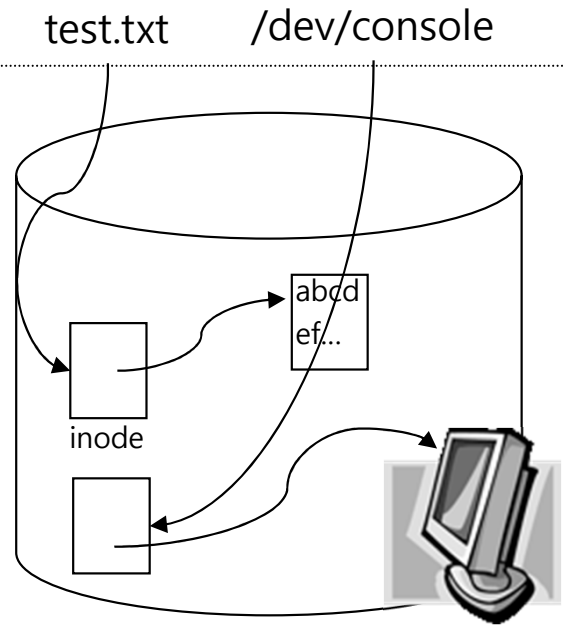
#define MAX_BUF 4
char fname[] = "/usr/member/choijm/test.txt";
char tmp_data[] = "abcdefghijklmn";

int main()
{
    int fd, size;
    char buf[MAX_BUF];

    fd = open(fname, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    write(fd, tmp_data, sizeof(tmp_data));
    close(fd);

    fd = open(fname, O_RDONLY);
    lseek(fd, 5, SEEK_SET);
    size = read(fd, buf, MAX_BUF);
    close(fd);

    fd=open("/dev/console", O_WRONLY)
    write(fd, buf, MAX_BUF);
    close(fd);
}
```



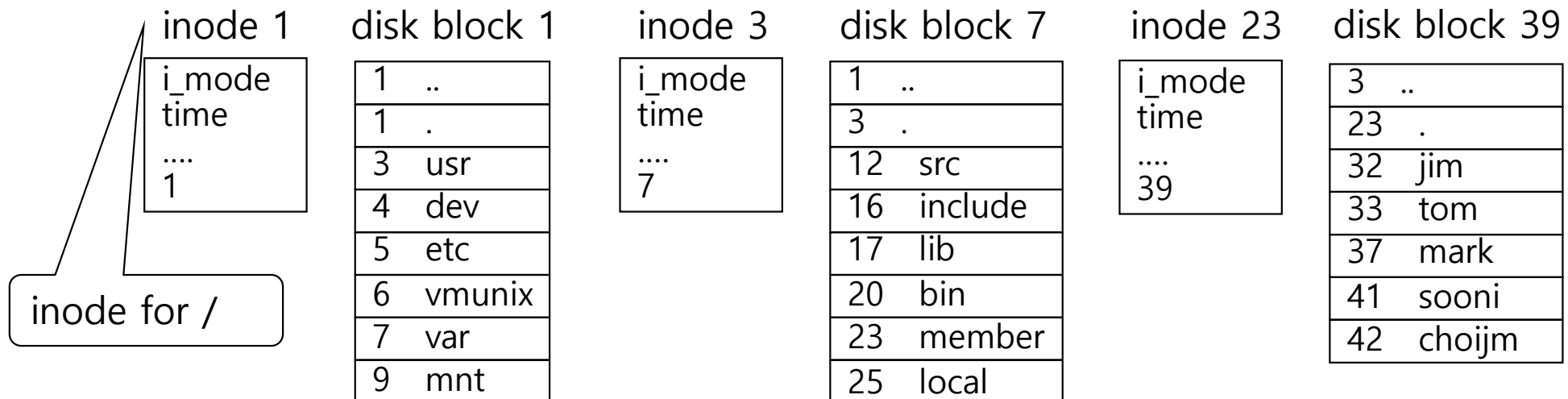
offset을 이동하는 인터페이스

터미널 같은 장치도 파일 인터페이스로 접근된다.

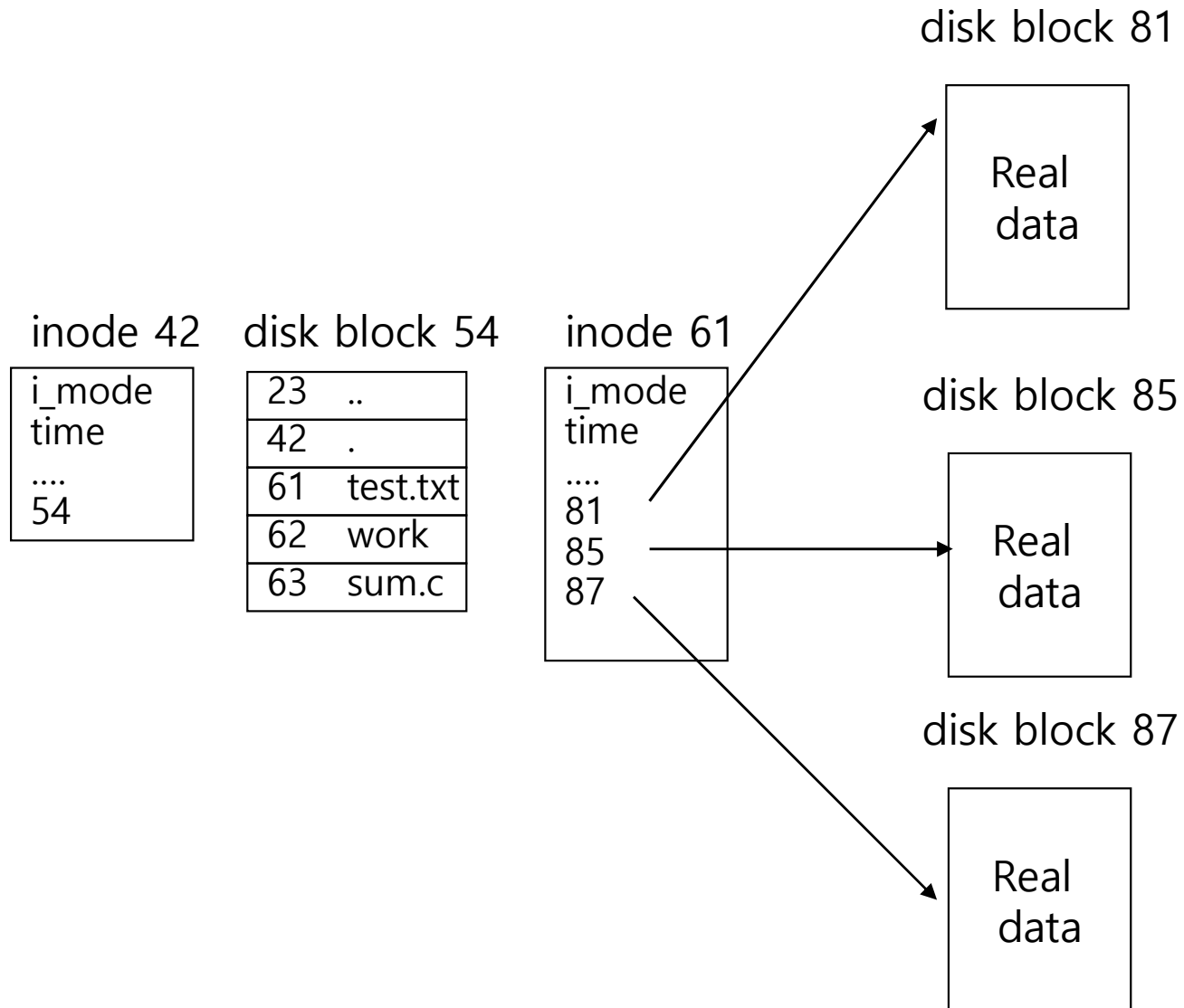
Directory 구조

- open()에서 요청한 파일에 대응되는 inode를 어떻게 발견할까?
 - “/usr/member/choijm/test.txt” → 해당 inode
- 디렉토리 (폴더)
 - 파일 이름과 inode를 연결하는 객체
 - 디렉토리 자체도 파일임
 - 계층 구조 제공
 - 디렉토리 각 항의 구조

inode number	file name
--------------	-----------

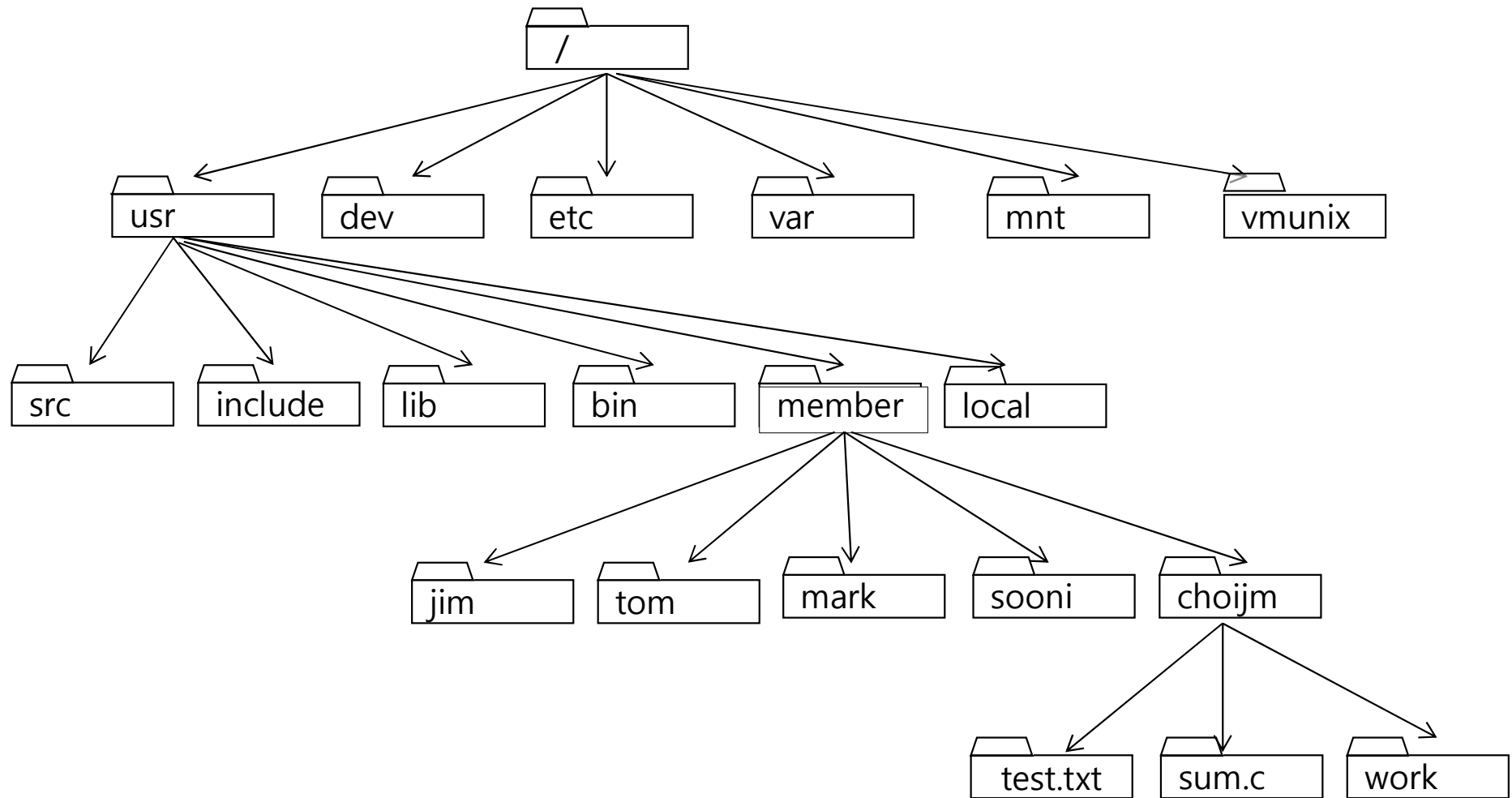


Directory 구조



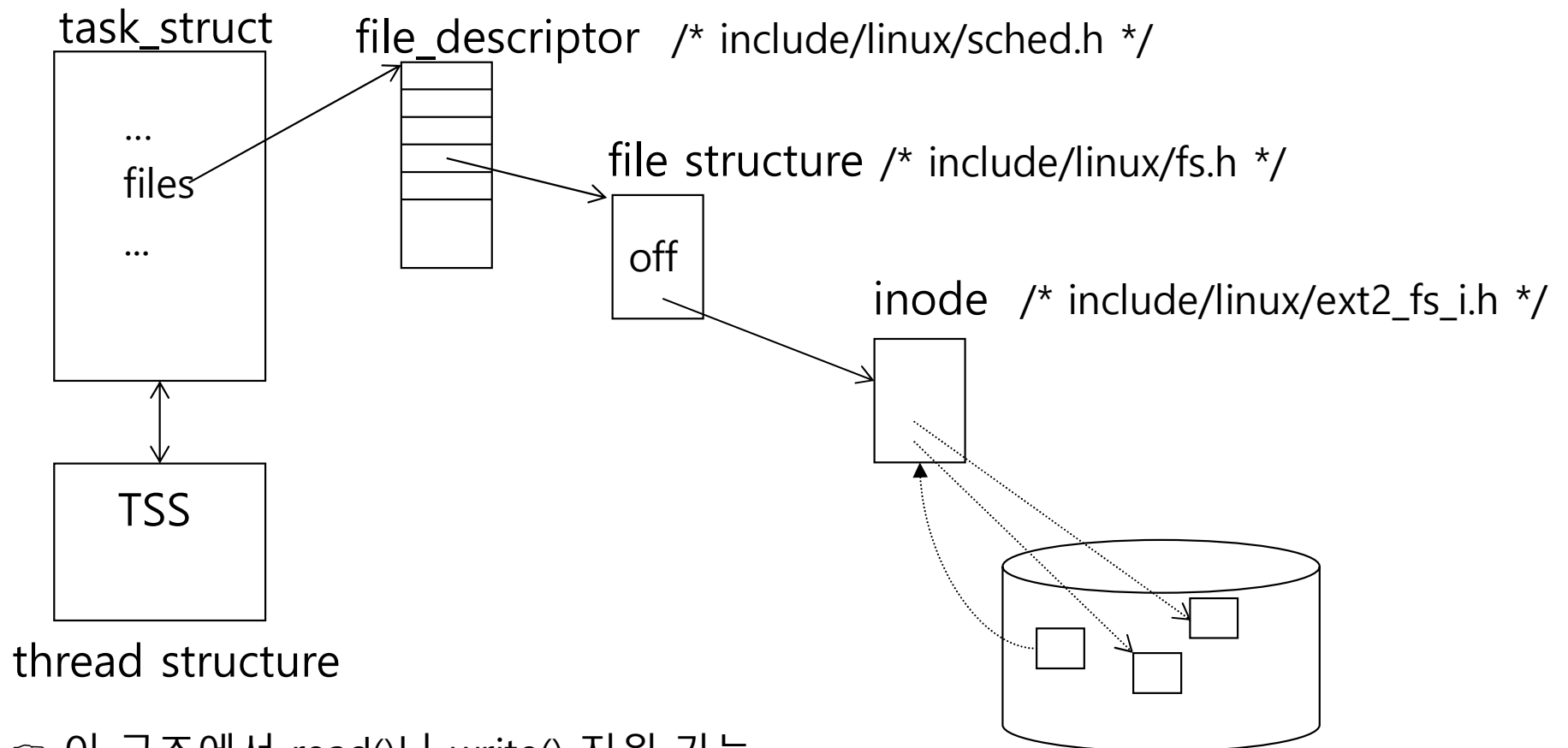
Directory 구조

- 파일 계층 구조 (hierarchical structure)



File Interface

- fd(file descriptor): details of open() system call
 - 디스크에서 접근하려는 파일의 inode를 찾는다.
 - inode를 메모리로 읽는다.
 - inode와 task 자료 구조를 연결한다 (이때 fd 사용)



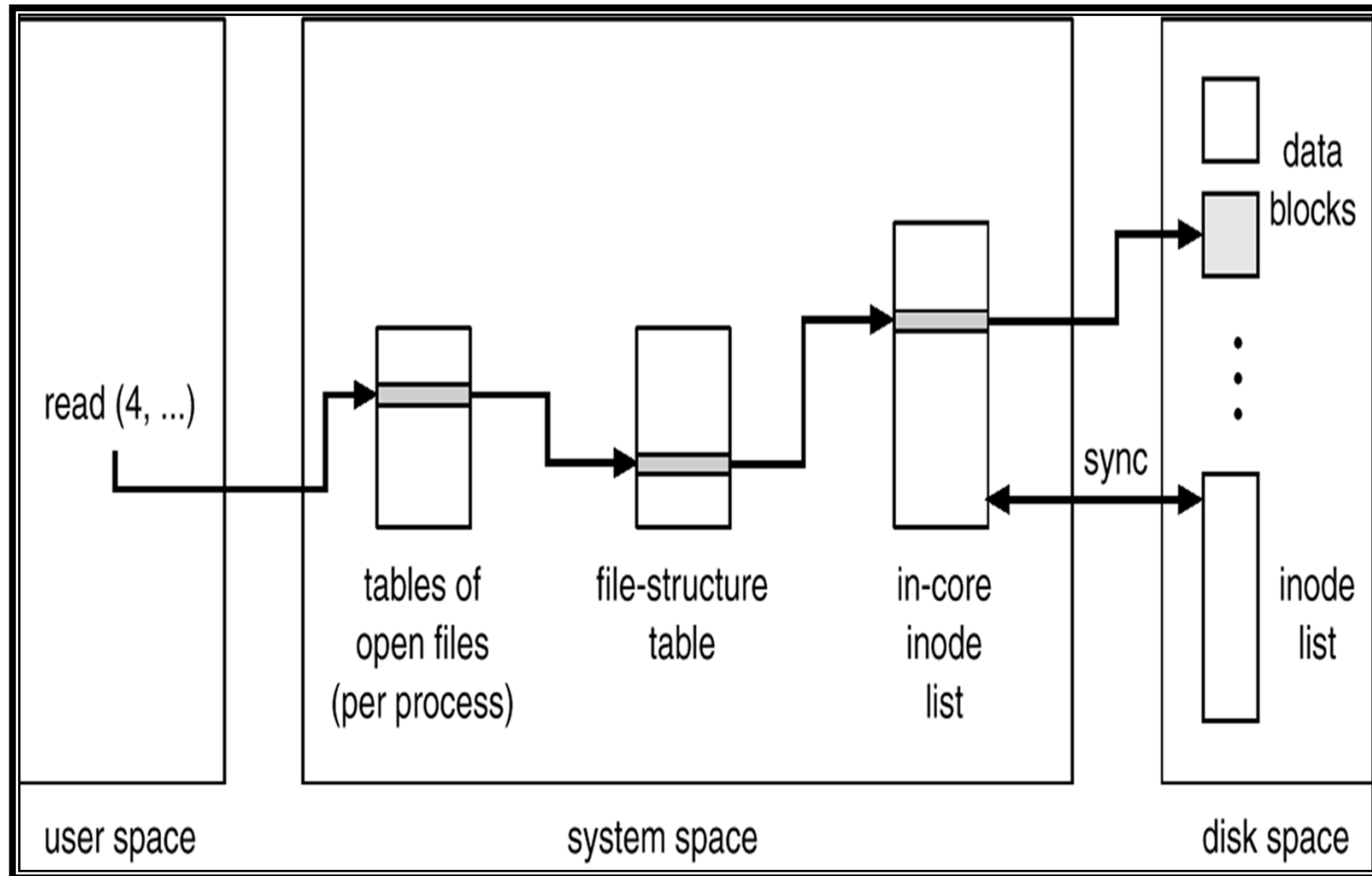
Kernel data structures for open files

- user file descriptor table
 - allocated per process
 - identifies all open files for a process
 - when a process “open” or “creat” a file, the kernel allocates an entry
 - return value of “open” and “creat” is the index into the user file descriptor table
 - contains pointer to file table entry

Kernel data structures for open files

- file table
 - global kernel structure
 - contains the description of all open files in the system
 - file status flag (open mode)
 - current file offset
 - contains pointer to in-core inode table entry
- in-core inode table
 - global kernel structure
 - when a process opens a file, the kernel converts the filename into an identity pair(device number, inode number)
 - the kernel then loads the corresponding inode into in-core inode table

Kernel data structures for open files



UNIX file access primitives

- open: opens a file for reading, writing or creating a file
- creat: creates an empty file
- close: closes a previously opened file
- read: extracts information from a file
- write: places information into a file
- lseek: moves to a specified byte in a file
- unlink: removes a file
- remove: alternative method to remove a file
- fcntl: controls attributes associated with a file

open() System Call

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(const char *pathname, int flag, [mode_t mode]);
```

- pathname: absolute or relative path name
- flag: some macros in <fcntl.h>
 - O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC, O_APPEND
- mode: used with O_CREAT flag
- return of open:
 - success: a file descriptor(≥ 0), fail: -1
- return of close:
 - success: 0, fail: -1

open() System Call

Process A

User file descriptor table		
fd	fd_flag	ptr
0(stdin)		
1(stdout)		
2(stderr)		
3		
4 :		

Process B

User file descriptor table		
fd	fd_flag	ptr
0(stdin)		
1(stdout)		
2(stderr)		
3		
4		

file table

flag	cnt	offset	ptr
0			
1			
0			
0			
0			
1			
0			
1			
0			
0			
1			
0			

v-node table

v-node 정보
i-node 정보
v-node 정보
i-node 정보
v-node 정보
i-node 정보

open() System Call

```
/* 초보적인 프로그램 예 */

/* 이 헤더 파일들은 아래에서 논의한다 */
#include <fcntl.h>
#include <unistd.h>

main()
{
    int fd;
    ssize_t nread;
    char buf[1024];

    /* 화일 "data"를 읽기 위해 개방한다 */
    fd = open("data", O_RDONLY);

    /* 데이터를 읽어 들인다 */
    nread = read(fd, buf, 1024);

    /* 화일을 폐쇄한다 */
    close(fd);
}
```

open() System Call

```
#include <stdlib.h>                /* exit 호출을 위한 것임 */
#include <fcntl.h>

char *workfile="junk"; /* workfile 이름을 정의한다 */

main()
{
    int filedes;

    /* <fcntl.h>에 정의된 O_RDWR를 사용하여 개방한다 */
    /* 화일을 읽기/쓰기로 개방한다 */
    if ((filedes = open (workfile, O_RDWR)) == -1)
    {
        printf ("Couldn't open %s\n", workfile);
        exit (1);          /* 오류이므로 퇴장한다 */
    }

    /* 프로그램의 나머지 부분이 뒤따른다 */

    exit (0);              /* 정상적인 퇴장 */
}
```

open() System Call

```
#include <stdlib.h>
#include <fcntl.h>

#define PERMS 0644 /* O_CREAT를 사용하는 open을 위한 허가 */
char *filename="newfile";

main()
{
    int filedес;
    if ((filedes = open (filename, O_RDWR | O_CREAT, PERMS)) == -1)
    {
        printf ("Couldn't create %s\n",filename);
        exit (1);          /*오류이므로 퇴장한다*/
    }
    /* 프로그램의 나머지 부분이 뒤따른다 */
    exit (0) ;
}
```

creat() System Call

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int creat(const char *pathname, mode_t mode);
```

- pathname: absolute or relative path name
- mode: access permission (in octal)
 - 4(read), 2(write) and 1(execute) for owner, group and others
 - eg. 0644 means r/w for owner, r for group and others
- Return:
 - success: a file descriptor(>= 0)
 - fail: -1

```
filedes = creat("/tmp/newfile", 0644);
```

```
filedes = open("/tmp/newfile", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

close() System Call

```
#include <unistd.h>
```

```
int close(int filedes);
```

- Return of close:
 - success: 0, fail: -1
- 프로그램의 수행이 끝나면 모든 개방된 파일은 자동적으로 close됨

```
filedes = open ("file", O_RDONLY);
```

```
.
```

```
.
```

```
.
```

```
close(filedes);
```

read() System Call

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buffer, size_t n);
```

- fd: file descriptor returned from open or creat
- buffer: starting address where data is stored
 - The user program should reserve enough buffer area
- n: # of bytes to read
- Return:
 - # of successfully read bytes

```
int fd;
```

```
ssize_t nread;
```

```
char buffer[SOMEVALUE];
```

```
/* fd는 open에 대한 호출로부터 얻은 것임 */
```

```
.
```

```
.
```

```
.
```

```
nread = read(fd, buffer, SOMEVALUE);
```

read-write pointer

- *file pointer*
 - 특정 파일 기술자를 통해 읽혀질/쓰여질 파일의 다음 바이트 위치를 기록 (bookmark)

```
int fd;
ssize_t n1,n2;
char buf1[512], buf2[512];
.
.
.
if(( fd = open("foo", O_RDONLY)) == -1)
    return (-1);

n1 = read(fd, buf1, 512);
n2 = read(fd, buf2, 512);
```

read-write pointer

```
/* count -- 한 파일내의 문자 수를 센다 */
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 512
main()
{
    char buffer[BUFSIZE];
    int filedес;
    ssize_t nread;
    long total = 0;
    /*"anotherfile"을 읽기 전용으로 개방 */
    if (( filedес = open ("anotherfile", O_RDONLY)) == -1)
    {
        printf ("error in opening anotherfile\n");
        exit (1);
    }
    /* EOF까지 반복하라. EOF는 복귀값 0에 의해 표시된다. */
    while( (nread = read(filedes, buffer, BUFSIZE)) > 0)
        total += nread;          /* total을 증가시킨다. */
    printf ("total chars in anotherfile: %ld\n", total);
    exit (0);
}
```


write() System Call

```
#include <unistd.h>
```

```
ssize_t write(int fd, void *buffer, size_t n);
```

- fd: file descriptor returned from open or creat
- buffer: 쓰여질 데이터에 대한 포인터
- n: # of bytes to write
- Return:
 - # of successfully written bytes

```
int fd;
```

```
ssize_t w1, w2;
```

```
char header1[512], header2[1024];
```

```
.
```

```
.
```

```
if( ( fd = open("newfile", O_WRONLY | O_CREAT | O_EXCL, 0644)) == -1)
```

```
    return (-1);
```

```
w1 = write(fd, header1, 512);
```

```
w2 = write(fd, header2, 1024);
```

The copyfile Example

```
#include <unistd.h>
#include <fcntl.h>
#define BUFSIZE 512
#define PERM 0644
int copyfile(const char *name1, const char *name2){
    int infile, outfile; ssize_t nread; char buffer[BUFSIZE];

    if((infile=open(name1, O_RDONLY))==-1) return -1;
    if((outfile=open(name2, O_WRONLY|O_CREAT|O_TRUNC, PERM))==-1){
        close(infile); return -2;
    }
    while((nread=read(infile, buffer, BUFSIZE))>0){
        if(write(outfile, buffer, nread)<nread){
            close(infile); close(outfile); return -3;
        }
    }
    close(infile); close(outfile);
    if(nread==-1) return -4;
    else return 0;
}
```

read와 write의 효율성

- Results of copyfile test

BUFSIZE	Real time	User time	System time
-----	-----	-----	-----
1	24.49	3.13	21.16
64	0.46	0.12	0.33
512	0.12	0.02	0.08
4096	0.07	0.00	0.05
8192	0.07	0.01	0.05

➔ 가장 좋은 성능은 시스템의 blocking factor의 배수 일 때

- 프로그램의 효율성 향상 방법

- 시스템 호출의 횟수를 줄여야 한다

dup() and dup2() System Calls

- synopsis

#include <unistd.h>

int dup(int oldfd);

int dup2(int oldfd, int newfd);

- description

- create a copy of the file descriptor oldfd
- close_on_exec flag is not copied.
- The old and new descriptors may be used interchangeably
 - if the file position is modified by using lseek on one of the descriptors the position is also changed for the other
- two descriptors do not share the close-on-exec flag

dup() and dup2() System Calls

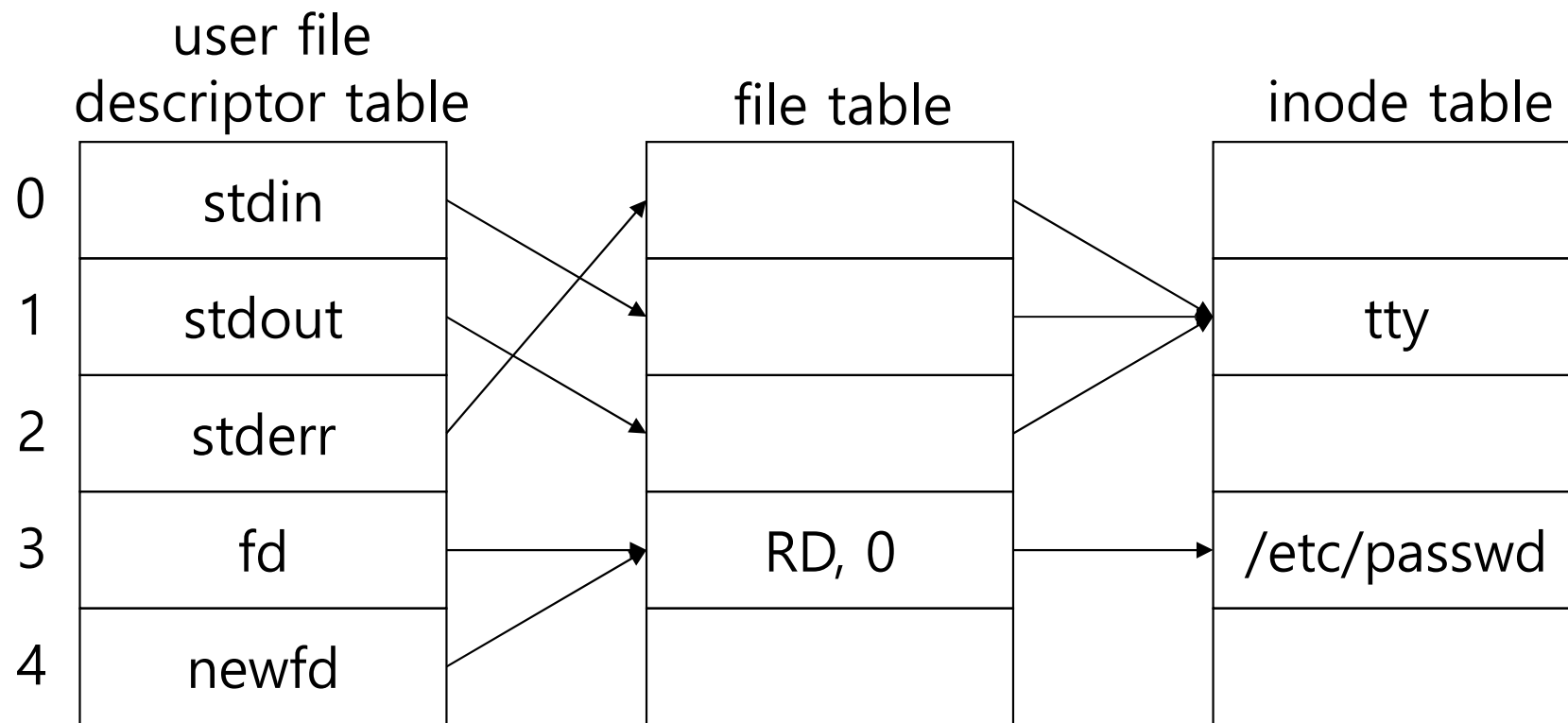
- dup uses the lowest-numbered unused descriptor for the new descriptor
- dup2 makes newfd be the copy of oldfd, closing newfd first if necessary.
- return value
 - the new descriptor, or -1 if an error occurred

- **example**

```
1) int fd = dup(STDOUT_FILENO);  
2) fd = open("input_file", O_RDONLY);  
   dup2(fd, STDIN_FILENO);  
3) fd = open("input_file", O_RDONLY);  
   close(STDIN_FILENO);  
   dup(fd);
```

dup() and dup2() System Calls

```
fd = open("/etc/passwd", O_RDONLY);  
newfd = dup(fd);
```



dup() and dup2() System Calls

- Example

```
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    int fd;
    fd = creat("dup_result", 0644);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    printf("hello world\n");
    return 0;
}
```

dup() and dup2() System Calls

```
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

main()
{
    char *fname = "test.txt";
    int fd1, fd2, cnt;
    char buf[30];

    fd1 = open(fname, O_RDONLY);
    if(fd1 < 0) {
        perror("open( )");
        exit (-1);
    }
    fd2 = dup(fd1);
    cnt = read(fd1, buf, 12);
    buf[cnt] = '\0';
    printf("fd1's printf : %s\n", buf);
    lseek(fd1, 1, SEEK_CUR);
    cnt = read(fd2, buf, 12);
    buf[cnt] = '\0';
    printf("fd2's printf : %s\n", buf);
}
```


dup() and dup2() System Calls

[test.txt의 내용]

Hello, Unix! How are you?

[수행 결과]

fd1's printf : Hello, Unix!

fd2's printf : How are you?

dup() and dup2() System Calls

```
#include <fcntl.h>
main()
{
    char *fname = "test.txt";
    int fd;

    if((fd = creat(fname, 0666)) < 0) {
        perror("creat( )");
        exit(-1);
    }
    printf("First printf is on the screen.\n");
    dup2(fd,1);
    printf("Second printf is in this file.\n");
}
```

[수행 결과]

\$ a.out

First printf is on the screen.

\$ cat test.txt

Second printf is in this file.

lseek와 random access

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int start_flag);
```

- read-write 포인터의 위치 변경
- fd: file descriptor returned from open or creat
- offset: relative offset
- start_flag:
 - SEEK_SET, SEEK_CUR, SEEK_END
- Return:
 - success: a new position in the file
 - fail: -1

lseek와 random access

```
off_t newpos;
```

```
.
```

```
.
```

```
newpos = lseek(fd, (off_t) -16, SEEK_END);
```

```
filedes = open(filename, O_RDWR);
```

```
lseek(filedes, (off_t)0, SEEK_END);
```

```
write(filedes, outbuf, OBSIZE);
```

```
off_t filesize;
```

```
int filedes;
```

```
.
```

```
.
```

```
filesize = lseek(filedes, (off_t)0, SEEK_END);
```

The Hotel Example(1)

- residents: 호텔에 투숙한 사람들의 이름을 기록한 파일
- 각 줄의 길이는 41개의 문자 (41번째는 '\n')

```
/* getoccupier -- residents 화일로부터 투숙객의 이름을 얻는다. */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#define NAMELENGTH 41
char namebuf[NAMELENGTH];          /*이름을 보관할 버퍼 */
int infile = -1;                   /*화일 기술자를 보관할 것임 */

char *getoccupier(int roomno)
{
    off_t offset;
    ssize_t nread;

    /* 화일을 처음으로 개방한다 */
    if ( infile == -1 &&
        (infile = open ("residents", O_RDONLY)) == -1)
    {
        return (NULL);              /* 화일을 개방하지 못함 */
    }
    offset = (roomno - 1) * NAMELENGTH;
```

The Hotel Example(2)

```
/* 방의 위치를 찾아 투숙객의 이름을 읽는다 */
if (lseek(infile, offset, SEEK_SET) == -1)
    reurn (NULL);
if ( (nread = read (infile, namebuf, NAMELENGTH)) <= 0)
    return (NULL);
/* 개행문자를 널 종결자(terminator)로 대체하여 하나의 스트링을 생성하라. */
namebuf[nread -1] = '\0';
return (namebuf);
}
```

```
/* listoc -- 모든 투숙객의 이름을 리스트하라 */
#define NROOMS          10
main()
{
    int j;
    char *getoccupier (int), *p;
    for ( j = 1; j<= NROOMS; j++)
    {
        if (p = getoccupier(j))
            printf ("Room %2d, %s\n", j, p);
        else
            printf ("Error on room %d\n", j);
    }
}
```

한 파일 끝에 자료 추가 하기

/* 화일의 끝으로 이동 */

```
lseek(filedes, (off_t)0, SEEK_END);
```

```
write(filedes, appbuf, BUFSIZE);
```

- 보다 일반적인 방법

```
filedes = open("yetanother", O_WRONLY | O_APPEND);
```

```
write(filedes, appbuf, BUFSIZE);
```

The unlink/remove System Call

```
#include <unistd.h>
```

```
int unlink(const char *pathname);
```

```
#include <stdio.h>
```

```
int remove(const char *pathname);
```

- 파일의 제거
- pathname: absolute or relative path name
- Return:
 - success: 0
 - fail: -1
- 디렉토리에 대해서는 remove 를 사용해야한다.

fcntl System Call

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ...);
```

- fd: file descriptor returned from open or creat
- cmd: F_GETFL, F_SETFL, ...
- Return:
 - success: an integer(≥ 0)
 - compute Return&O_ACCMODE
 - can be one of O_RDONLY, O_WRONLY, ...
 - fail: -1

fcntl System Call

```
/* filestatus -- 파일의 현재 상태를 기술한다. */
```

```
#include <fcntl.h>
```

```
int filestatus(int filedес)
```

```
{
```

```
    int arg1;
```

```
    if(( arg1 = fcntl (filedes, F_GETFL)) == -1)
```

```
    {
```

```
        printf ("filestatus failed\n");
```

```
        return (-1);
```

```
    }
```

```
    printf("File descriptor %d: ",filedes);
```

```
/* 개방시의 플래그를 테스트한다. */
```

```
switch ( arg1 & O_ACCMODE){
```

```
case O_WRONLY:
```

```
    printf ("write-only");
```

```
    break;
```

```
case O_RDWR:
```

```
    printf ("read-write");
```

```
    break;
```

```
case O_RDONLY:
```

```
    printf ("read-only");
```

```
    break;
```

```
default:
```

```
    printf("No such mode");
```

```
}
```

```
if (arg1 & O_APPEND)
```

```
    printf (" -append flag set");
```

```
printf ("\n");
```

```
return (0);
```

```
}
```

Standard input, output and Error

- Standard input, output and error
 - standard input(fd=0): keyboard
 - standard output(fd=1): terminal screen
 - standard error(fd=2): terminal screen
- File redirections on UNIX shell
 - standard input redirection:
\$ command < file
 - standard output redirection:
\$ command > file
 - standard error redirection:
\$ command 2> file

I/O의 예

```
/* io -- 표준 입력을 표준 출력으로 복사 */
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#define SIZE 512
```

```
main()
```

```
{
```

```
    ssize_t nread;
```

```
    char buf[SIZE];
```

```
    while ( (nread = read (0, buf, SIZE)) > 0)
```

```
        write (1, buf, nread);
```

```
    exit (0);
```

```
}
```

The Standard I/O Library

- The file access system calls
 - The basis for all input and output by UNIX programs
 - The primitives to handle data only in the form of simple sequences of bytes
- The standard I/O library
 - ANSI C standard
 - Offers many more facilities than the system calls
 - Offers efficient mechanisms
 - FILE instead of file descriptor
 - eg. `fopen()`, `getchar()`, `putchar()`, `getc()`, `putc()`, `scanf()`, `printf()`, ...

fopen() Library

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
main()
```

```
{
```

```
    FILE *stream;
```

```
    if ( ( stream = fopen ("junk", "r")) == NULL)
```

```
    {
```

```
        printf ("Could not open file junk\n");
```

```
        exit (1);
```

```
    }
```

```
}
```

getc(), putc() Library

```
#include <stdio.h>
```

```
int getc(FILE *istream); /* istream으로부터 한 문자를 읽어라. */
```

```
int putc(int c, FILE *ostream); /* ostream에 한 문자를 넣어라. */
```

- 예

```
int c;
```

```
FILE *istream, *ostream;
```

```
/* istream을 읽기전용으로 개방하고, ostream을 쓰기전용으로 개방하라. */
```

```
.
```

```
.
```

```
while( ( c=getc(istream)) !=EOF)
```

```
    putc(c, ostream);
```

fprintf를 이용한 오류 메시지 출력

```
#include <stdio.h> /*표준 에러의 정의를 위해 */
```

```
.
```

```
.
```

```
fprintf(stderr, "error number %d\n", errno);
```

```
/* notfound 파일 오류를 출력하고 퇴장(exit)한다. */
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int notfound (const char *programe, const char *filename)
```

```
{
```

```
    fprintf (stderr, "%s: file %s not found\n", programe,  
                                                     filename);
```

```
    exit (1);
```

```
}
```


The errno variable and perror library

- errno variable

```
#include <errno.h>
```

```
...
```

```
if((fd=open("nonesuch", O_RDONLY))==-1)
```

```
    fprintf(stderr, "error %d\n", errno);
```

```
...
```

- perror library routine

Program:

```
...
```

```
perror("error opening nonesuch");
```

```
...
```

Run:

```
error opening nonesuch: No such file or directory
```

File System 연결(mount)

- 파티션의 장점

- 신뢰성(Reliability) 향상
- 서로 다른 종류의 파일 시스템 지원 가능 (NT와 Ext2)
- 같은 파일 시스템일지라도 access pattern에 따라 다른 설정 가능
- 특정 프로세스에 의한 디스크 공간 독점 방지

- 파티션을 지원하기 위해서는

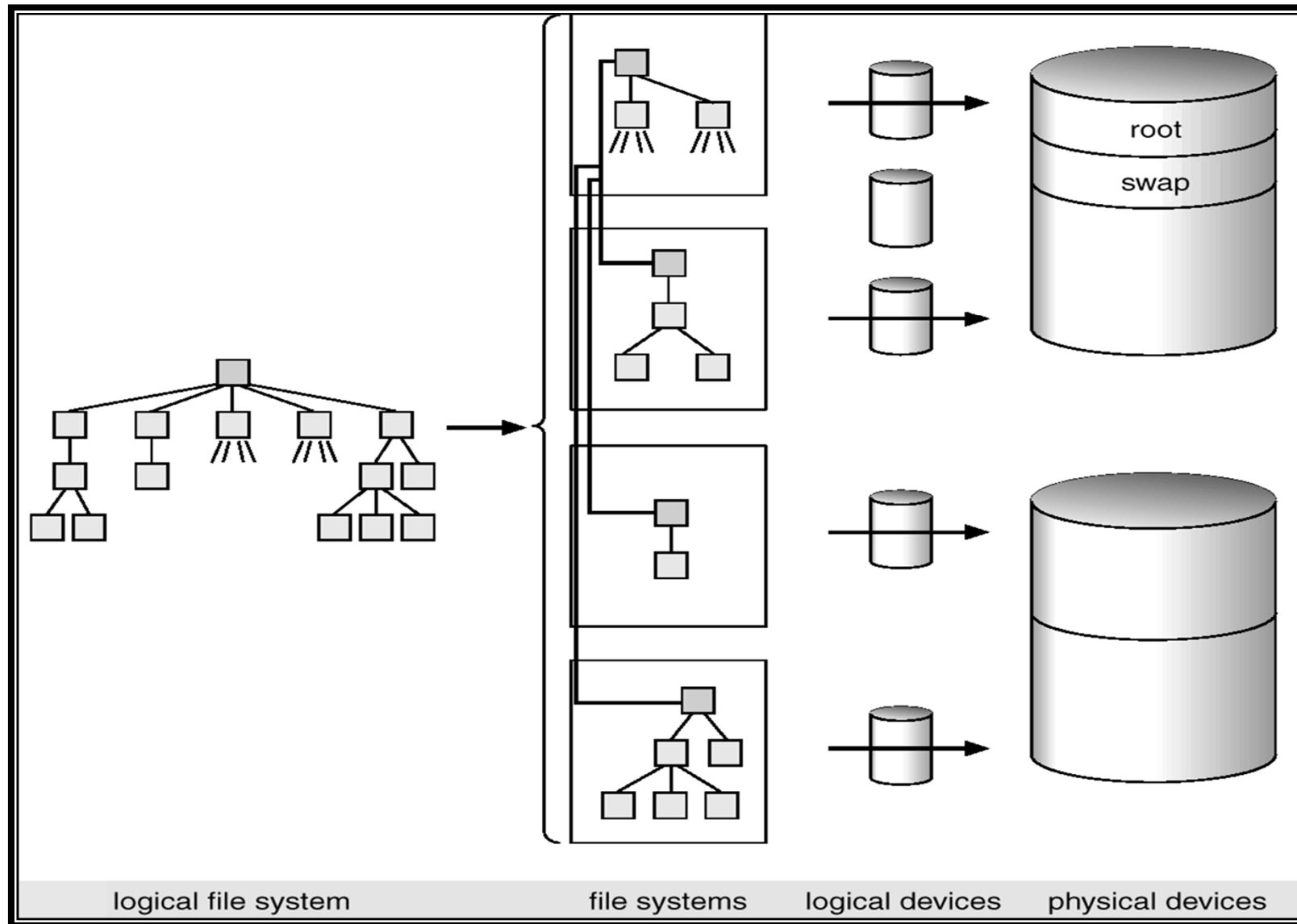
- 논리적인 하나의 파일 시스템이 실제로는 여러 디스크 or 파티션에 존재 가능

File System 연결(mount)

- File system 생성
 - format: /usr/bin/superformat
 - create: /sbin/mke2fs
 - initializes superblock and group descriptor
 - defective block on the partition 확인
 - 각 블록의 inode bit map과 data bit map 초기화
 - 각 블록의 inode table 초기화
 - root directory 생성
 - lost+found directory 생성

File System 연결(mount)

논리적인 파일 시스템



File System 연결(mount)

- 기준점 설정

- root file system
- 커널이 항상 접근 가능
- 꼭 필요한 시스템 명령(eg: ifconfig)들은 root file system에 존재

- 파일 시스템 연결

- mount: integrated into the directory hierarchy of the root file system
- mounted point

```
# /etc/fstab
# device        directory      type    options
#
/dev/hda1       /              ext2    defaults
/dev/hdb2       /home          ext2    defaults
/dev/hdb1       none           swap    sw
/proc           /proc          proc    defaults
```

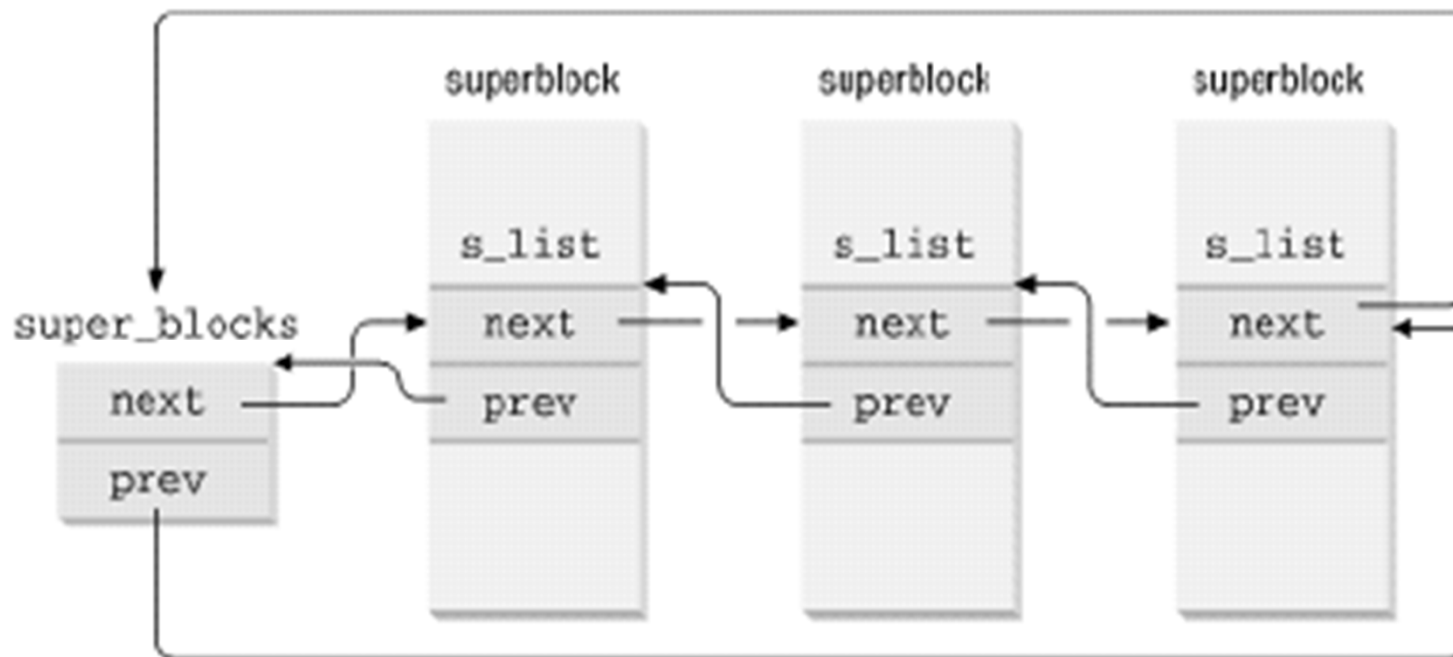
File System 연결(mount)

Virtual file system

mount 명령

```
mount -t msdos /dev/fd0 /mnt
```

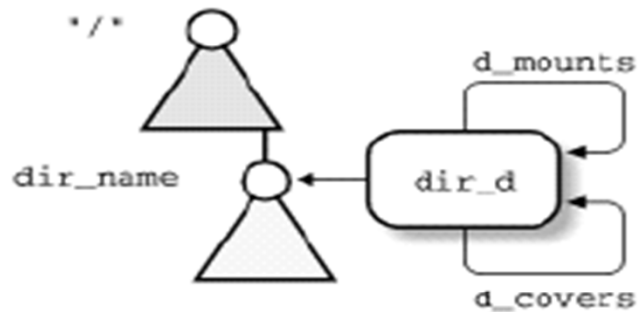
mount 결과



File System 연결(mount)

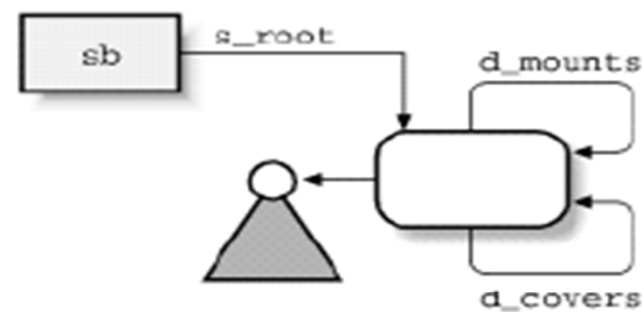
- mount 예

System's Directory Tree Before Mounting



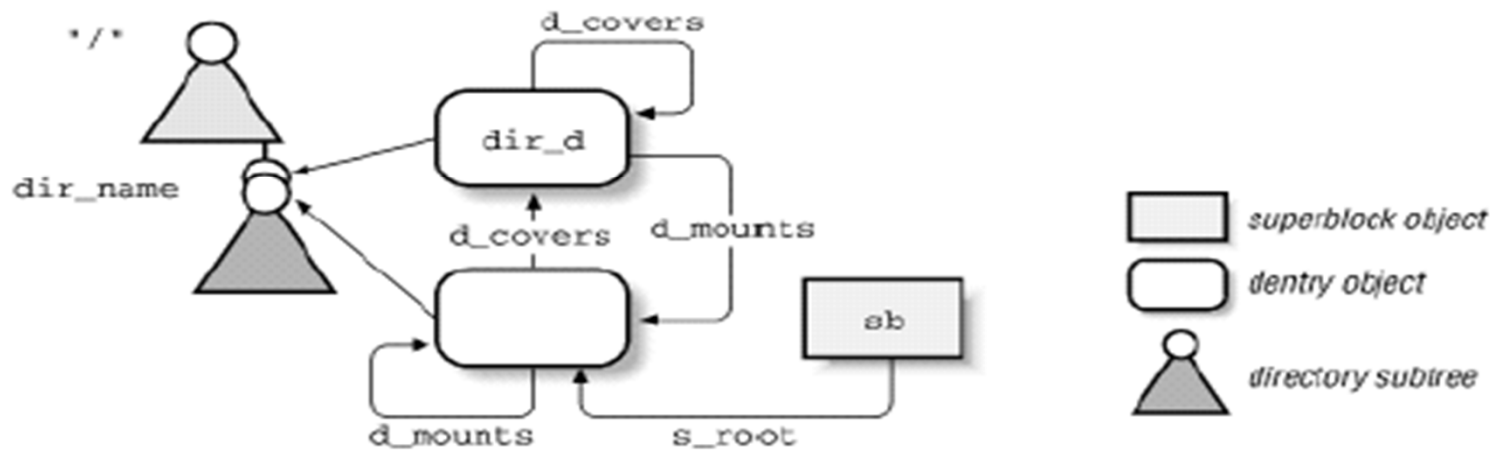
(a)

File System to Be Mounted



(b)

System's Directory Tree After Mounting



(c)

