# 5장 파일 시스템

# 파일 시스템

- df 혹은 du와 같은 명령을 통해
  - 사용자 혹은 관리자 입장에서
  - 파일시스템 정보 혹은 디스크 사용 정보를 획득

- 파일 시스템
  - 물리적으로 디스크 혹은 다양한 저장장치들로 구성된 저장 공간을 논리적 형태로 (운영체제가) 변환시켜 관리하며,
    사용자가 이를 접근하여 사용할 수 있도록 해 준다.

- 이와 같은 파일 시스템은 내부적으로 어떠한 구조를 가지고 있는가?

# 파일 시스템 보기

- 사용법

```
$ df 파일시스템*
```

파일 시스템에 대한 디스크 사용 정보를 보여준다.

- 예

```
$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/mapper/root 51606140  7570736 41413964 16%    /
tmpfs             1030972      676  1030296   1%    /dev/shm
/dev/sda1          495844    29048  441196    7%    /boot
/dev/mapper/home 424544656  3577668 399401344 1%    /home
```

  - /  루트 파일 시스템 현재 16% 정도 사용
  - /dev/shm  시스템의 가상 공유 메모리를 위한 파일 시스템
  - /boot  리눅스 커널의 메모리 이미지와 부팅을 위한 파일 시스템
  - /home  여러 사용자들의 홈 디렉터리를 위한 파일 시스템

# 디스크 사용량 보기

- 사용법

```
$ du [-s] 파일명*
```

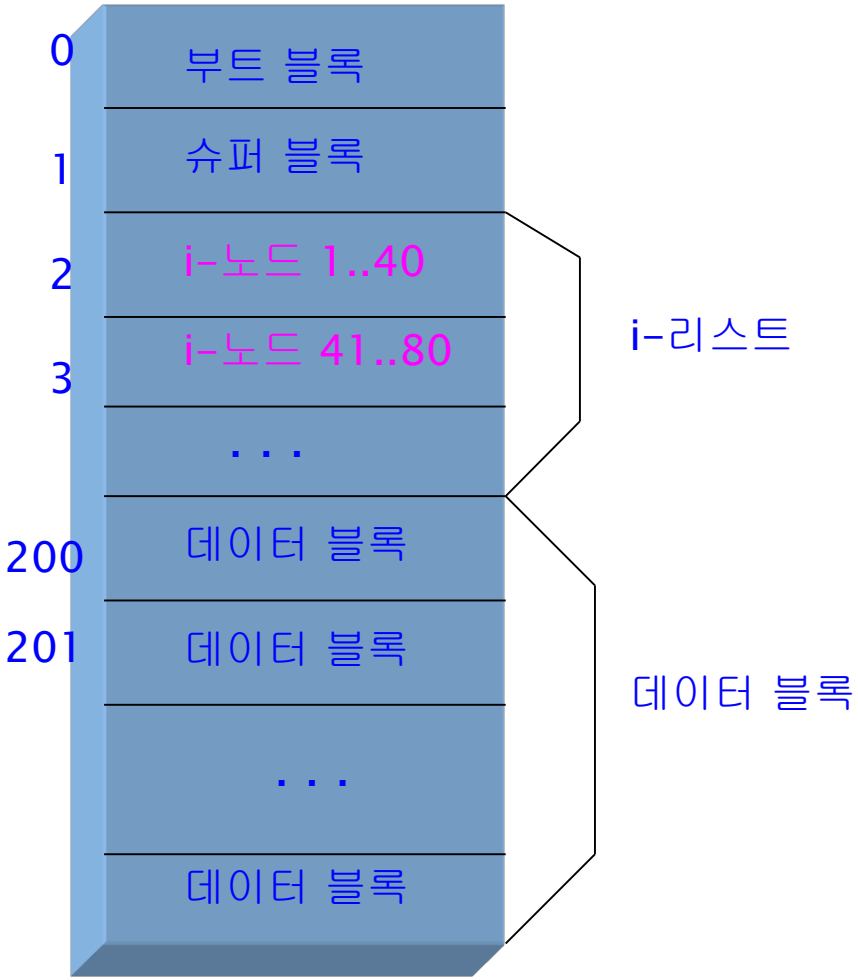
파일 혹은 디렉터리의 사용량을 보여준다. 파일을 명시하지 않으면 현재 디렉터리 내의 모든 파일들의 사용 공간을 보여준다.

- 예

```
$ du
12 ./htdocs/graphics
258 ./htdocs/images
42 ./htdocs/lecture/math
2582 ./htdocs/lecture/sp/lab
33196 ./htdocs/lecture/sp
...
```

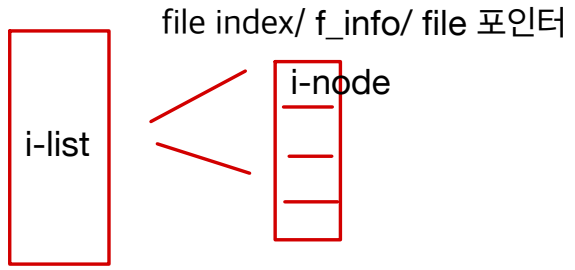
# 5.1 파일 시스템 구현

# 파일 시스템 구조


0
부트 블록
1
슈퍼 블록
2
i-노드 1..40
3
i-노드 41..80
. . .
200
데이터 블록
201
데이터 블록
. . .
데이터 블록
i-리스트
데이터 블록

# 파일 시스템 구조

- 부트 블록(Boot block)
  - 파일 시스템 시작 부분에 위치하고 보통 첫 번째 섹터에 위치
  - 부트스트랩 코드가 저장되는 블록
- 슈퍼 블록(Super block)
  - 전체 파일 시스템에 대한 정보를 저장
    - 총 블록 수, 사용 가능한 i-노드 개수, 사용 가능한 블록 비트 맵, 블록의 크기, 사용 중인 블록 수, 사용 가능한 블록 수 등
- index i-리스트(i-list)
  - 각 파일을 나타내는 모든 i-노드들의 리스트 (i-node vs. i-list)
  - 한 블록은 약 40개 정도의 i-노드를 포함
- 데이터 블록(Data block)
  - 실제 파일의 내용(데이터)을 저장하기 위한 블록들



# i-노드(i-node)

- 하나의 파일은 하나의 i-노드를 갖는다.
  - 리눅스에서 각 파일은 i-노드라 불리는 구조에 의해서 표현
  - c.f., i-list vs. 윈도우에서의 FAT (File Allocation Table)

- 파일에 대한 모든 정보를 가지고 있음
  - 파일 타입: 일반 파일, 디렉터리, 블록 장치, 문자 장치 등
  - 파일 크기
  - 사용권한
  - 파일 소유자 및 그룹
  - 접근 및 갱신 시간
  - 데이터 블록에 대한 포인터(파일의 위치 정보) 등

  - ls –l 명령어의 출력 결과들이 i-node에서 읽어 온 정보임

# i-노드(i-node)

- i-node가 제공하는 파일의 상태 정보

| 파일 상태 정보 | 의미 |
|---|---|
| 파일 크기 | 파일의 크기(K 바이트 단위) |
| 파일 종류 | 파일 종류를 나타낸다. |
| 접근권한 | 파일에 대한 소유자, 그룹, 기타 사용자의 읽기/쓰기/실행 권한 |
| 하드 링크 수 | 파일에 대한 하드 링크 개수 |
| 소유자 및 그룹 | 파일의 소유자 ID 및 소유자가 속한 그룹 |
| 파일 크기 | 파일의 크기(바이트 단위) |
| 최종 접근 시간 | 파일에 최후로 접근한 시간 |
| 최종 수정 시간 | 파일을 생성 혹은 최후로 수정한 시간 |
| 데이터 블록 주소 | 실제 데이터가 저장된 데이터 블록의 주소 |

# 데이터 블록 포인터 (i-node 내부에 위치)

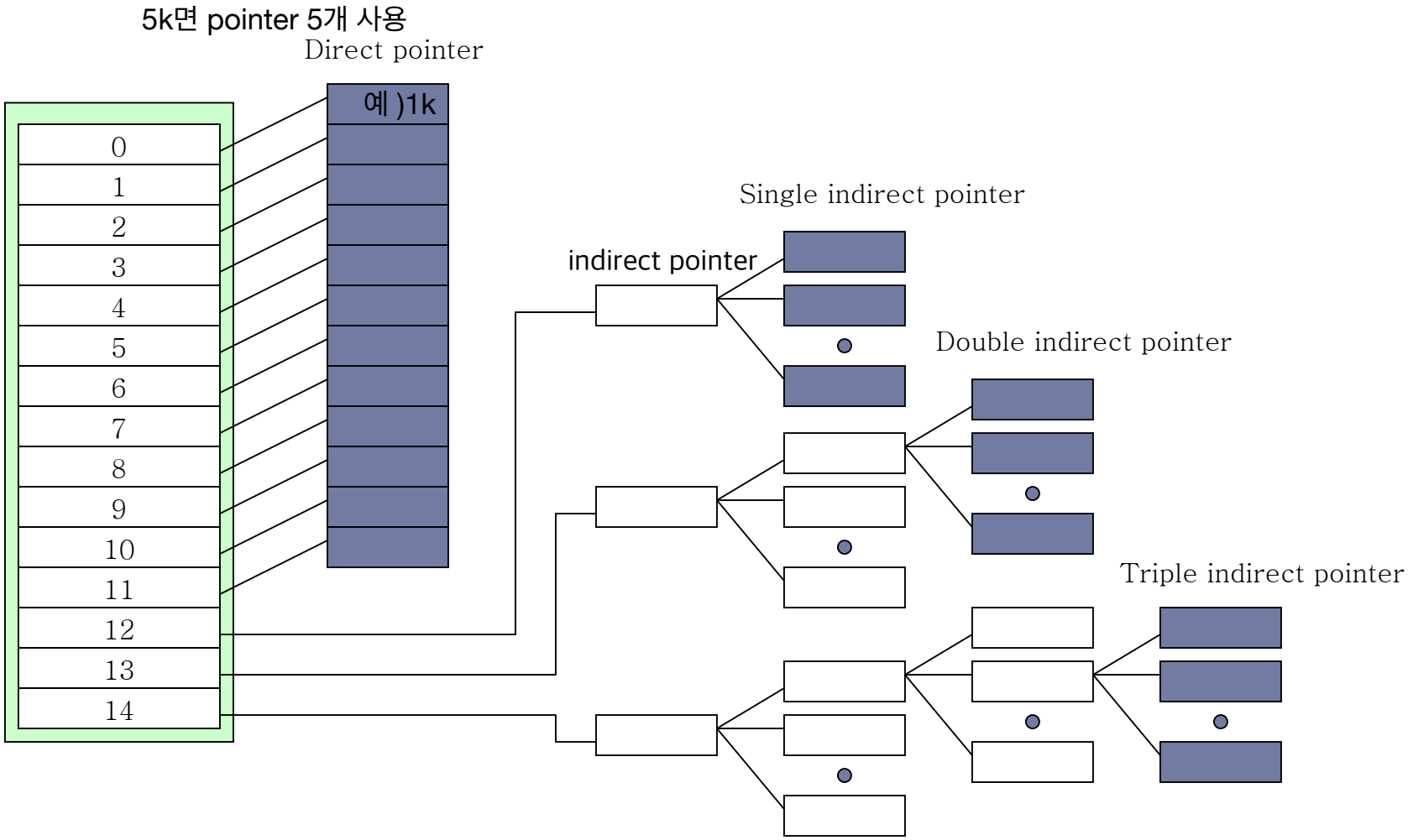
- i-node의 중요한 기능
  - 실제 데이터가 저장된 블록의 위치 정보를 관리
- 데이터 블록에 대한 포인터
  - 파일의 내용을 저장하기 위해 할당된 데이터 블록의 주소
- 하나의 i-노드 내의 블록 포인터 구조
  - 직접 블록 포인터 12개
  - 간접 블록 포인터 1개
  - 이중 간접 블록 포인터 1개
  - 삼중 간접 블록 포인터 1개

Block size 4KB = 4096 bytes
Pointer size 4 bytes

| 블록수 | 데이터 크기 |
| --- | --- |
| 12block | 4KB * 12 direct pointers |
| 4KB / 4bytes = 1024 | 4KB * 1024 indirect pointers |
| 1024 * 1024 = 1,048,576 | 4KB * 1024 * 1024 |
| 1024 * 1024 * 1024 = 1,073,741,824 | 4KB * 1024 * 1024 * 1024 |

- 최대 몇 개의 데이터 블록을 가리킬 수 있을까?

# i-node 내부의 블록 포인터 구조 상세



# 블록 포인터 구조

- 최대 파일의 크기

-Assumption : 8KB block size and 4 byte pointer size (then, each block has 2048 pointers (8KB/4byte))

간접 블록 포인터 8KB/4B = 2048

96KB(8KB *12 direct pointers)
+ 16,384KB(8KB * 2048 indirect pointers)
+ 33,554,432KB(8KB * 2048 * 2048)
+ 68,719,476,736KB (8KB * 2048 * 2048 * 2048)
= about 64TB

i-list의 i-node 항목 수에 따라 결정되며, 운영 체제가 관리할 수 있는 파일 수에 제한이 생긴다

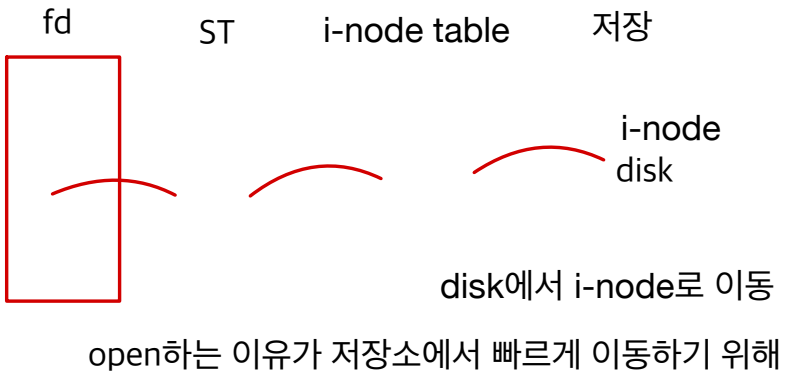
→Depends on the number of i-node entries in i-list, Restriction on the number of files held by OS.

→Depends on the number of pointers and their structures, Restriction on the largest file size.

포인터의 수와 구조에 따라 결정되며, 파일의 최대 크기에 제한이 생긴다

# 파일 입출력 구현

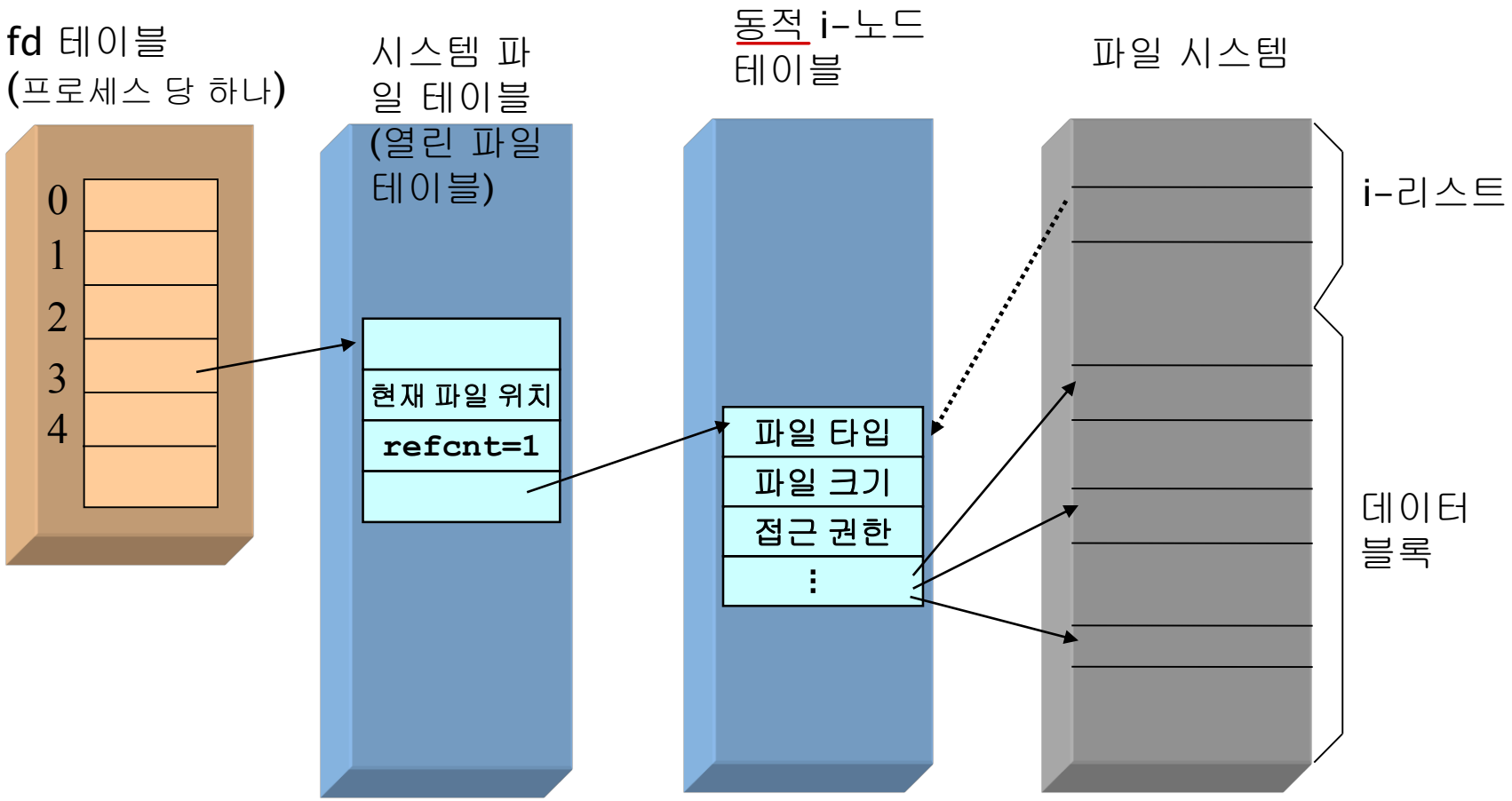
- 파일 입출력 구현을 위한 커널 내부의 파일 시스템 자료 구조
  - 파일 디스크립터 테이블 (fd Table)
    - 파일 디스크립터 배열 형태의 테이블
    - 각 프로세스마다 하나씩 존재




  - (시스템) 파일 테이블 혹은 열린 파일 테이블(open File Table)
    - 시스템 레벨에서 전역적으로 하나 존재
  - 동적 i-노드 테이블(active i-node table or incore i-node table)
    - 시스템 레벨에서 전역적으로 하나 존재

# 파일을 위한 커널 자료 구조

- fd = open("file", O_RDONLY);



# 파일 디스크립터 테이블(FD Table)

- 프로세스 당 하나씩 갖는다.
  - Per-process data structure
- 파일 디스크립터 배열이라고도 한다.
  - 해당 프로세스가 open 한 파일의 디스크립터 (fd) 저장 공간
  - 시스템 파일 테이블(열린 파일 테이블)로의 엔트리를 가리킨다.
- 파일 디스크립터 (fd)
  - 파일 디스크립터 테이블의 인덱스
  - 열린 파일을 나타내는 번호
  - open() 함수의 반환값
    - 즉, pathname이 변환되어 반환된 값
  - 프로그래머가 해당 파일에 접근하기 위해 사용되는 값

# 시스템 파일 테이블 (열린 파일 테이블)

- 시스템 파일 테이블
  - System-wide Kernel Data Structure
  - 열려진 모든 파일 목록을 저장하기 위한 공간
  - 파일을 열 때마다, 파일 테이블의 엔트리가 하나씩 생성

- 시스템 파일 테이블 항목
  - 파일 상태 플래그

    (read, write, append, sync, nonblocking,…)
  - 파일의 현재 위치 (current offset)
  - 파일의 Reference counter
    - Reference Counter의 역할과 파일 close() 함수와의 관계?
  - 동적 i-node 테이블의 해당 엔트리를 가리킨다

# 동적 i-노드 테이블(Active i-node table)

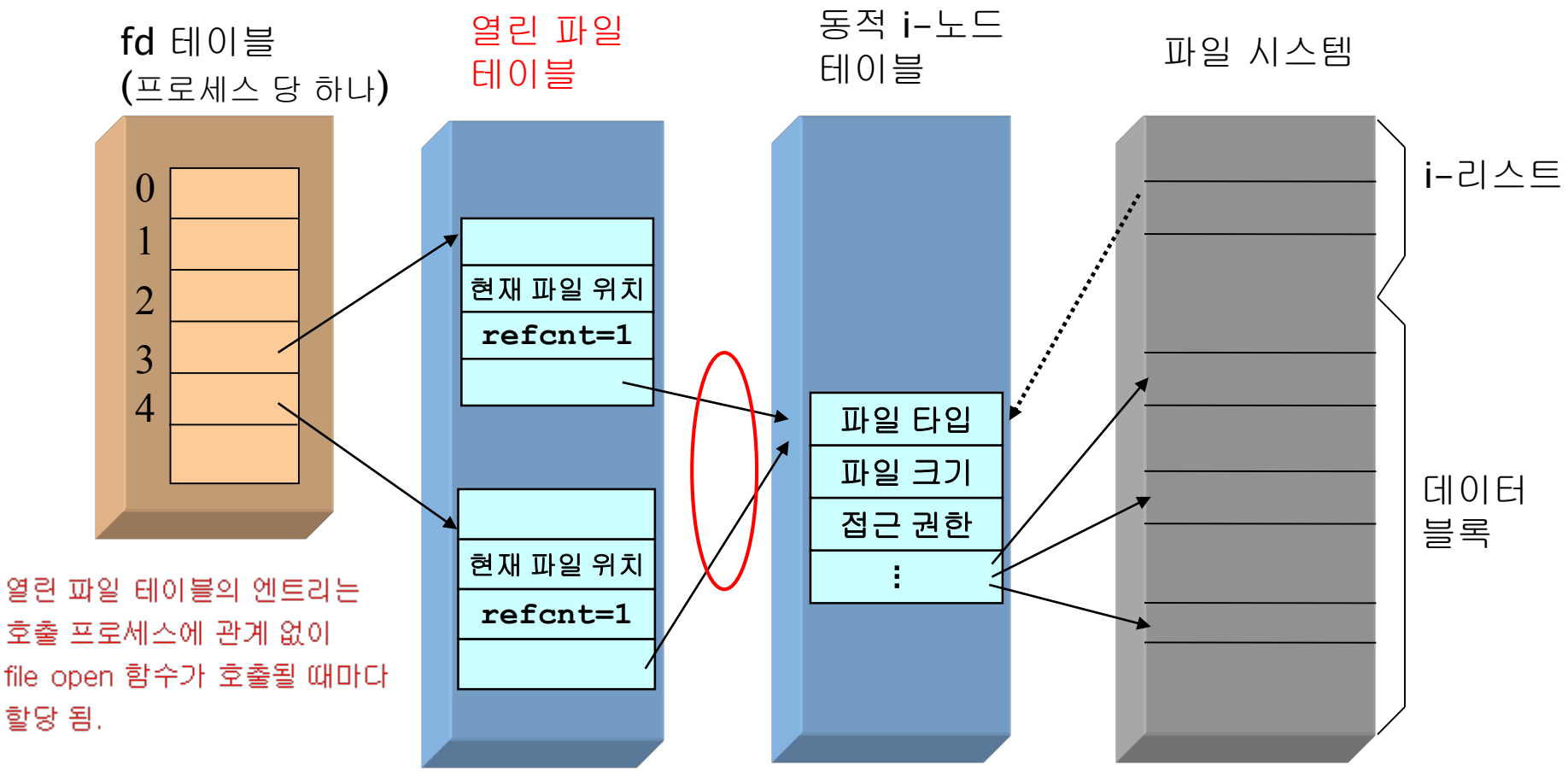
- 동적 i-노드 테이블
  - System-wide Kernel Data Structure
  - Open 된 파일들의 i-node를 별도로 저장하기 위한 테이블
    - 즉, 파일을 열면 파일 시스템 내에서 i-node 내용을 가져와 메모리 상의 동적 i-node 테이블 엔트리로 만든다.

- (review) i-노드란?
  - 하드 디스크에 저장되어 있는 파일에 대한 자료구조
    - 디스크에 저장된 파일의 정보(meta data)를 저장하고 있는 형태
  - 하나의 파일에 하나의 i-node
  - 하나의 파일에 대한 모든 정보 저장
    - 소유자, 크기
    - 파일이 위치한 장치
    - 파일 내용 디스크 블럭에 대한 포인터 등

# 동작 사례 1

- fd = open("file", O_RDONLY);

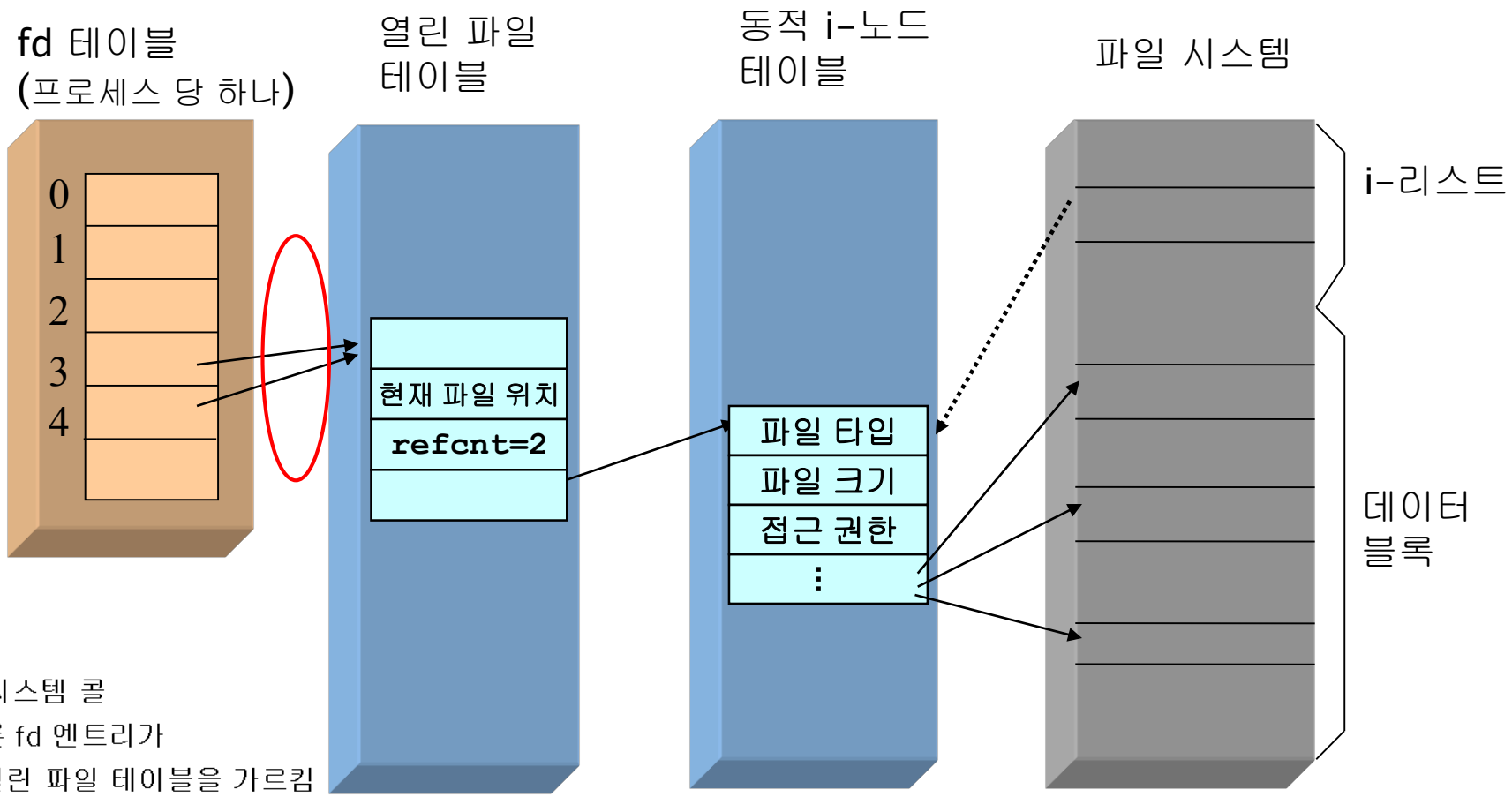
하나의 파일을 두 번 열 때의 커널 자료구조



# 동작 사례 2

- fd = dup(3); 혹은 fd = dup2(3,4);

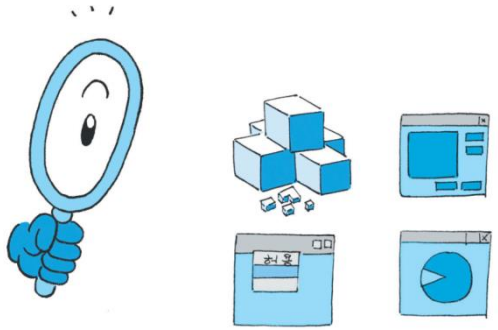
dup() 시스템 호출 후의 자료 구조




dup( ) 시스템 콜
서로 다른 fd 엔트리가
하나의 열린 파일 테이블을 가르킴

# 5.2 파일 상태 정보

# 파일 상태(file status)



- 파일 상태
  - 파일에 대한 모든 정보
  - 블록 수, 파일 타입, 접근 권한, 링크 수, 파일 소유자의 사용자 ID, 그룹 ID, 파일 크기, 최종 수정 시간 등의 정보
  - ls – l 명령어가 이와 같은 파일의 상태를 보여주는 명령어

- 예

```
$ ls -l hello.c
2     -rw-r--r-- 1    chang  cs   617    11월 17일 15:53 hello.c
```

블록수 | 사용권한 | 링크수 | 사용자ID | 그룹ID | 파일 크기 | 최종 수정 시간 | 파일이름

파일 타입

# stat 명령어 : 파일 상태를 출력하는 명령어

- 사용법

```
$ stat [옵션] 파일
```

파일의 자세한 상태 정보를 출력한다.

- 예

```
$ stat cs1.txt
File: `cs1.txt'
Size: 2088   Blocks: 8 IO Block: 4096 일반 파일
Device: fd02h/64770d Inode: 268456513 Links: 1
Access: (0664/-rw-rw-r--) Uid: ( 1000/chang) Gid: ( 1002/cs)
Context: system_u:object_r:user_home_t:s0
Access: 2016-10-31 17:09:23.082488375 +0900
Modify: 2012-10-23 12:51:04.000000000 +0900
Change: 2016-10-04 09:17:212.543444408 +0900
Birth: -
```

# stat() 함수 : 파일 상태를 출력하는 시스템 호출 함수

- 파일 하나당 하나의 i-노드가 있으며,
  i-노드 내부에 파일에 대한 모든 상태 정보가 저장되어 있다.
- i-노드에 저장되어 있는 상태정보를 가져오는 역할

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *filename, struct stat *buf);  (파일이름으로지정)
int fstat (int fd, struct stat *buf);   (대상 파일을 fd로 지정)
int lstat (const char *filename, struct stat *buf);  (링크 자체의 stat)
```

기본적으로 stat()과 동일, 단 대상 파일이 심볼릭 링크일 경우, 링크가 가리키는 파일이 아니라 링크 자체에 대한 정보를 가져옴

파일의 상태 정보를 가져와서 **stat 구조체 buf에 저장**. 성공하면 0, 실패하면 -1을 리턴한다.

# stat 구조체

- stat( ) 시스템 호출은 i-node로 부터 파일의 정보를 가져와 **stat 구조체**에 저장하는 역할을 수행 함

```
struct stat {
    mode_t st_mode;        // 파일 타입(slide 27~28)과 사용권한(slide 31)
    ino_t st_ino;          // i-노드 번호
    dev_t st_dev;          // 장치 번호
    dev_t st_rdev;         // 특수 파일 장치 번호
    nlink_t st_nlink;      // 링크 수
    uid_t st_uid;          // 소유자의 사용자 ID
    gid_t st_gid;          // 소유자의 그룹 ID
    off_t st_size;         // 파일 크기
    time_t st_atime;       // 최종 접근 시간 (slide 35~37)
    time_t st_mtime;       // 최종 수정 시간 (slide 35~37)
    time_t st_ctime;       // 최종 상태 변경 시간
    long st_blksize;       // 최적 블록 크기
    long st_blocks;        // 파일의 블록 수
};
```

# 파일 타입

리눅스에서 지원하는 파일의 타입

| 파일 타입 | 설명 |
|---|---|
| 일반 파일 | 데이터를 갖고 있는 텍스트 파일 또는 이진 파일 |
| 디렉터리 파일 | 파일의 이름들과 파일 정보에 대한 포인터를 포함하는 파일 |
| 문자 장치 파일 | 문자 단위로 데이터를 전송하는 장치를 나타내는 파일 |
| 블록 장치 파일 | 블록 단위로 데이터를 전송하는 장치를 나타내는 파일 |
| 파이프 (FIFO 파일) | 커널 내의 프로세스 간 통신에 사용되는 파일 |
| 소켓 | 네트워크를 통한 프로세스 간 통신에 사용되는 파일 |
| 심볼릭 링크 | 다른 파일을 가리키는 포인터 역할을 하는 파일 |

# 파일 타입 검사 함수

- 파일 타입을 검사하기 위한 매크로 함수
  - struct stat 구조체 st_mode 필드 내부를 조사하는 대신(slide 31), 다음과 같은 매크로 함수를 통해 해당 값을 확인하는 것도 가능

| 파일 타입 | 파일 타입을 검사하기 위한 매크로 함수 |
|---|---|
| 일반 파일 | S_ISREG()  // stat_is_regular? |
| 디렉터리 파일 | S_ISDIR() |
| 문자 장치 파일 | S_ISCHR() |
| 블록 장치 파일 | S_ISBLK() |
| FIFO 파일 | S_ISFIFO() |
| 소켓 | S_ISSOCK() |
| 심볼릭 링크 | S_ISLNK() |

# ftype.c

파일의 이름을 주면 타입을 출력하는 프로그램
$ftype /bin
/bin: 디렉터리

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
/* 파일 타입을 검사한다. */
int main(int argc, char *argv[])
{
    int i;
    struct stat buf;                    //stat 구조체 타입의 일반 변수 선언
    for (i = 1; i < argc; i++) {        // argc만큼 반복 출력 가능 (인자로 복수개의 파일 가능)
        printf("%s: ", argv[i]);         // 예시에서의 "/bin:" 출력 역할
        if (lstat(argv[i], &buf) < 0) {      //파일의 상태 정보를 stat 구조체 변수
            perror("lstat()");                //buf에 저장
            continue;
        } //if
```

# ftype.c

```
        if (S_ISREG(buf.st_mode))                    /* st_mode : slide 24 */
            printf("%s ₩n", "일반 파일");
        if (S_ISDIR(buf.st_mode))
            printf("%s ₩n", "디렉터리");
        if (S_ISCHR(buf.st_mode))
            printf("%s ₩n", "문자 장치 파일");
        if (S_ISBLK(buf.st_mode))
            printf("%s ₩n", "블록 장치 파일");
        if (S_ISFIFO(buf.st_mode))
            printf("%s ₩n", "FIFO 파일");
        if (S_ISLNK(buf.st_mode))
            printf("%s ₩n", "심볼릭 링크");
        if (S_ISSOCK(buf.st_mode))
            printf("%s ₩n", "소켓");
    }  // for
    exit(0);
}
```

# 파일 사용권한(File Permissions)

- 각 파일에 대한 권한 관리
  - 각 파일마다 사용권한이 있다.
  - 소유자(owner)/그룹(group)/기타(others)로 구분해서 관리한다.

- 파일에 대한 사용 권한
  - 읽기 r
  - 쓰기 w
  - 실행 x

# 사용권한

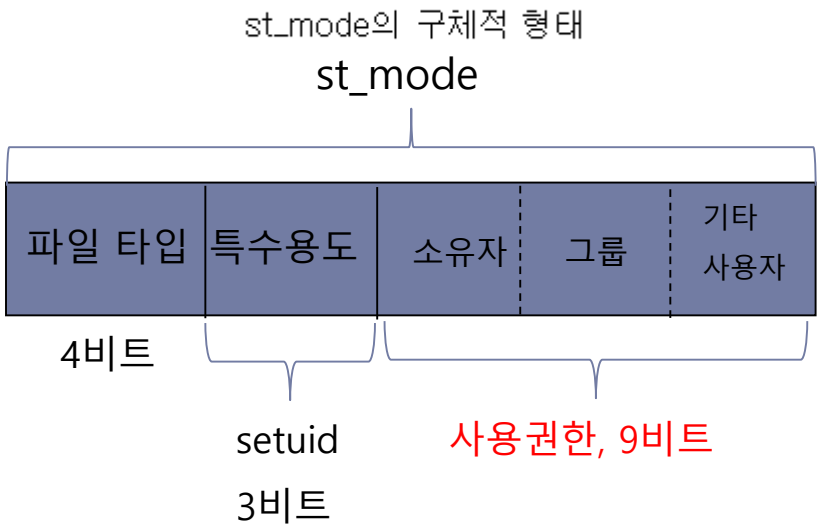
- 파일에 대한 read 권한이 있어야
  - O_RDONLY 혹은 O_RDWR을 사용하여 파일을 열 수 있다
- 파일에 대한 write 권한이 있어야
  - O_WRONLY 혹은 O_RDWR을 사용하여 파일을 열 수 있다
- 디렉터리에 write 권한이 있어야
  - 그 디렉터리에 파일을 생성할 수 있고
  - 그 디렉터리의 파일을 삭제할 수 있다

# 파일 사용권한

- 파일 사용 권한(file access permission)
  - stat 구조체의 st_mode 의 값 (파일 타입과 사용 권한을 가짐)

S_IRUSR : stat_is_read_user

#include <sys/stat.h>




Chmod/umask의 명령어 결과가 이곳에 저장됨

| st_mode mask | Meaning |
|---|---|
| S_IRUSR | user-read |
| S_IWUSR | user-write |
| S_IXUSR | user-execute |
| S_IRGRP | group-read |
| S_IWGRP | group-write |
| S_IXGRP | group-execute |
| S_IROTH | other-read |
| S_IWOTH | other-write |
| S_IXOTH | other-execute |

# chmod(), fchmod()

명령어 vs. 시스템 호출

```
#include <sys/stat.h>
#include <sys/types.h>
int  chmod (const char *path, mode_t mode );  //파일 이름 사용
int  fchmod (int fd, mode_t mode );           //fd 사용
```

- 파일의 사용 권한을 변경하는 시스템 호출 함수

- 리턴 값
  - 성공하면 0, 실패하면 -1

기본 000과 신규 new 모드가 주어지면
이들 사이에 논리합 연산 수행

- *mode* 값 12비트 (3 setuid + 9 ugo) 각 영역
  - S_IRUSR, S_IWUSR, S_IXUSR  //user R W X
  - S_IRGRP, S_IWGRP, S_IXGRP  //group R W X
  - S_IROTH, S_IWOTH, S_IXOTH  //other R W X
  - S_ISUID, S_ISGID+ sticky bit  //set uid/gid (특수용도 3비트)

# fchmod.c

명령어 사용 사례
$ fchmod 664 you.txt

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
/* 파일 사용권한을 변경한다. */
main(int argc, char *argv[])
{
    long strtol( );                    // 함수 프로토타입 선언 (생략해도 무관)
    int newmode;
    newmode = (int) strtol(argv[1], (char **) NULL, 8);  (문자열을 8진수 정수형태로 변환)
    if (chmod(argv[2], newmode) == -1) {      // chmod()는 system call 함수
        perror(argv[2]);
        exit(1);
    }
    exit(0);
}
```

문자열을 정수로
String to Long integer

시스템 프로그램을 통한 새롭게 작성된 명령어 사례

# (참고) chown()

```
#include <sys/types.h>
#include <unistd.h>
int   chown (const char *path, uid_t owner, gid_t group );
int   fchown (int filedes, uid_t owner, gid_t group );
int   lchown (const char *path, uid_t owner, gid_t group );
```

- 파일의 user ID와 group ID를 변경한다.
- 리턴
  - 성공하면 0, 실패하면 -1

- lchown()은 심볼릭 링크 자체의 소유자를 변경한다
  - 심볼릭 링크가 가리키는 대상 파일의 소유자 변경이 아님
- 파일 소유자 혹은 super-user만 변환 가능

# (참고) utime()

```
#include <sys/types.h>
#include <utime.h>                                    다음 슬라이드 구조체
int utime (const char *filename, const struct utimbuf *times );
```

- 파일의 최종 접근 시간과 최종 변경 시간을 조정한다.
- *times*가 NULL 이면, 현재시간으로 설정된다.
- 리턴 값
  - 성공하면 0, 실패하면 -1

- const struct utimbuf 사용 (next slide)

# (참고) utime()

```
struct utimbuf {
      time_t  actime;       /* access time */
      time_t  modtime;      /* modification time */
}
```

- 각 필드는 1970-1-1 00:00 부터 현재까지의 경과 시간을 초로 환산한 값

$cptime a.c b.c
a.c의 시간으로 b.c를 설정

# (참고) 예제: cptime.c

```
#include <sys/types.h>      #include <sys/stat.h>      #include <sys/time.h>
#include <utime.h>          #include <stdio.h>         #include <stdlib.h>

int main(int argc, char *argv[])
{
    struct stat buf;        // 파일 상태 저장을 위한 변수
    struct utimbuf time;

    if (argc < 3) {
        fprintf(stderr, "사용법: cptime file1 file2₩n");
        exit(1);
    }

    if (stat(argv[1], &buf) <0) { // 상태 가져오기 (a.c의 상태 가져오기)
        perror( " stat() " );
        exit(-1);
    }

    time.actime = buf.st_atime;       //접근(atime), 수정(mtime) 시간 설정
    time.modtime = buf.st_mtime;

    if (utime(argv[2], &time))    // 접근, 수정 시간 복사 (b.c 파일에 &time 정보 전달)
        perror("utime");
    else exit(0);
}
```

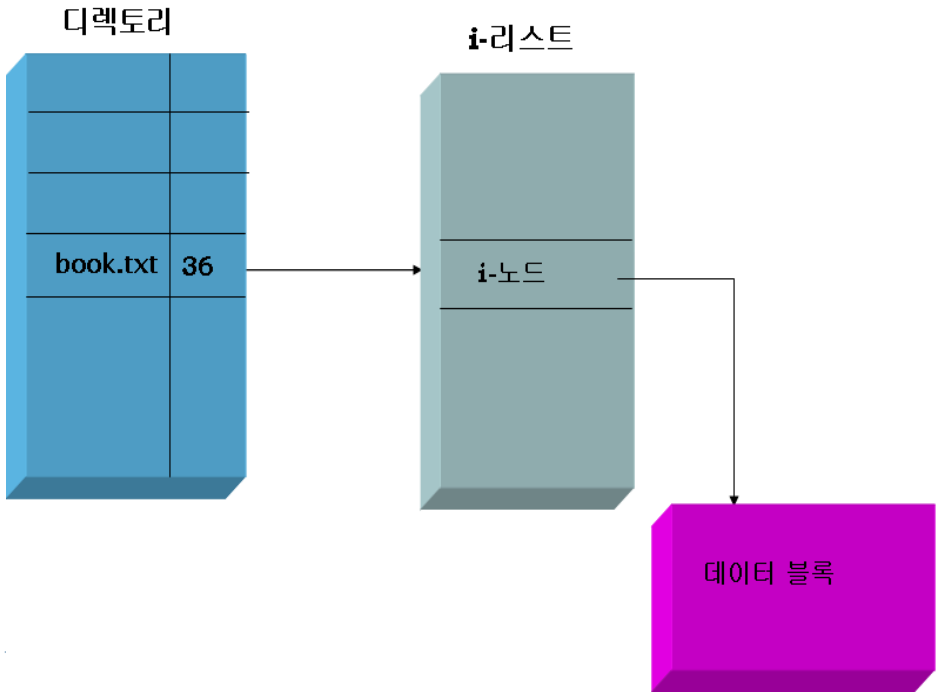
# 5.3 디렉터리

# 디렉터리 구현

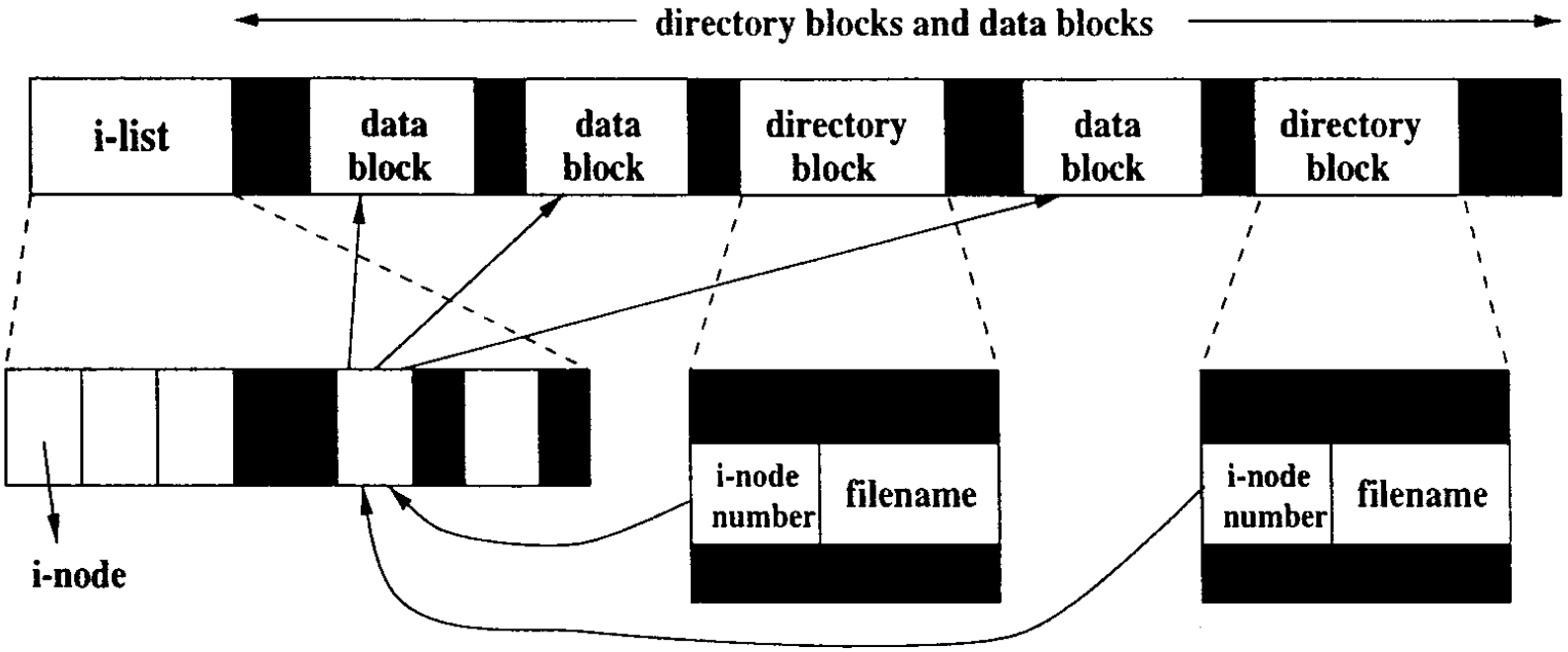
- 디렉토리
  - 여러 파일들과 다른 디렉터리를 조직하는데 사용
  - 디렉터리도 일종의 파일
  - 디렉터리 내에는 무엇이 저장되어 있을까?

- 디렉터리 엔트리 내용
  - 이름과 i-노드 번호 저장

```
#include <dirent.h>
                directory entry
struct dirent {
    ino_t d_ino; // i-노드 번호
    char d_name[NAME_MAX + 1];
                        // 이름
};
```



# 디렉터리 구현

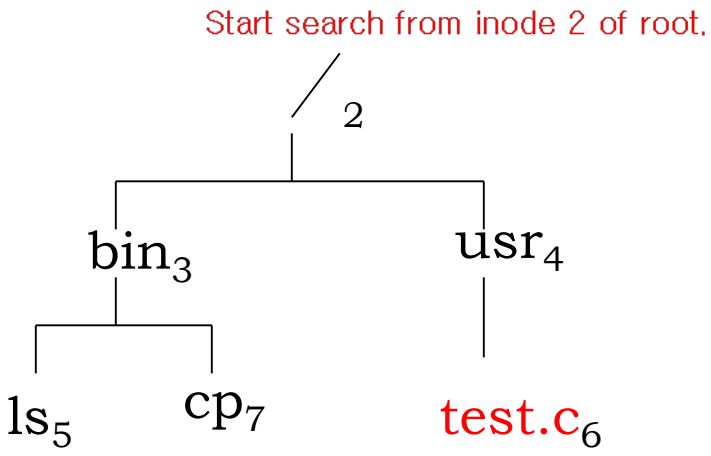

directory blocks and data blocks
i-list
data block
data block
directory block
data block
directory block
i-node
i-node number
filename
i-node number
filename

# 디렉터리 구현

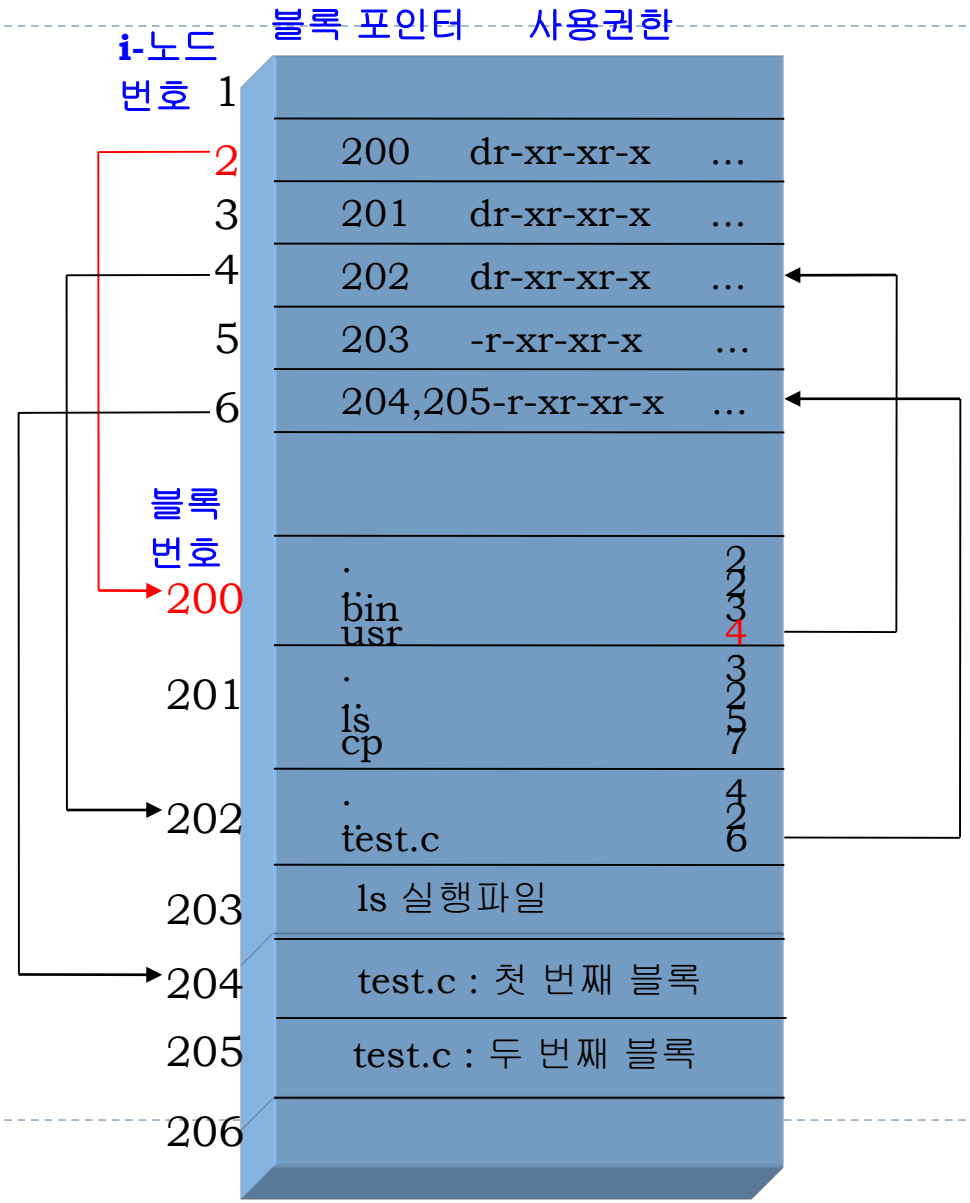
- 디렉터리를 위한 별도의 구조는 없다.
  - 단순히 이름과 inode 번호만 저장
- 파일 시스템 내에서 디렉터리를 어떻게 구현할 수 있을까?
  - 디렉터리도 일종의 파일로 다른 파일과 동일하게 구현된다.
  - 디렉터리도 다른 파일처럼 하나의 i-노드로 표현된다.
  - 디렉터리의 내용은 디렉터리 엔트리 구조체 struct dirent를 참조한다.

# 디렉터리 구현

Inode 0 is used as a NULL value, to indicate that there is no inode.
Inode 1 is used to keep track of any bad blocks on the disk




/usr/test.c 파일의 접근 사례



# 디렉터리 리스트

디렉토리도 파일이므로 기본적으로 open(), read(), close() 함수를 사용할 수 있으나, 별도의 디렉토리만의 편리한 시스템 콜 함수들이 제공

- opendir()
  - 디렉터리 열기 함수
  - 열기 작업 후, DIR 구조체 포인터(열린 디렉터리를 가리키는 포인터) 반환
  - DIR 구조체는 FILE 구조체와 같이 열린 디렉터리에 대한 정보 저장
- readdir()
  - 디렉터리 읽기 함수, 디렉터리 엔트리 내용(즉, 파일들)을 하나씩 읽음

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *path);   // DIR 구조체를 반환
Path 디렉터리를 열고 성공하면 DIR 구조체 포인터를, 실패하면 NULL을 리턴
struct dirent *readdir(DIR *dp);   //인자로 DIR 구조제 변수 사용 (like FILE *fp)
                                   //반환값은 dirent 구조체,  slide 39 */
한 번에 하나씩 디렉터리 엔트리를 읽어서 리턴한다.
```

# list1.c 디렉토리 내부의 파일 이름 출력



```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <dirent.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 /* 디렉터리 내의 파일 이름들을 리스트한다. */
8 int main(int argc, char **argv)
9 {
10    DIR *dp;
11    char *dir;  //디렉터리(문자열 형태) 지정 용도로 사용
12    struct dirent *d;
13    struct stat st;
14    char path[BUFSIZ+1];
15
```

# list1.c 디렉토리 내부의 파일 이름 출력

```
16   if (argc == 1)
17      dir = "."; // 인자 없이 사용할 경우 현재 디렉터리를 대상으로
18   else dir = argv[1];
19
                DIR 구조체          경로
20   if ((dp = opendir(dir)) == NULL) // 디렉터리 열기 ( like fp=fopen() )
21      perror(dir);
22
                dirent 구조체
23   while ((d = readdir(dp)) != NULL) // open한 dp에 대한 dirent 변수 d에 대해
24      printf("%s ₩n", d->d_name);     // 디렉터리 내부 파일 이름을 하나씩 출력
25                                      //dirent 구조체 두번째 필드 : d_name
26   closedir(dp);
27   exit(0);
28 }
```

사용자 : ls 명령어를 사용하는 방법 습득
개발자 : ls 명령어를 개발할 수 있는 능력

# (고급) 파일 이름과 크기도 함께 출력

$ ls –s 명령어의 작성 사례

- list1.c 프로그램을 디렉터리 내에 있는 파일 이름과 그 파일의 크기(블록의 수)를 출력하도록 확장 해 보자.

```
while ((d = readdir(dp)) != NULL) {                 //디렉터리 내의 각 파일에 대해
    sprintf(path, "%s/%s", dir, d->d_name);         // 파일경로명 만들기
                          "디렉터리이름/파일이름"
    if (lstat(path, &st) < 0)                       // 파일 상태 정보 가져오기
        perror(path);
    printf("%5d %s", st.st_blocks, d->d_name);  // 블록 수, 파일 이름 출력
    putchar('₩n');                                  state와 dirent 구조체 활용
}
```

sprintf()
표준출력 대신 문자열로 printf 가능
문자열 path에 출력 저장

# 디렉터리 생성

- mkdir() 시스템 호출
  - path가 나타내는 새로운 디렉터리를 만든다.
  - "." 와 ".." 파일은 자동적으로 만들어진다

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode );
```

새로운 디렉터리 만들기에 성공하면 0, 실패하면 -1을 리턴한다.

사용자 명령어 mkdir이 사용되면 이와 같은 대응 시스템 호출이 사용됨

# 디렉터리 삭제

- rmdir() 시스템 호출
  - path가 나타내는 디렉터리가 비어 있으면 삭제한다.

```
#include <unistd.h>
int rmdir (const char *path);
디렉터리가 비어 있으면 삭제한다. 성공하면 0, 실패하면 -1을 리턴
```

# 5.4 링크

# 링크

- 링크는 기존 파일에 대한 또 다른 이름
  - 링크를 사용하면 하나의 파일에 여러 개의 이름 부여 가능
- 하드 링크와 심볼릭 링크가 있다.
- 하드 링크 생성을 위한 시스템 호출

```
#include <unistd.h>
int link(char *existing, char *new);   // link old new
int unlink(char *path);  //해당 path 삭제 역할
```

- 링크 생성을 위한 명령어

```
$ ln [-s] 파일1 파일2
```

파일1에 대한 새로운 이름(링크)로 파일2를 만들어 준다. -s 옵션은 심볼릭 링크

```
$ ln [-s] 파일1 디렉터리
```

파일1에 대한 링크를 지정된 디렉터리에 같은 이름으로 만들어 준다.

# 하드 링크(hard link)

- 하드 링크
  - 기존 파일에 대한 새로운 이름이라고 생각할 수 있다.
  - 실제로 기존 파일을 대표하는 **"같은 하나의 i-노드"**를 가리켜 구현한다.

i-node가 하나이기 때문에 결국 같은 파일임

- 예

```
$ ln hello.txt hi.txt
$ ls -l
-rw------- 2 chang cs 15 11월 7일 15:31 hello.txt
-rw------- 2 chang cs 15 11월 7일 15:31 hi.txt
```

- 확인

```
$ ls -i hello.txt hi.txt
537384090 hello.txt 537384090 hi.txt
```

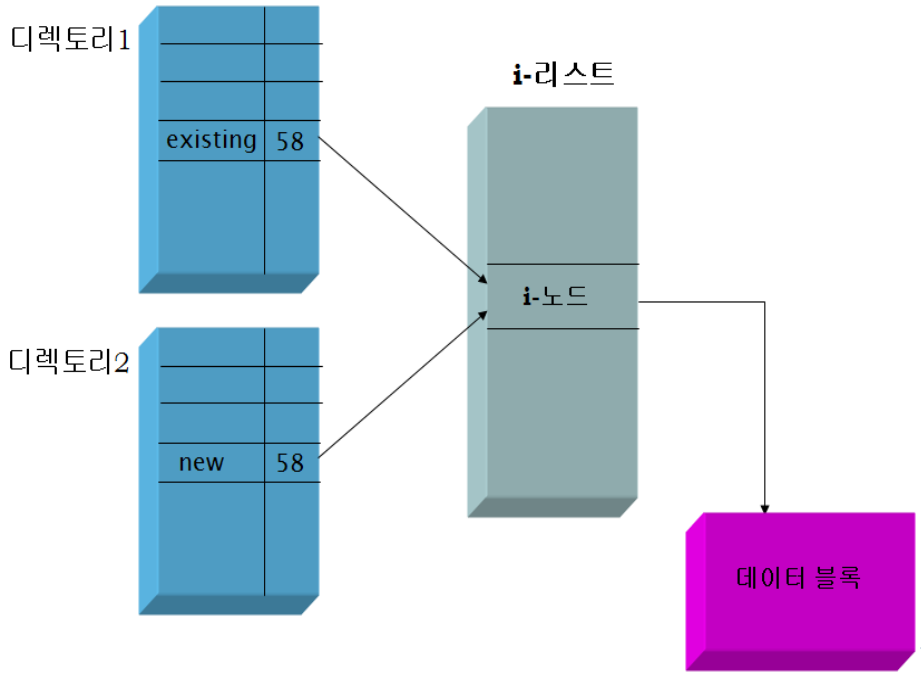
← 같은 i-number를 가짐을 확인

```
yu22312072@acslab-146:~/10$ ln -s hello.c hi.txt
yu22312072@acslab-146:~/10$ ls -i hello.c hi.txt
9311236 hello.c 9314677 hi.txt
```

i-node 번호가 같다는 의미는
디스크 상에 하나의 파일이라는 의미

# 링크의 구현

- link() 시스템 호출
  - 기존 파일 existing에 대한 새로운 이름 new 즉 링크를 만든다.
  - 새롭게 만들어진 링크는 동일한 i-node 번호를 가진다.
  - i-node가 하나이기 때문에, 실제 파일은 하나만 존재하는 것이며, 이름을 하나 더 만들었다고 생각할 것



# link.c

```
#include <unistd.h>
int main(int argc, char *argv[ ])
{
```

하드링크

```
   if (link(argv[1], argv[2]) == -1) {    // link (old, new)
      exit(1);
   }
   exit(0);
}
```

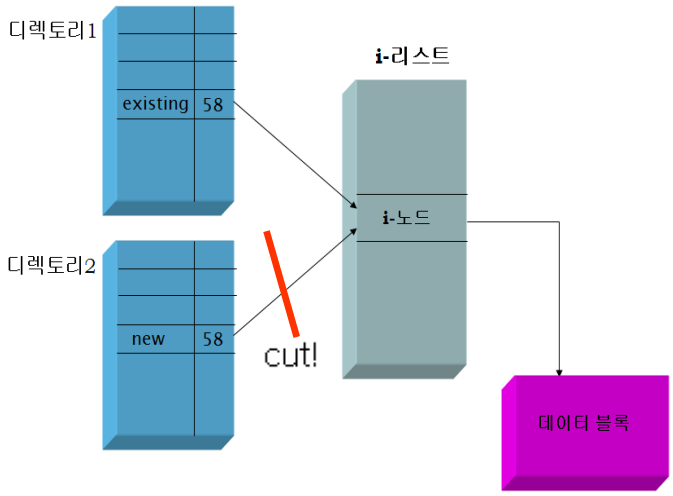
링크를 추가할 때 마다 link count가 증가됨

명령어 레벨에서의 사용 예제

```
$ ls  file
-rw-r—r--  1  user1  2007   50  4월10일 12:00  file
$ ln  file  file2
-rw-r—r--  2  user1  2007   50  4월10일 12:00  file
-rw-r—r--  2  user1  2007   50  4월10일 12:00  file2
```

# unlink.c

```
#include <unistd.h>
main(int argc, char *argv[ ])
{
   int unlink( );
   if (unlink(argv[1]) == -1 {      // link 파일의 삭제, 즉 해당 path name 삭제,
      perror(argv[1]);              //link count가 0보다 크면 파일은 여전히 존재
      exit(1);
   }
   exit(0);
}
```




링크를 삭제할 때마다, link count를 감소시킴

# 심볼릭 링크(symbolic link)

- 심볼릭 링크
  - 다른 파일을 가리키고 있는 별도의 파일이다. (즉, 서로 다른 i_no)
  - 실제 파일의 경로명을 저장하고 있는 일종의 특수 파일이다.
  - 이 경로명이 다른 파일에 대한 간접적인 포인터 역할을 한다.

- 예

```
$ ln -s hello.txt hi.txt   (기존 hello에 hi라는 신규 링크 생성)
$ ls -l
-rw------- 1 chang cs 15 11월 7일 15:31 hello.txt
lrwxrwxrwx 1 chang cs 9 1월 24일 12:56 hi.txt -> hello.txt

$ ln -s /usr/bin/gcc cc
$ ls -l cc
lrwxrwxrwx. 1 chang chang 12 7월 21 20:09 cc -> /usr/bin/gcc
```

# 심볼릭 링크 생성 시스템 호출

int symlink (const char *actualpath, const char *sympath );

심볼릭 링크를 만드는데 성공하면 0, 실패하면 -1을 리턴한다.

```
#include <unistd.h>
int main(int argc, char *argv[ ])
{
   if (symlink(argv[1], argv[2]) == -1) {    //심볼릭 링크 생성
                                             //symlink(old, new)
      exit(1);
   }
   exit(0);
}
```

# 심볼릭 링크의 대상 경로 내용 확인

```
#include <unistd.h>
int readlink (const char *path, char *buf, size_t bufsize);
```

path 심볼릭 링크의 내용(즉, 원본 파일로의 대상 경로임)을 읽어서 buf에 저장한다. (링크가 가리키는 대상 원본 파일의 실제 경로를 읽음)

성공하면 buf에 저장한 바이트 수를 반환하며 실패하면 –1을 반환한다.

심볼릭 링크의 내용
(= 가리키고 있는 대상 파일의 경로)
를 읽어서 이를 buf에 저장하는 역할

즉, 가리키고 있는 원본 파일의 경로명을 확인하는 역할

# rlink.c

심볼릭 링크를 읽어
가리키는 대상 파일을 출력하는 프로그램

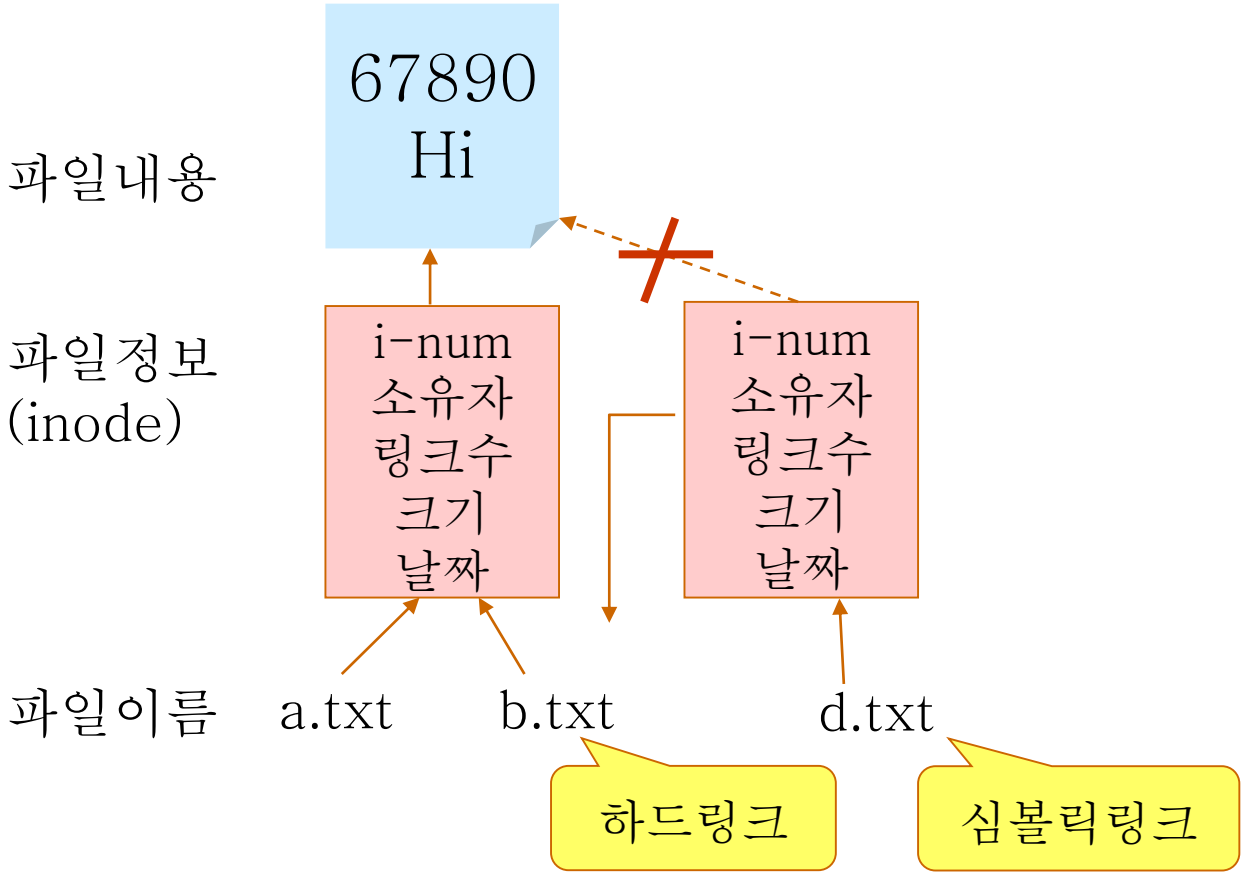
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[ ])
{
    char buffer[1024];
    int nread;
    nread = readlink(argv[1], buffer, 1024);      //cc의 원래 내용을 buffer에 저장
    if (nread > 0) {
        write(1, buffer, nread);                  //buffer의 내용을 nread 만큼 1(표준 출력)에 씀
        exit(0);
    } else {
        fprintf(stderr, "오류 : 해당 링크 없음₩n");
        exit(1);
    }
}
```

사용 예

```
$ rlink cc
/usr/bin/gcc
```

심볼릭 링크 cc 파일의
원본 대상 파일이 usr/bin/gcc 임을 확인

# (정리) 하드 링크 vs. 심볼릭 링크



# (참고) 하드 링크 vs. 심볼릭 링크

- 하드 링크(hard link)
  - 파일 시스템 내의 i-노드를 가리키므로
  - 같은 파일 시스템 내에서만 사용될 수 있다
  - 원본이 삭제되어도 남은 하드링크로 접근 가능
  - 디렉터리에 대한 하드 링크는 슈퍼 유저만 사용 가능

- 심볼릭 링크(symbolic link)
  - 소프트 링크(soft link)라고도 함
  - 실제 파일의 경로명을 데이터 블록에 저장하고 있는 링크
  - 파일에 대한 간접적인 포인터 역할을 한다. (윈도우 시스템의 바로가기)
  - 다른 파일 시스템에 있는 파일도 링크할 수 있다.
  - 원본이 삭제되면 심볼릭 링크로는 접근 불가
  - 디렉터리에 대한 심볼릭 링크는 일반 사용자도 생성 가능

# 핵심 개념

- 표준 유닉스 파일 시스템은 부트 블록, 슈퍼 블록, i-리스트, 데이터 블록 부분으로 구성된다
- 파일 입출력 구현을 위해서 커널 내에 파일 디스크립터 배열, 파일 테이블, 동적 i-노드 테이블 등의 자료구조를 사용한다.
- 파일 하나당 하나의 i-노드가 있으며 i-노드 내에 파일에 대한 모든 상태 정보가 저장되어 있다.
- 디렉터리는 일련의 디렉터리 엔트리들을 포함하고 각 디렉터리 엔트리는 파일 이름과 그 파일의 i-노드 번호로 구성된다.
- 링크는 기존 파일에 대한 또 다른 이름으로 하드 링크와 심볼릭(소프트) 링크가 있다.

- 본 챕터의 주요 시스템 호출을 사용하여 구현된 명령어를 기존의 리눅스 명령어와 비교하여 보자.