chapter05. Files and Directories

↳ UNIX에서는 관리 어떻게?

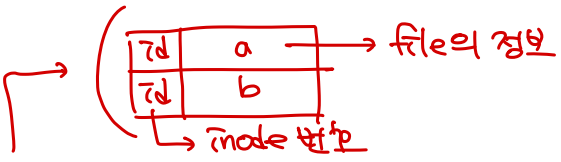
objectives

1. Learn about file systems and directories
2. Experiment with directory traversal → file의 컨텐츠와 관리
3. Explore UNIX inode implementation
4. Use functions for accessing directories(system call)
5. Understand hard links and symbolic(soft) links
   ↳ 대부분 OS에 존재하는 file

File Systems

OS : organize physical disks into file systems.

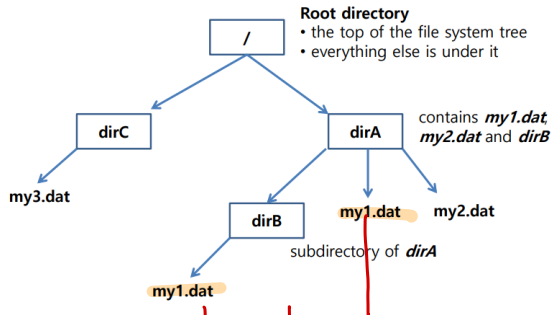
- collection of files and attributes (location, name)
- filename, offset으로 file의 경로를 알려줌 (disk의 위치 X, system 내에서의 위치)



Directory

file은 disk 위의 physical location과 관련된 directory entries 가지고 있음

대부분의 file system → dir들의 tree 구조



↳ 다른 dir에 있다면 같은 이름의 파일 존재 가능

- **Root directory** : file system tree의 top dir → 모든 directory는 root 자식
  - “/”
- **Parent directory** : 현재 작업 중인 dir의 부모 dir
  - “..”
- **Current directory** : 현재 작업 중인 dir
  - “.”
- **Home directory** : login → terminal 오픈 시에 default로 들어가는 dir (보통 user id가 이름)
- **Sub-directory** : 다른 dir에 속해있는 dir

Pathname → slash(/)로 구분자 사용

- **Absolute pathname**(fully qualified pathname)

  : 절대 경로
  - root부터 시작
  - ex) /dirA/dirB/my1.dat

다른 경로
1. /dirA/my1.dat 절대경로
2. ../my1.dat 상대경로
3. ./../my1.dat
   ↓ 현재 표시

- **Relative pathname : 상대 경로**
  - Current부터 시작 (/로 시작하지 않고 ../로 시작)
  - ex) 현재 dir이 dirA라면, ../dirB/my1.dat

Change Directory → chdir

현재 dir을 parameter의 경로의 dir로 옮겨줌

→ chdir() : 현재 dir에만 영향을 미치는 function

```
#include <unistd.h>
int chdir(const char *path);
//path : 옮기고자 하는 경로

//return 0 > successful
//return -1 > unsuccessful
```

```
//Example - mychdir.c
#include <unistd.h>
```

```c
int main(void){
  char *directory = "/tmp";
  if (chdir(directory) == -1)
    perror("Failed to change current working directory to /tmp");
}
```

Get Current Directory → getcwd

현재 dir의 경로를 return

→ terminal에서의 pwd 명령어와 같은 기능 수행

```c
#include <unistd.h>
char *getcwd(char *buf, size_t size);
//buf : 현재 dir의 경로명을 저장하고 있는 buffer
//      PATH_MAX -> POSIX에서 pathname의 maximum length에 대한 constant 값
//size : the maximum length pathname that buf can accommodate

//return (a pointer to buf) > successful
//return NULL > unsuccessful
```

```c
//Example
#include <limits.h>
#include <stdio.h>
#include <unistd.h>
#ifndef PATH_MAX
#define PATH_MAX 255
#endif

int main(void){
  char mycwd[PATH_MAX];

  if (getcwd(mycwd, PATH_MAX) == NULL) {
    perror("Failed to get current working directory");
    return 1;
  }
  printf("Current working directory: %s\n", mycwd);
  return 0;
}
```

Search Path

terminal 명령어처럼 실행파일명만 입력해도 실행이 되는 이유?

→ shell : PATH 환경 변수에 있는 모든 dir 경로를 찾아가 해당 file name만 찾아서 실행

- which * : 실행가능한 *의 절대경로 출력하는 program
- ". (PATH)" : risky !!
  - shell이 같은 이름의 system program 대신에 local program을 실행시킬 때 이상한 결과가 나오거나 보안 상의 위험으로 여겨질 수 있음.

Directory Access → System call function !

- **Directory open**

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

DIR *opendir(const char *dirname);

//dirname : 열고 싶은 directory 이름

//return DIR pointer -> success
//return NULL -> fail
```

  - DIR type
    - <dirent.h>에 define
    - Directory Stream을 represent
    - **Directory Stream**
      - an ordered sequence of all of the directory entries in a particular directory
      - the order of the entries in a directory stream : file name에 의해 정렬될 필요 x

- **Directory read**

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

struct dirent *readdir (DIR *dirptr);

//dirptr : read할 DIR pointer(opendir의 return 값)

//return first struct DIR pointer -> success
//return NULL -> fail 혹은 end of the directory
```

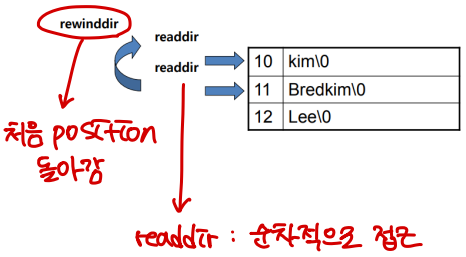
  - dirptr pointer가 가리키는 directory stream 안의 entry들 return 함으로써 directory read
  - struct dirent
    - next directory entry에 대한 정보를 담고 있는 structure

- **Directory close and rewind**

```
#include <dirent.h>

int closedir (DIR *dirptr);
//dirptr : close할 dir path name

//return 0 -> success
//return -1 -> fail
```

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

void *rewinddir (DIR *dirptr);
//dirptr의 dir stream의 시작으로 reposiiton
```

Directory read and rewind

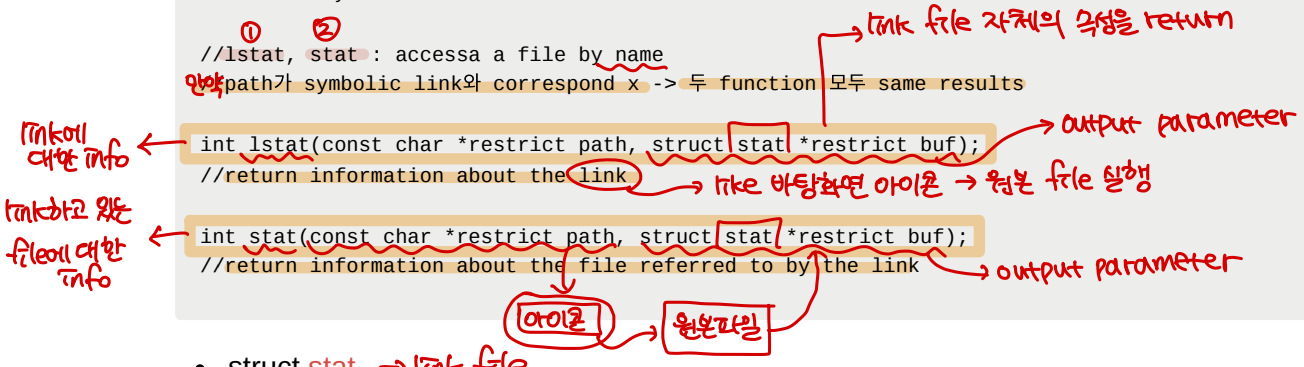



```
//Example - showname.c   file명 출력 예제

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

int main(int argc, char *argv[]) {
  struct dirent *direntp;   → directory entry 구조체
  DIR *dirp;

  if (argc != 2) {   argv[1] = target dir
    fprintf(stderr, "Usage: %s
    directory_name\n", argv[0]);
    return 1;
  }
① open
  if ((dirp = opendir(argv[1])) == NULL) {
    perror ("Failed to open directory");
    return 1;
  }                          open한 dir path
② read
  while ((direntp = readdir(dirp)) != NULL)
    printf("%s\n", direntp->d_name);   → file name
    while ((closedir(dirp) == -1) && (errno == EINTR)) ;
  return 0;
③ close
}
```

File status Information

→ meta data [file access 권한
마지막 open 시간
⋮

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

//lstat, stat : accessa a file by name
//path가 symbolic link와 correspond x -> 두 function 모두 same results

int lstat(const char *restrict path, struct stat *restrict buf);
//return information about the link

int stat(const char *restrict path, struct stat *restrict buf);
//return information about the file referred to by the link
```



- struct stat ⇒ link file structure
  - defined in <sys/stat.h>
  - field
    - dev_t st_dev : device ID of device containing file
    - ino_t st_ino : file serial number
    - mode_t st_mode : file mode
      - the access permissions of the file and the type of file
      - POSIX : macro 존재

        ex) S_ISDIR → dir인지 확인하는 macro

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

int isdirectory(char *path) {
  struct stat statbuf;

  if (stat(path, &statbuf) == -1)
    return 0;
  else
    return S_ISDIR(statbuf.st_mode);
}
```

    - nlink_t st_nlink : number of hard links
    - uid_t st_uid : user ID of file
    - gid_t st_gid : group ID of file
    - off_t st_size : file size in bytes(regular files), path size(symbolic link)
    - time_t st_atime : time of last access
    - time_t st_mtime : time of last data modification
    - time_t st_ctime : time of last file status change

```c
//Example - printaccess.c → 마지막 접근 시간 출력 method

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

void printaccess(char *path) {
  struct stat statbuf;

  if (stat(path, &statbuf) == -1)
    perror("Failed to get file status");
  else
    printf("%s last accessed at %s", path, ctime(&statbuf.st_atime));
}
                                  ↳ 형변환 ↔ time_t

//ctime : string type으로 저장할 수 있도록 static storage 사용
//-> 두번째 호출에는 접근하는 시간인 string으로 덮어 쓰여짐
```

UNIX File Implementation

- Implementation
  - OS마다 file의 확장자 존재 → file type
  - but, UNIX는 구분을 크게 하지 않음 → 사용자의 입장에서 구분하기 위해 존재
  - UNIX는 실행 가능, 실행 가능하지 않음 정도만 구분

  - UNIX file → a modified tree structure이며 directory entry 존재
  - Directory entry → filename , inode(a reference to a fixed-length structure)

inode

file에 대한 정보를 담고 있는 data structures. (file 생성될 때 생성됨)

- 각각의 file이 가지고 있음 → file의 객체 개수만큼 inode 객체 생성 → inode number로 구분

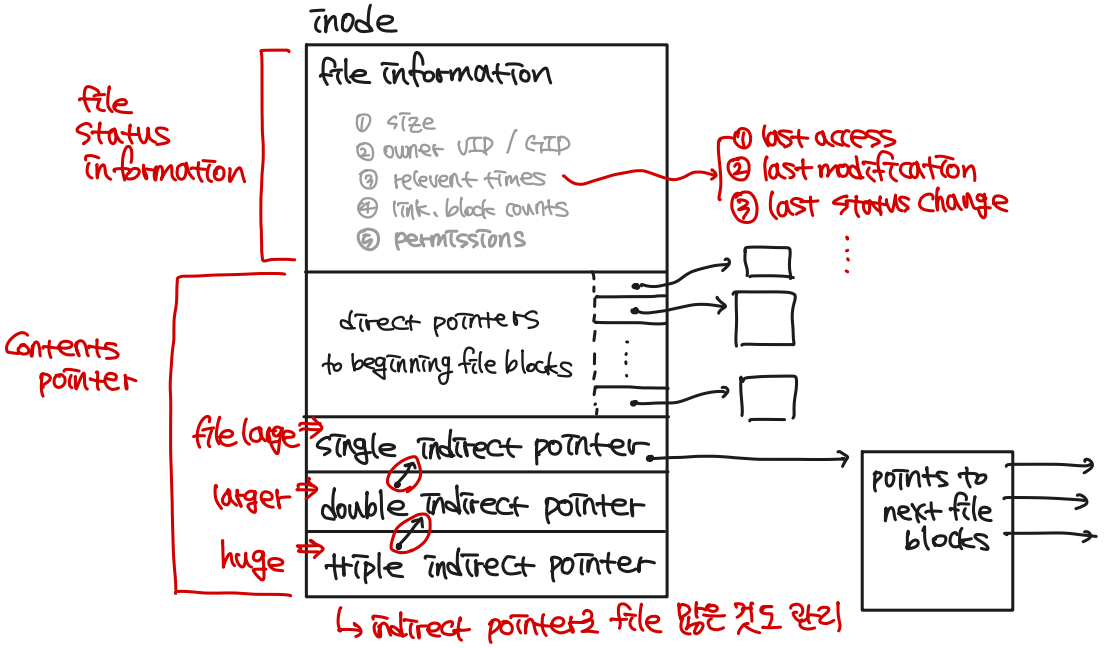
  ↳ ls -i : inode num 확인

- **inode가 가지고 있는 정보**
  - user, group, ownership
  - access mode (read, write, execute permissions)

⑤
- type of file

①
- inode의 크기와 max ②개수(=만들 수 있는 file 개수)는 정해져 있음
  - inode에 file의 content 값이 직접 들어가지는 않기에 크기가 고정되어 있음.
    - content에 접근할 수 있는 pointer + status information만 존재

+) stat, fstat : file 혹은 link(file)에 대한 정보 가져올 때 inode에서 가져온 뒤 structure stat에 담음



Directory implementation

a directory = a file containing a correspondence between file name and file location.

a directory entry = inode number + file name

① ② dir entry

- **access to inode**
  1. pathname에 의해 file 지정
  2. OS : file name, inode number를 알아내기 위해 file system tree 탐색
  3. OS : inode에 접근해서 file의 다른 정보들을 접근 가능

↳ inode의 번호!

- **advantage of inode + filename (따로 관리의 장점)**

↳ file name + inode

  1. Changing the filename : changing only the directory entry
  2. file move : directory entry 변경만 해도 가능(file node는 실제로 이동 x)

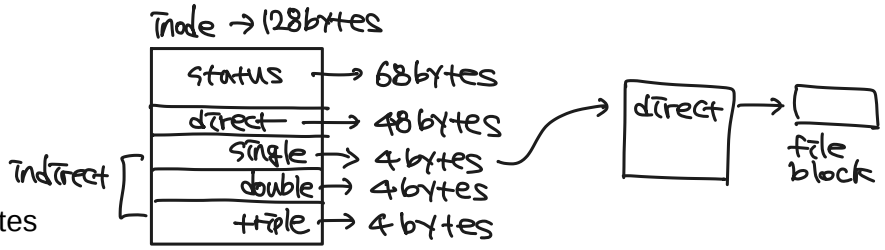
(만약 inode 새로 만들고, contents block도 새로 작성해버리면 문제없음)

3. Only one physical copy of the file needs to exist on disk, but the file may have several names or the same name in different directories

   → 같은 file을 가리키는 같은 dir 안에 다른 이름 혹은 다른 dir 안에 같은 이름 가능

4. Directory entries are small → 각 file에 대해 대부분의 정보가 inode에 있음

- **Exercise**
  - inode : 128 bytes
  - pointers : 4 bytes long
  - status information : 68 bytes
  - block size : 8 kilo bytes = 8KB, block pointers : 32 bits = 4 bytes



⇒하나의 pointer가 몇개의 file block 개수 representation에 따라 다름.

1. How much room is there for direct pointers in the inode? direct 공간 / pointer 하나의 크기
   ① direct → 48bytes (∵ 128 − (68 + 4×3) = 48)
   ② pointer → 하나 4byte ⇒ 12개 들어갈 수 있음 ∴ 12개
2. How big a file can be represented with direct pointers? Block size × pointer 개수
   ① block → 8k = 8×2^10 bytes = 8192 (block size)
   ② block × pointer 개수 8192×12 = 98,304 bytes (=96KB) ∴ 96KB
3. What about single indirect pointer?
   ① block의 pointer 개수 → 8k/4 = 8192/4 = 2048
   ② block × pointer 개수 2048 × 8192 = 16,777,216 bytes ∴ 16MB
   ↳ (2×2^10) × (8×2^10) = 2×8×2^20 = 16MB

Link

an association btw a filename(in directory entries) and an inode.

- UNIX의 link : ① hard links, ② symbolic links
  ① 연관수 ② 연관↓

- **link counter** → Hard Link만 사용
  - 대부분의 file system들은 hard link를 위해 link counter라는 counting reference 제공
  - integer value로 link의 총 개수를 저장 ⇒ inode 갯수 유지

1. count ++1 → new link is created
2. count --1 → a link is removed
3. count = 0

   → access 하기 위해 file을 열어 놓은 process가 없을 경우,

OS가 file의 data 공간을 자동으로 할당 해제 했다는 뜻

- **Hard Link**
  - filename들을 inode에 directly하게 link = directory entries
  - hard link는 따로 만들어지는 게 x , file 생성 시에 만들어진 directory entry가 hard link

i) create

  - 이미 존재하는 file에 대한 new hard link
    - new directory entyr 생성, 추가적인 디스크 공간 할당은 x
    - link count ++

ii) remove hard link

  - ***rm*** command , or ***unlink*** system call 사용
  - link count - -
  - count > 0 → directory entry만 삭제
  - count = 0 → inode도 삭제

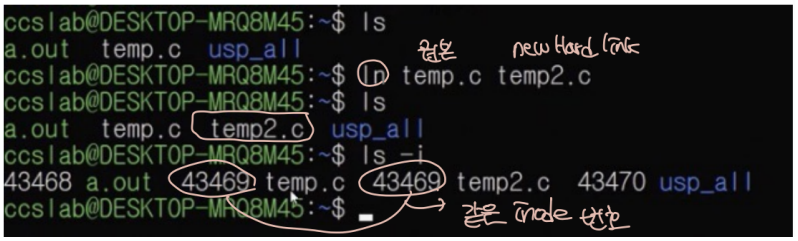
iii) Hard link APIs

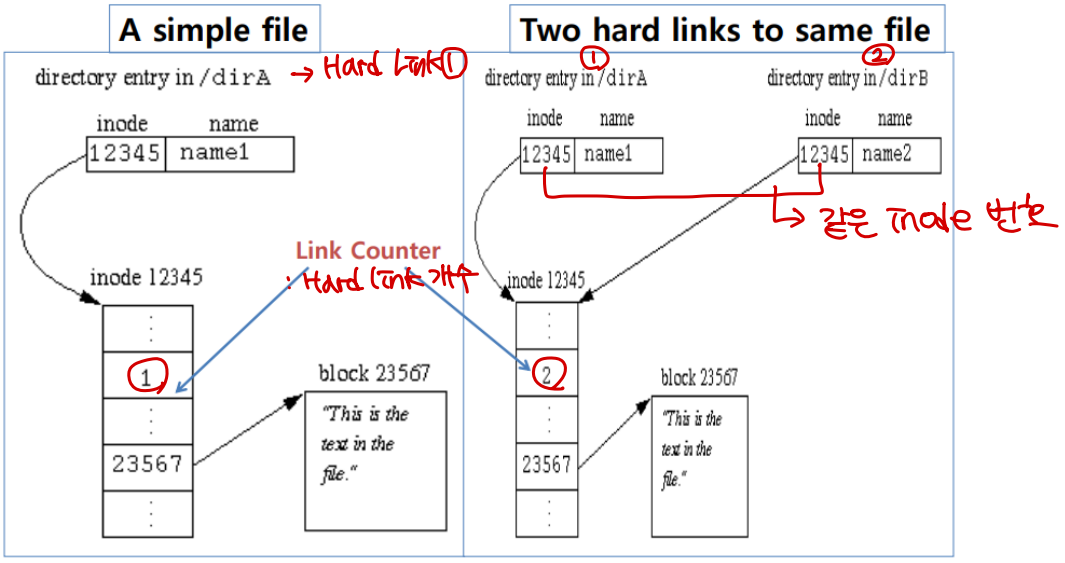
```
#include <stdio.h>

int link(const char *path1, const char *path2);
//creates a new directory entry(=hard link) for the existing file

//path1 : 원본 file 경로명
//path2 : new hard link 경로명

int unlink(const char *path);
//removes the directory entry

//path : 삭제할 hard link 경로명
//path의 link counter가 0이고, file을 open하고 있는 process가 없다면
//unlink는 file에 할당된 공간을 해제
```

  - example

    $ **ln** /dirA/name1 /dirB/name2

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

if (link("/dirA/name1", "/dirB/name2") == -1)
  perror("Failed to make a new link in /dirB");
```





② • **Symbolic Link** (= Soft Link)

- a special type of file that contains the name of another file or directory
- filename들을 inode에 간접적으로 link

+) lstat(← , O) → symbolic link일 경우, 이 link file의 status info return

- OS가 symbolic link를 이용하여 접근하는 방법
  1. inode에서 file status information을 통해 symbolic link 임을 알게 됨
  2. 원본 file의 경로를 탐색 멈추지 않고 계속 함
  3. 원본 file의 inode에 접근 후 file open

③ create a symbolic link with the command

+) Symbolic link 생성
↳ link에 대한 dir entry 생성
↳ inode : 원본 file 경로 포함

- ***ln -s*** target link_name

- link count에는 아무런 영향 X → 따로 사용하지 X
  → 다지워더라도 0이 되지 않음.

sym link APIs

```
#include <unistd.h>
int symlink (const char *path1, const char *path2);
//creates a new symbolic link

//path1 : 원본 file 경로명
//path2 : new hard link 경로명
```

- example

  $ ln -s /dirA/name1 /dirB/name2
  ↳ symbolic

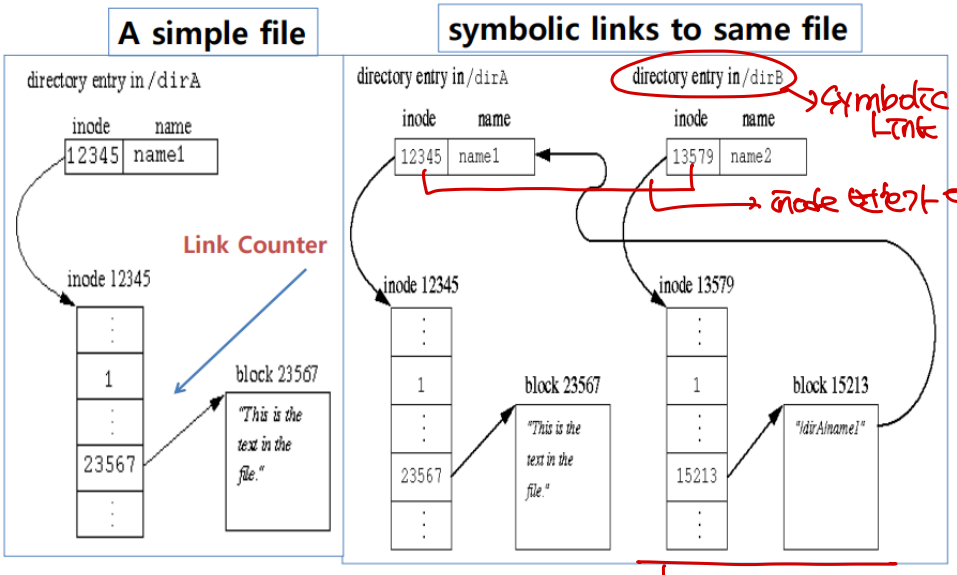
```
#include <stdio.h>
#include <unistd.h>

if (symlink("/dirA/name1", "/dirB/name2") == -1)
  perror("Failed to create symbolic link in /dirB");
```

→ error ⇒ return -1

```
ccslab@DESKTOP-MRQ8M45:~$ ln -s temp.c temp-sym.c
ccslab@DESKTOP-MRQ8M45:~$ ls -l
43468 a.out    320 temp-sym.c  43469 temp.c  43469 temp2.c  43470 usp_all
ccslab@DESKTOP-MRQ8M45:~$ ls -l
total 32
-rwxrwxrwx  1 ccslab ccslab 16688 Sep  1 00:39 a.out
lrwxrwxrwx  1 ccslab ccslab     6 Sep 19 21:55 temp-sym.c -> temp.c
-rw-rw-rw-  2 ccslab ccslab    63 Sep  1 00:39 temp.c
-rw-rw-rw-  2 ccslab ccslab    63 Sep  1 00:39 temp2.c
drwxr-xr-x 25 ccslab ccslab  4096 May 22  2003 usp_all
```

symbolic

symbolic



↳ symbolic을 지우는 것은 그냥 dir entry 하나 지우는 것과 동일

+) 만약 name2에 access 후 file name 변경
⇒ name1의 file name 변경됨 (Hard, Symbolic 모두 동일)

+) 만약 name1 삭제 ⇒ 원본 삭제면 sym에서도 활용 X → Point X
삭제 후 name1을 같은 dir에 생성 → Hard는 이전 것, sym은 새로운 file 가리킴.