8. 파일 시스템과 디렉토리 조작

8.1 파일 시스템의 개념 (inode 등)

개요

파일 시스템(File System) 은 운영체제에서 **데이터를 저장하고 관리하는 논리적 구조**다.
단순히 "파일 이름"과 "내용"만 저장하는 것이 아니라:

- **데이터 구조화**
- **메타데이터 관리**
- **액세스 제어**
- **저장 장치와의 효율적 인터페이스** 를 모두 담당한다.

리눅스에서는 주로 **Unix 계열 파일 시스템 설계 원칙**을 따름.
→ 대표적으로 **inode 기반 구조**.

파일 시스템 구조 개념도

```
+-----------------------------+
| File System (ex: ext4, xfs) |
+-----------------------------+
        |
        V
+-----------------------------+
| Superblock                  | → 파일시스템 전체 정보
+-----------------------------+
| Block Group Descriptors     | → 블록 그룹 관리
+-----------------------------+
| Inode Table                 | → 파일/디렉토리 메타데이터
+-----------------------------+
| Data Blocks                 | → 파일 실제 데이터 저장 영역
+-----------------------------+
```

---

inode란 무엇인가?

inode (index node) 는 **파일 자체를 표현하는 자료구조**다.

파일 이름 ≠ 파일 자체

- 디렉토리 항목: 이름 → inode 번호 매핑
- inode는:
  - **파일의 메타데이터**
  - **파일 데이터 블록의 위치**
    를 저장.

inode에 저장되는 정보

정보 항목	설명
파일 유형	Regular, Directory, Symlink, ...
파일 권한	rwxrwxrwx (mode)
소유자 정보	UID, GID
파일 크기	Byte 단위
생성/수정/접근 시간	ctime, mtime, atime
데이터 블록 포인터	실제 데이터 위치 정보 (직접, 간접, 이중 간접 등)
링크 수	Hard link count

inode에 저장되지 않는 정보

정보	설명
파일 이름	이름은 디렉토리 구조에 저장됨
경로 정보	inode는 해당 파일의 경로를 알지 못함

디렉토리와 inode

디렉토리는 단순히 **"파일 이름 → inode 번호" 쌍의 리스트**이다.

```
Directory entry:
+------------+---------------+
| File name  | Inode number  |
+------------+---------------+
```

따라서:

- 같은 inode 번호를 참조하는 **여러 디렉토리 엔트리 존재 가능 → Hard link**
- 삭제란: 디렉토리 엔트리에서 이름 → inode 매핑 제거 + inode 참조 수 감소

데이터 블록 관리 (블록 포인터)

일반적 구성 (예: ext2/3/4 inode)

포인터 종류	개수	설명
직접 블록 포인터	12	1블록 당 직접 포인터
간접 블록 포인터	1	1블록 → n개의 데이터 블록 포인터

포인터 종류	개수	설명
이중 간접 블록 포인터	1	1블록 → n개의 블록 → n개의 데이터 블록 포인터
삼중 간접 블록 포인터	1	1블록 → n개의 블록 → n개의 블록 → n개의 데이터 블록 포인터

→ **파일 크기가 클수록 점점 더 깊은 포인터 구조 사용**.

실습: 리눅스에서 inode 확인

ls -i 옵션

```
$ ls -i filename
123456 filename
```

→ `123456` → inode 번호.

stat 명령어

```
$ stat filename
  File: filename
  Size: 12345       Blocks: 24         IO Block: 4096   regular file
Device: 802h/2050d  Inode: 123456      Links: 1
Access: 2025-06-08 ...
Modify: ...
Change: ...
```

→ Inode 번호 및 메타데이터 확인 가능.

find로 inode 기준 검색

```
$ find . -inum 123456
```

→ 해당 inode 번호를 참조하는 **모든 경로(링크 포함)** 출력.

Hard link vs Symbolic link

구분	Hard link	Symbolic link
구조	동일 inode 공유	별도 파일(링크 파일), 원본 경로 저장
inode 번호	동일	다름
cross-fs 지원	불가 (동일 FS 내에서만 가능)	가능
원본 삭제 영향	inode 참조수 0이 되기 전까지 유지	원본 삭제 시 Broken link 됨

→ Hard link는 **inode 레벨에서 동일 파일 공유** → 완전한 복제와 같음.

실전 응용

- 대용량 파일 → **블록 포인터 설계 튜닝** 중요 → 성능 영향 발생
- 링크 수 확인 → Hard link 분석
- inode 부족 → "No space left on device" 발생할 수 있음 → 실제는 블록 남아 있어도 inode 부족이면 파일 생성 불가
- 고성능 시스템에서는 **inode reservation 튜닝** 사용 (mkfs 시 옵션 지정 가능)

정리

요소	설명
inode	파일 자체 표현 구조 (메타데이터 + 데이터 위치)
디렉토리 엔트리	이름 → inode 번호 매핑
데이터 블록	실제 파일 데이터 저장
블록 포인터	파일 크기에 따라 다단계 사용
Hard link	동일 inode 공유
Symbolic link	별도 파일, 경로 문자열 저장

결론

- 리눅스/유닉스 계열 파일 시스템은 **inode 중심 설계**다.
- 이름과 경로는 **inode 외부에 존재** → Hard link가 가능함.
- inode는 **디스크에서 매우 중요한 리소스** → 부족하면 "디스크 full" 오류 발생 가능.
- 실전에서는:
  - **inode 상태 모니터링** (`df -i`)
  - **Hard link 활용**
  - **블록 포인터 튜닝** 까지 적극 활용 가능.

8.2 `stat()`, `fstat()`, `lstat()`

개요

리눅스에서 **파일의 메타데이터(inode 정보)** 를 가져오는 대표적인 시스템 호출이 바로:

- `stat()`
- `fstat()`
- `lstat()`

이다.

이 함수들은 모두 다음 정보를 반환:

- 파일 유형 (regular file, directory, symlink 등)
- 파일 크기
- 소유자 (UID, GID)
- 권한 (mode)
- inode 번호
- 접근/수정/변경 시간 (atime, mtime, ctime)
- 링크 수
- 데이터 블록 수 등

---

1 공통 구조체: `struct stat`

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

struct stat {
    dev_t     st_dev;     // 파일이 존재하는 디바이스 ID
    ino_t     st_ino;     // Inode 번호
    mode_t    st_mode;    // 파일 유형 + 권한
    nlink_t   st_nlink;   // 하드 링크 수
    uid_t     st_uid;     // 소유자 UID
    gid_t     st_gid;     // 소유자 GID
    dev_t     st_rdev;    // 특수 파일의 장치 ID
    off_t     st_size;    // 파일 크기 (byte 단위)
    blksize_t st_blksize; // 파일 시스템의 I/O 블록 크기
    blkcnt_t  st_blocks;  // 파일이 사용하는 블록 수
    time_t    st_atime;   // 최근 접근 시간
    time_t    st_mtime;   // 최근 수정 시간
    time_t    st_ctime;   // 최근 상태 변경 시간
};
```

---

2 함수 원형 및 차이점

`stat()`

```c
int stat(const char *pathname, struct stat *statbuf);
```

- **pathname** 에 해당하는 파일의 정보를 가져옴.
- 심볼릭 링크인 경우 → **링크가 가리키는 원본 파일의 정보**를 반환.

`lstat()`

```c
int lstat(const char *pathname, struct stat *statbuf);
```

- pathname에 해당하는 파일의 정보를 가져옴.

- 심볼릭 링크인 경우 → **링크 자체에 대한 정보**를 반환.
  (링크가 가리키는 대상이 아닌, 링크 파일 자체의 inode 정보 반환)

→ **symbolic link 분석 시 반드시 `lstat()` 사용**.

`fstat()`

```
int fstat(int fd, struct stat *statbuf);
```

- **열려 있는 파일 디스크립터(fd)** 에 대해 정보를 가져옴.
- 예: `open()` 한 뒤, `fstat()` 호출 가능.

→ 파일 경로 대신 **열린 파일 핸들에 대해 메타데이터 확인**할 때 사용.

---

3 정리 비교표

함수	입력	Symbolic link 처리 방식	사용 용도
`stat()`	경로명	원본 대상 정보 반환	일반 파일 정보 확인
`lstat()`	경로명	링크 자체 정보 반환	Symbolic link 분석
`fstat()`	파일 디스크립터(fd)	N/A	열린 파일 핸들에 대한 정보 조회

---

4 실전 예제

stat 예제

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    struct stat sb;

    if (stat(argv[1], &sb) == -1) {
        perror("stat");
        return 1;
    }

    printf("File size: %lld bytes\n", (long long)sb.st_size);
    printf("Inode: %lu\n", (unsigned long)sb.st_ino);
    printf("Hard links: %lu\n", (unsigned long)sb.st_nlink);
    return 0;
```

```
}
```

lstat 예제

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    struct stat sb;

    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        return 1;
    }

    if (S_ISLNK(sb.st_mode)) {
        printf("%s is a symbolic link.\n", argv[1]);
    } else {
        printf("%s is not a symbolic link.\n", argv[1]);
    }

    printf("Inode: %lu\n", (unsigned long)sb.st_ino);
    return 0;
}
```

fstat 예제

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    struct stat sb;
```

```
    if (fstat(fd, &sb) == -1) {
        perror("fstat");
        close(fd);
        return 1;
    }

    printf("File size: %lld bytes\n", (long long)sb.st_size);
    printf("Inode: %lu\n", (unsigned long)sb.st_ino);

    close(fd);
    return 0;
}
```

---

5 실전 활용 예시

- **파일 크기 검사** → stat
- **파일 유형 확인 (Regular file? Directory? Symlink?)** → stat / lstat
- **심볼릭 링크 분석** → lstat 사용 필수
- **파일 핸들 기반 정보 조회 (예: stdin/out/err 등)** → fstat
    - `fstat(0, &sb);` → stdin의 상태 확인 가능.

---

6 정리

함수	대표적 활용
stat	파일 정보 확인 (파일 크기, 권한 등)
lstat	Symbolic link 분석 시 필수 사용
fstat	열린 파일 디스크립터에서 정보 확인 (파일 핸들 기반 API 작성 시 유용)

결론

- `stat()`, `lstat()`, `fstat()`는 **파일 메타데이터 접근의 기본 API**다.
- 시스템 프로그래밍, 파일 분석 도구, 백업 프로그램 등에서 **반드시 사용하게 되는 함수**다.
- 심볼릭 링크가 등장하는 복잡한 파일 구조에서는 **lstat 사용 여부가 매우 중요**하다 → 실수 잦은 포인트.

8.3 디렉토리 열기/읽기 (`opendir`, `readdir`)

개요

리눅스에서는 디렉토리도 **파일의 특수한 형태**다.

- **디렉토리 파일**은 **"파일 이름 → inode 번호" 쌍의 리스트**를 저장한다.
- C 표준 라이브러리 + POSIX 표준은 디렉토리 탐색을 위해 다음 API 제공:

함수	기능
`opendir()`	디렉토리 열기
`readdir()`	디렉토리 엔트리 읽기
`closedir()`	디렉토리 닫기

추가로:

함수	기능
`rewinddir()`	디렉토리 읽기 포인터를 처음으로 되돌리기
`seekdir()` / `telldir()`	디렉토리 내 위치 이동 / 위치 저장

---

1 데이터 구조

`DIR *`

```
typedef struct __dirstream DIR;
```

- `opendir()` 호출 시 반환 → 이후 `readdir()`, `closedir()` 등에 사용.

`struct dirent`

```
#include <dirent.h>

struct dirent {
    ino_t          d_ino;       // Inode 번호
    off_t          d_off;       // Offset (사용 여부 시스템마다 다름)
    unsigned short d_reclen;    // Directory record length
    unsigned char  d_type;      // 파일 유형 (optional, 지원 안하는 FS도 있음)
    char           d_name[];    // Null-terminated 파일 이름
};
```

---

2 주요 함수 설명

`opendir()`

```
DIR *opendir(const char *name);
```

- 디렉토리 스트림 오픈 → 성공 시 `DIR *` 반환, 실패 시 NULL.

`readdir()`

```
struct dirent *readdir(DIR *dirp);
```

- 디렉토리에서 **다음 엔트리** 읽기.
- 더 이상 읽을 엔트리 없으면 NULL 반환.
- 주의 → 반환된 `struct dirent *`는 **static 영역에 저장됨** → **매 호출마다 덮어쓰기됨** → 복사 필요 시 주의.

`closedir()`

```
int closedir(DIR *dirp);
```

- 디렉토리 스트림 닫기.

---

3 실전 예제

디렉토리 엔트리 목록 출력

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    DIR *dirp = opendir(argv[1]);
    if (dirp == NULL) {
        perror("opendir");
        return 1;
    }

    struct dirent *entry;
    while ((entry = readdir(dirp)) != NULL) {
        printf("Name: %s", entry->d_name);

        // d_type 사용 가능 시 파일 유형 출력
        if (entry->d_type == DT_REG) printf(" [Regular file]");
        else if (entry->d_type == DT_DIR) printf(" [Directory]");
        else if (entry->d_type == DT_LNK) printf(" [Symbolic link]");

        printf("\n");
    }

    closedir(dirp);
    return 0;
```

```
}
```

실행 예시

```
$ ./dir_list .
Name: .
Name: ..
Name: file1.txt [Regular file]
Name: subdir [Directory]
Name: link_to_file [Symbolic link]
```

---

4 주의사항

`.` 와 `..` 항목 존재

- `.` → 현재 디렉토리
- `..` → 상위 디렉토리
- 항상 포함됨 → 원하면 skip 가능:

```
if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
    continue;
}
```

`d_type` 의 주의점

- `d_type` 은 **디렉토리 항목 자체에 유형 저장 가능**한 FS에서만 유효.
  - 예: ext4 → 지원
  - 일부 FS (NFS 등) → `d_type == DT_UNKNOWN` 반환 → 이 경우 `stat()` **또는** `lstat()` **호출해서 확인** 필요.

```
if (entry->d_type == DT_UNKNOWN) {
    // lstat() 사용 권장
}
```

---

5 고급 사용법

rewinddir()

```
void rewinddir(DIR *dirp);
```

- 디렉토리 읽기 포인터를 처음으로 되돌림.

---

telldir() / seekdir()

```
long telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
```

- 디렉토리 내 위치 저장 / 복원 → 대형 디렉토리 탐색 시 유용.

---

6 실전 활용 예시

- 디렉토리 탐색 프로그램 구현 (`ls` 유사 기능)
- 파일 백업/복원 시 디렉토리 재귀 탐색 → `readdir()` + `stat()` 조합 사용
- 정적 분석 도구 → 디렉토리 구조 분석 필요
- 리눅스 시스템 프로그램에서 **파일 tree crawler** 작성 시 기본 API

---

7 정리

함수	기능	사용 시 주의사항
`opendir()`	디렉토리 열기	성공 시 DIR * 반환
`readdir()`	다음 디렉토리 엔트리 읽기	static 구조 → 복사 주의
`closedir()`	디렉토리 닫기	사용 후 반드시 닫기
`rewinddir()`	읽기 위치 처음으로 이동	반복 탐색 시 유용
`seekdir()` / `telldir()`	디렉토리 내 위치 저장/복원	대형 디렉토리에서 유용
`d_type`	파일 유형	일부 FS에서 DT_UNKNOWN 발생 가능 → `stat()` 필요

결론

- `opendir()` + `readdir()` + `closedir()` 는 리눅스에서 디렉토리 구조를 탐색하는 **표준 API**다.
- `d_type`은 가용 시 빠르게 유형 판별 가능하지만, 일부 FS에서는 보장되지 않으므로 `lstat()` **와 병행 사용** 설계 필요.
- 실전에서는:
  - **ls / find 구현**
  - **백업 도구 구현**
  - **파일 tree crawler 구현**
    등에 반드시 활용된다.

8.4 파일 권한 변경 (`chmod`, `chown`)

개요

리눅스/유닉스 계열 시스템에서 **파일 권한**은 매우 중요한 보안 메커니즘이다.
→ 각 파일/디렉토리에 대해:

- **소유자(User)**
- **그룹(Group)**
- **기타(Others)**

별로 **읽기/쓰기/실행 권한**을 설정할 수 있다.

파일 권한은 **inode 구조체 내 st_mode 필드**에 저장된다.
이를 변경하는 대표적인 시스템 호출이:

함수	기능
`chmod()`	파일 권한 변경
`fchmod()`	열린 파일 디스크립터의 권한 변경
`fchmodat()`	상대경로 + 디렉토리 핸들 기반 권한 변경
`chown()`	파일 소유자/그룹 변경
`fchown()`	열린 파일 디스크립터의 소유자/그룹 변경
`fchownat()`	상대경로 + 디렉토리 핸들 기반 소유자/그룹 변경

1️⃣ 파일 권한 비트 구조

```
st_mode (16비트 등) → mode_t 타입
```

비트	의미
S_IFMT	파일 유형 비트 마스크
S_IFREG	Regular file
S_IFDIR	Directory
…	…
S_IRUSR	사용자 읽기
S_IWUSR	사용자 쓰기
S_IXUSR	사용자 실행
S_IRGRP	그룹 읽기

비트	의미
S_IWGRP	그룹 쓰기
S_IXGRP	그룹 실행
S_IROTH	기타 읽기
S_IWOTH	기타 쓰기
S_IXOTH	기타 실행

예

```
-rwxr-xr--   → 0754
```

부분	의미	8진수
rwx	사용자(User): rwx → 7	
r-x	그룹(Group): r-x → 5	
r--	기타(Others): r-- → 4	

→ `chmod 0754 filename`

2️⃣ chmod()

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

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

설명

- `pathname` 경로에 해당하는 파일/디렉토리의 **권한을 mode로 변경**.
- mode는 **위에서 설명한 S_IRUSR 등 상수의 OR 조합**.

예제

```
chmod("file.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);  // -rw-r--r--
```

→ 사용자 읽기/쓰기, 그룹 읽기, 기타 읽기.

명령어 사용

```
chmod 0755 filename
chmod u+x script.sh
chmod g-w file.txt
chmod o-rwx file.txt
```

3 chown()

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
```

설명

- 해당 파일의 **소유자(UID), 그룹(GID)** 를 변경.
- **uid/gid == -1** → 해당 항목은 변경하지 않음.

예제

```
chown("file.txt", 1000, 1000);  // 소유자 UID=1000, 그룹 GID=1000으로 변경
```

명령어 사용

```
sudo chown user:group filename
sudo chown user filename   # 그룹 변경 안함
```

4 fchmod(), fchown()

함수	기능
`fchmod(fd, mode)`	열린 파일 디스크립터의 권한 변경
`fchown(fd, uid, gid)`	열린 파일 디스크립터의 소유자/그룹 변경

→ 사용 예: **open()한 파일에 대해 동적 설정 시 유용**.

예제

```
int fd = open("file.txt", O_WRONLY);
fchmod(fd, S_IRUSR | S_IWUSR);
fchown(fd, 1000, -1);  // 그룹 유지, UID만 변경
close(fd);
```

5 fchmodat(), fchownat()

```
int fchmodat(int dirfd, const char *pathname, mode_t mode, int flags);
int fchownat(int dirfd, const char *pathname, uid_t owner, gid_t group, int flags);
```

설명

- **디렉토리 핸들(dirfd)** + 상대경로 사용 가능.
- `AT_FDCWD` 사용 시 현재 작업 디렉토리 기준.
- `flags` → `AT_SYMLINK_NOFOLLOW` 등 옵션 가능.

→ 고급 API → **재귀 디렉토리 작업, chroot 환경 등에서 유용**.

---

6 실전 예제: chmod + chown 조합

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

int main() {
    const char *filename = "file.txt";

    // 권한 변경 → rw-r--r--
    if (chmod(filename, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1) {
        perror("chmod");
    }

    // 소유자 변경 → UID=1000, GID=1000
    if (chown(filename, 1000, 1000) == -1) {
        perror("chown");
    }

    return 0;
}
```

---

7 주의사항

함수	주의사항
`chmod()`	일반 사용자 → 그룹/기타 쓰기 권한 설정/해제 제한 가능
`chown()`	일반 사용자 → 자신이 소유한 파일만 가능 / root 권한 필요
setuid/setgid/sticky bit 설정 시	별도 주의 (S_ISUID, S_ISGID, S_ISVTX 사용 가능)
Symlink에 적용 시	기본 `chmod()` → 링크가 아니라 링크 대상에 적용됨 → `lchmod()` 필요 (deprecated, 일부 시스템에만 존재)

---

8 권한 변경과 보안

- **정확한 권한 설정은 시스템 보안에 매우 중요**하다.
- 일반적으로:

파일 유형	권장 기본 권한
일반 파일	0644 (-rw-r--r--)
실행 파일	0755 (-rwxr-xr-x)
디렉토리	0755 (drwxr-xr-x)
private 설정 파일	0600 (-rw-------)

- 프로그래밍 시 `umask()` 호출 여부도 함께 고려해야 함 → 기본 생성 권한 제한.

---

9 정리

함수	기능	비고
`chmod()`	파일/디렉토리 권한 변경	기본 사용
`fchmod()`	열린 파일 핸들의 권한 변경	fd 기반 제어
`chown()`	파일/디렉토리 소유자/그룹 변경	root 권한 필요
`fchown()`	열린 파일 핸들의 소유자/그룹 변경	fd 기반 제어
`fchmodat()` / `fchownat()`	디렉토리 핸들 + 상대경로 제어	고급 API (secure coding용도 많음)

결론

- `chmod()` / `chown()`는 리눅스 시스템 프로그래밍에서 **파일 보호/접근제어의 핵심**이다.
- **권한 설정 실수 → 보안 취약점 발생 가능** → 프로그래밍 시 반드시 정확히 설정.
- 고급 상황 (chroot, secure chdir 등)에서는 **fchmodat()/fchownat()** 사용 적극 추천.

8.5 하드 링크 vs 심볼릭 링크

개요

리눅스/유닉스 계열 시스템에서 **링크(link)** 는 **파일의 또 다른 이름/접근 경로**를 제공하는 기능이다.

대표적인 링크 방식:

종류	영어 명칭
하드 링크	Hard Link

종류	영어 명칭
심볼릭 링크	Symbolic Link / Symlink / Soft Link

링크 기능을 이용하면:

- **하나의 파일을 여러 경로에서 접근 가능**
- **백업, 참조, 가상 경로 구성 등에 활용 가능**

기본 원리

리눅스 파일 시스템의 **파일 이름은 inode 번호에 대한 매핑**이다:

```
파일 이름 → inode 번호 → 파일 메타데이터 + 데이터 블록
```

- 하드 링크는 **동일 inode 번호를 가진 새로운 디렉토리 엔트리**를 추가.
- 심볼릭 링크는 **별도의 inode에 저장된 경로 문자열**을 가진 별도 파일 생성.

1️⃣ 하드 링크 (Hard Link)

원리

- **inode 번호를 공유**.
- 링크 수 (`st_nlink`) 가 증가 → inode 참조 수 증가.
- 동일 파일로 취급됨 → 어느 링크에서 수정해도 내용은 동일.

특징

항목	설명
inode 번호	원본과 동일
파일 내용	동일
원본 삭제 시 영향	Hard link가 남아있으면 파일 유지됨 (데이터 유지)
cross-filesystem 사용	불가 (같은 파일 시스템 내에서만 가능)
디렉토리 링크	제한됨 (`ln` 기본적으로 디렉토리에 Hard link 금지)

생성 명령어

```
ln file1 file2_hard
```

확인

```
ls -li
```

→ 동일한 inode 번호 출력됨.

st_nlink 활용

```
stat file1
```

→ st_nlink → **링크 수 증가** 확인 가능.

2️⃣ 심볼릭 링크 (Symbolic Link, Symlink)

원리

- 별도의 inode를 가짐 → 파일 유형은 `l` (symbolic link).
- 데이터 영역에 **"원본 파일 경로 문자열"** 저장.
- 원본 경로를 참조 → 원본 변경/삭제 시 링크 깨짐 가능 (broken link).

특징

항목	설명
inode 번호	서로 다름
파일 내용	경로 문자열
원본 삭제 시 영향	broken link 발생 (dangling symlink)
cross-filesystem 사용	가능
디렉토리 symlink	가능 (일반적으로 많이 사용)

생성 명령어

```
ln -s file1 file2_symlink
```

확인

```
ls -li
ls -l
```

→ `l` 표시됨 → `file2_symlink -> file1` 형태 출력.

lstat 사용 주의

```
lstat("file2_symlink", &sb);  // symlink 자체 정보 조회
stat("file2_symlink", &sb);   // 원본 파일 정보 조회
```

---

3 차이점 비교표

구분	하드 링크	심볼릭 링크
inode 번호	동일	다름
원본 삭제 시	데이터 유지	Broken link 발생
파일 시스템 간 링크	불가	가능
디렉토리 링크	일반적으로 제한됨	가능
사용 목적	동일 파일의 복수 경로 확보	유연한 경로 참조 (cross-fs 포함)
성능	직접 접근 (inode 직접 접근)	추가 경로 해석 필요 (간접)
권장 사용	백업, 동일 FS 내 중복 경로	유저 친화적 shortcut, cross-fs link

---

4 실전 예제

1 하드 링크 실험

```
echo "Hello World" > file1
ln file1 file1_hard
ls -li file1 file1_hard
```

확인:

- inode 번호 동일
- `cat file1_hard` → Hello World 출력
- `rm file1` → `file1_hard` 여전히 사용 가능

---

2 심볼릭 링크 실험

```
ln -s file1 file1_symlink
ls -li file1 file1_symlink
```

확인:

- inode 번호 다름
- `ls -l` → `file1_symlink -> file1` 출력
- `rm file1` → `file1_symlink` broken link 됨 → `ls -l file1_symlink` → 빨간색 표시됨.

5 실전 활용 팁

상황	권장 링크 유형
동일한 파일 여러 경로에서 **동등하게 사용 (백업 등)**	Hard Link
cross-fs 사용 필요 / 디렉토리 링크 필요	Symbolic Link
패키지 관리, 유틸리티 경로 통일	Symbolic Link
롤백/스냅샷 구성을 위해 복수 inode 확보 필요	Hard Link + 조합 사용 가능

6 고급 활용

하드 링크 기반 백업 시스템

- 기존 백업의 파일과 동일한 파일은 **Hard link로만 구성** → 디스크 공간 절약.

Symbolic link 활용

- /etc/alternatives 시스템:
  - 다양한 버전의 유틸리티 (`java`, `gcc` 등)를 symlink로 연결해 version switching 구성.

라이브러리 경로 구성

```
libfoo.so → libfoo.so.1 → libfoo.so.1.2.3
```

→ 단계적 symbolic link 구성.

7 정리

특성	Hard Link	Symbolic Link
inode 공유 여부	O	X
경로 해석	필요 없음 (inode 직접)	경로 문자열 해석 필요
원본 삭제 시 영향	없음 (데이터 유지)	Broken link 발생 가능
cross-filesystem 지원	X	O
디렉토리 지원	제한됨	O
성능	빠름	약간 느림 (간접 경로 해석 필요)

결론

- **Hard Link** → inode 레벨에서 동일한 파일을 **물리적으로 공유**.
  (백업, 중복 제거, 파일 시스템 레벨 최적화 등에서 활용)
- **Symbolic Link** → 별도의 inode에 **경로 문자열 저장**.
  (cross-fs, 디렉토리 참조, 유틸리티 구성 등에서 활용)
- 실전에서는 두 링크 유형을 **용도에 맞게 적절히 사용**하는 것이 중요하다.

📎 실습

디렉토리 트리 탐색기 만들기

목표

- 지정한 디렉토리부터 시작해서 **재귀적으로 하위 디렉토리까지 전부 탐색**.
- **트리 구조 출력** → `tree` 명령어 비슷한 출력 형태.
- 각 파일/디렉토리에 대해:
  - 이름 출력
  - 파일 유형 (디렉토리/일반 파일/링크 등) 구분

1️⃣ 설계 원리

핵심 흐름

1️⃣ 디렉토리 열기 → `opendir()`
2️⃣ 디렉토리 엔트리 반복 → `readdir()`
3️⃣ 각 엔트리에 대해:

- `.` 와 `..` 건너뛰기
- 디렉토리이면 → 재귀 호출
- 파일이면 → 출력

파일 유형 판별 방법

- `d_type` → DT_REG, DT_DIR, DT_LNK 사용 가능 (일부 FS에서 DT_UNKNOWN 나올 수 있음).
- 정확하게 하려면 `lstat()` 사용 → 파일 유형 확인.

2️⃣ 전체 예제 코드 (tree_traverse.c)

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>
```

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

void print_indent(int depth) {
    for (int i = 0; i < depth; i++) {
        printf("  ");
    }
}

void traverse(const char *path, int depth) {
    DIR *dirp = opendir(path);
    if (dirp == NULL) {
        perror(path);
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dirp)) != NULL) {
        // Skip "." and ".."
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;

        // Full path 구성
        char fullpath[4096];
        snprintf(fullpath, sizeof(fullpath), "%s/%s", path, entry->d_name);

        struct stat sb;
        if (lstat(fullpath, &sb) == -1) {
            perror(fullpath);
            continue;
        }

        // 출력
        print_indent(depth);
        if (S_ISDIR(sb.st_mode)) {
            printf("[DIR]  %s\n", entry->d_name);
            // 재귀 호출
            traverse(fullpath, depth + 1);
        } else if (S_ISLNK(sb.st_mode)) {
            printf("[LINK] %s\n", entry->d_name);
        } else if (S_ISREG(sb.st_mode)) {
            printf("[FILE] %s\n", entry->d_name);
        } else {
            printf("[OTHER] %s\n", entry->d_name);
        }
    }

    closedir(dirp);
}

int main(int argc, char *argv[]) {
    const char *start_path = ".";
    if (argc >= 2) {
```

```
        start_path = argv[1];
    }

    printf("Directory tree for: %s\n", start_path);
    traverse(start_path, 0);

    return 0;
}
```

---

3 컴파일 및 실행

```
$ gcc -o tree_traverse tree_traverse.c
$ ./tree_traverse /etc
```

→ 출력 예시:

```
Directory tree for: /etc
[DIR]  alternatives
[DIR]  apt
[FILE] bash.bashrc
[DIR]  cron.d
  [FILE] myjob
[LINK] localtime
...
```

---

4 주요 기능 설명

기능	구현 포인트
디렉토리 열기	`opendir()` 사용
엔트리 반복	`readdir()` 사용
"." ".." skip	`strcmp` 사용으로 건너뜀
파일 유형 판별	`lstat()` 사용 → 정확한 판별 가능
출력 포맷	depth 기준 indent 적용
재귀 호출	디렉토리일 경우 → `traverse()` 재귀 호출

---

5 고급 확장 아이디어

기능 추가 예제

- **파일 크기 출력** → `sb.st_size` 출력
- **inode 번호 출력** → `sb.st_ino` 출력
- **링크 대상 경로 출력** → `readlink()` 사용
- **파일 정렬** → `scandir()` + `alphasort()` 사용 가능
- **depth 제한 기능** → depth max parameter 추가

---

6 주의사항

상황	주의사항
심볼릭 링크 순환	symlink 재귀 탐색 시 순환 발생 가능 → depth 제한 또는 symlink skip 추천
Permission denied	`opendir()` 실패 처리 필요 → `perror()` 사용
경로 길이	`snprintf()` 사용하여 버퍼 오버플로 방지

---

7 정리

API	기능
`opendir()`	디렉토리 열기
`readdir()`	디렉토리 엔트리 읽기
`closedir()`	디렉토리 닫기
`lstat()`	정확한 파일 유형 판별
`readlink()`	symlink 대상 확인 (고급)
`scandir()`	정렬된 디렉토리 탐색 (고급)

결론

- **디렉토리 트리 탐색기**는 **파일 시스템 탐색의 기본 패턴**이다.
- `opendir()` / `readdir()` / `lstat()` 를 조합하면 **구조적인 tree 탐색기 구현 가능**.
- 실전에서는:
  - **backup/restore 유틸리티**
  - **보안 검사 스크립트**
  - **파일 indexer**
    등에 기본으로 들어가는 기술이다.

`stat` 정보로 파일 정렬 프로그램

목표

- 지정한 디렉토리 내 파일들의:
  - **이름**
  - **파일 크기**
  - **inode 번호**
  - **파일 유형**
- 을 수집하고
- **파일 크기 기준으로 정렬 후 출력** (기본 예시)

→ 확장하면 **mtime 기준 정렬, 파일 이름 정렬 등 다양한 정렬 기준 적용 가능**.

---

1 설계 원리

- 디렉토리 열기 → `opendir()`
- 디렉토리 엔트리 반복 → `readdir()`
- 각 엔트리에 대해:
  - `lstat()` 호출 → **struct stat 정보 수집**
- 수집한 정보를 구조체 배열에 저장
- 정렬 → `qsort()` 사용
- 출력

---

2 전체 예제 코드 (stat_sort.c)

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#define MAX_FILES 10000

typedef struct {
    char path[4096];
    off_t size;
    ino_t inode;
    mode_t mode;
} file_info_t;

```

```c
int compare_size(const void *a, const void *b) {
    const file_info_t *fa = (const file_info_t *)a;
    const file_info_t *fb = (const file_info_t *)b;

    // 내림차순 정렬
    if (fb->size > fa->size) return 1;
    else if (fb->size < fa->size) return -1;
    else return 0;
}

void print_file_type(mode_t mode) {
    if (S_ISDIR(mode)) printf("[DIR]  ");
    else if (S_ISREG(mode)) printf("[FILE] ");
    else if (S_ISLNK(mode)) printf("[LINK] ");
    else printf("[OTHER] ");
}

int main(int argc, char *argv[]) {
    const char *dirpath = ".";
    if (argc >= 2) {
        dirpath = argv[1];
    }

    DIR *dirp = opendir(dirpath);
    if (dirp == NULL) {
        perror(dirpath);
        return 1;
    }

    struct dirent *entry;
    file_info_t files[MAX_FILES];
    size_t file_count = 0;

    while ((entry = readdir(dirp)) != NULL) {
        // Skip "." and ".."
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;

        // Build full path
        char fullpath[4096];
        snprintf(fullpath, sizeof(fullpath), "%s/%s", dirpath, entry->d_name);

        struct stat sb;
        if (lstat(fullpath, &sb) == -1) {
            perror(fullpath);
            continue;
        }

        // Save info
        strncpy(files[file_count].path, entry->d_name, sizeof(files[file_count].path) -
1);
        files[file_count].size = sb.st_size;
        files[file_count].inode = sb.st_ino;
```

```
        files[file_count].mode = sb.st_mode;
        file_count++;

        if (file_count >= MAX_FILES) {
            fprintf(stderr, "Too many files (limit %d)\n", MAX_FILES);
            break;
        }
    }

    closedir(dirp);

    // Sort by size
    qsort(files, file_count, sizeof(file_info_t), compare_size);

    // Print result
    printf("Sorted files in '%s' by size:\n", dirpath);
    printf("SIZE\tINODE\tTYPE\tNAME\n");
    for (size_t i = 0; i < file_count; i++) {
        printf("%8lld\t%lu\t", (long long)files[i].size, (unsigned
long)files[i].inode);
        print_file_type(files[i].mode);
        printf("%s\n", files[i].path);
    }

    return 0;
}
```

3 컴파일 및 실행

```
$ gcc -o stat_sort stat_sort.c
$ ./stat_sort .
```

→ 출력 예시:

```
Sorted files in '.' by size:
SIZE    INODE   TYPE    NAME
 123456  123456 [FILE]  bigfile.dat
   4096  234567 [DIR]   subdir
   2048  345678 [FILE]  notes.txt
    512  456789 [LINK]  mylink
    128  567890 [FILE]  README.md
```

4 주요 기능 설명

기능	구현 포인트
디렉토리 열기	`opendir()` 사용
엔트리 반복	`readdir()` 사용

기능	구현 포인트
파일 메타정보 수집	`lstat()` 호출로 `struct stat` 수집
정보 저장	구조체 배열(file_info_t) 사용
정렬	`qsort()` 사용, compare 함수 구현
출력	유형 구분 후 출력

---

5 고급 확장 아이디어

다양한 정렬 기준 추가

- **mtime 기준 정렬** → `sb.st_mtime` 사용
- **inode 번호 기준 정렬** → `sb.st_ino` 사용
- **파일 이름 알파벳 정렬** → strcmp 기반 compare 함수 작성

트리 탐색으로 확장

- 하위 디렉토리까지 재귀 탐색 후 정렬하기 (디렉토리 트리 탐색기와 결합 가능).

출력 형식 개선

- **컬러 출력** 적용 (`isatty()` 활용)
- **출력 정렬 패딩/정렬** 개선
- **json/csv 형식 출력** → log 시스템과 연계 가능.

---

6 주의사항

상황	주의사항
디렉토리 엔트리 개수	배열 크기 주의 필요 → 동적 할당으로 개선 가능
d_type 신뢰성	정확한 유형은 `lstat()` 으로 확인 추천
symlink 순환 문제	기본적으로 lstat 사용 시 안전 (symlink 자체 정보만 읽음)

---

7 정리

API	기능
`opendir()` / `readdir()`	디렉토리 탐색
`lstat()`	정확한 파일 메타데이터 수집
`qsort()`	정렬 처리

API	기능
`struct stat`	size, inode, mode 등 다양한 정보 제공

결론

- **stat 정보 기반 파일 정렬 프로그램**은 **파일 시스템 분석/관리 도구의 핵심 패턴**이다.
- 응용:
  - **대용량 디스크 사용량 분석기** (`du` 대체용)
  - **중복 파일 탐지기**
  - **파일 정렬 backup 툴**
  - **디렉토리 용량 시각화 툴**의 기본 구조로 사용 가능.

하드링크/심볼릭링크 실험

준비

실험용 디렉토리 생성

```
$ mkdir link_test
$ cd link_test
```

실험용 원본 파일 생성

```
$ echo "This is the original file." > original.txt
```

1️⃣ 하드링크 실험

1.1 하드링크 생성

```
$ ln original.txt hardlink.txt
```

1.2 확인

```
$ ls -li
```

→ 결과 예시:

```
123456 -rw-r--r-- 2 user user 28 Jun  8 21:00 hardlink.txt
123456 -rw-r--r-- 2 user user 28 Jun  8 21:00 original.txt
```

관찰 포인트

항목	관찰 결과
inode 번호	동일 (123456)
st_nlink	2 (링크 수 증가됨)

1.3 내용 확인

```
$ cat hardlink.txt
This is the original file.

$ cat original.txt
This is the original file.
```

→ **내용 동일**.

1.4 원본 삭제 후 확인

```
$ rm original.txt
$ ls -li
```

→ `hardlink.txt` 남아있고 내용 정상 확인 가능.

```
$ cat hardlink.txt
This is the original file.
```

관찰 포인트
→ 하드링크는 **inode 직접 참조** → 원본 삭제해도 **데이터 유지됨**.

2️⃣ 심볼릭링크 실험

2.1 원본 다시 생성

```
$ echo "New content." > original.txt
```

2.2 심볼릭링크 생성

```
$ ln -s original.txt symlink.txt
```

2.3 확인

```
$ ls -li
```

예시:

```
654321 lrwxrwxrwx 1 user user 13 Jun  8 21:10 symlink.txt -> original.txt
123457 -rw-r--r-- 1 user user 14 Jun  8 21:10 original.txt
```

관찰 포인트

항목	관찰 결과
inode 번호	서로 다름
파일 유형	`l` (symbolic link)
symlink 내용	"original.txt" 문자열 저장됨

2.4 내용 확인

```
$ cat symlink.txt
New content.
```

→ 정상 출력.

2.5 원본 삭제 후 확인

```
$ rm original.txt
$ ls -li
```

```
$ cat symlink.txt
cat: symlink.txt: No such file or directory
```

관찰 포인트

→ 심볼릭링크는 **경로 문자열만 저장** → 원본 삭제 시 **broken link** 발생.

→ `ls -l`에서 symlink가 빨간색으로 표시됨 (broken).

---

3 정리 비교

항목	하드링크	심볼릭링크
inode 공유	O (동일 inode)	X (별도 inode)
원본 삭제 영향	영향 없음 (데이터 유지)	Broken link 발생
cross-fs 지원	불가 (같은 FS 내에서만 가능)	가능
디렉토리 링크	제한됨	가능
파일 유형	regular file	symbolic link (`l`)

항목	하드링크	심볼릭링크
링크 대상	데이터 블록 직접 공유	경로 문자열 저장

---

4 실전 팁

하드링크 활용

- 동일 파일을 여러 이름으로 관리 (백업 시스템, deduplication 등).
- 특정 상태 snapshot 구현.

심볼릭링크 활용

- `/etc/alternatives` → 유틸리티 버전 관리.
- `/usr/bin/python` → 여러 버전 간 switching.
- 디렉토리 링크 → config 디렉토리 가상화 가능.

---

5 실습 응용 아이디어

- 하드링크 / 심볼릭링크 **N개 생성 후** `stat` 정보 확인 (`st_nlink` 변화 관찰).
- 심볼릭링크 대상으로 **상대경로 / 절대경로** 차이 실험.
- 심볼릭링크 → 심볼릭링크 → 심볼릭링크 **multi-hop symlink** 실험 (가능하나 너무 깊으면 문제 발생 가능).

결론

- **하드링크** → inode 직접 참조 → 원본과 완전히 동일한 "정식 파일" 생성.
- **심볼릭링크** → "경로 shortcut" → 유연성 높음, cross-fs 가능, 원본 삭제 시 broken 가능.
- 실전에서는 용도에 따라 **정확하게 구분해서 사용**해야 함.