Operating System:
Files and Directories

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

Persistent Storage

파워가 공급되지 않아도 데이터유지

- Keep a data **intact** even if there is a power loss
  - Hard disk drive
  - Solid-state storage device

- Two key abstractions in the virtualization of storage
  - File
  - Directory

어떻게관리 →

어떻게 유저에게 보여질 건지

실제 OS 와 어떻게 매핑?

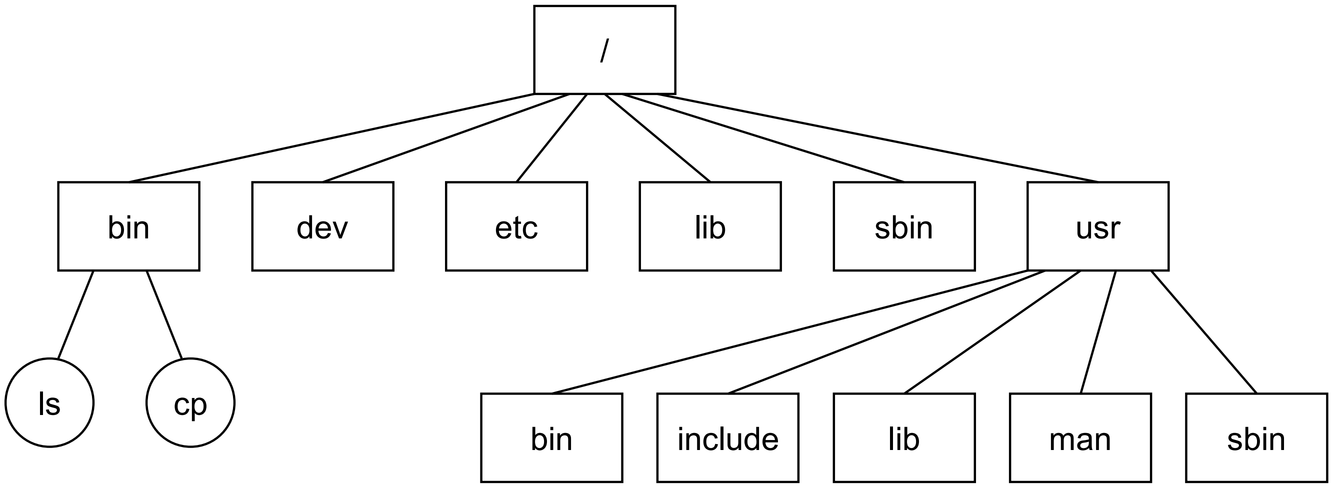
File System

- File system
  - A software for organizing data on a storage device

계층적구조

User's viewpoint

유저관점



system 내에서 관리하는 관점

Storage management's viewpoint

HDD

NAND flash memory

SSD

File System (Cont.)

- File system of the user's viewpoint
  - File system interface
  - How to show the file system to user?
  - File, directory, attribute, and operation
    read, write ...
  - tree structure

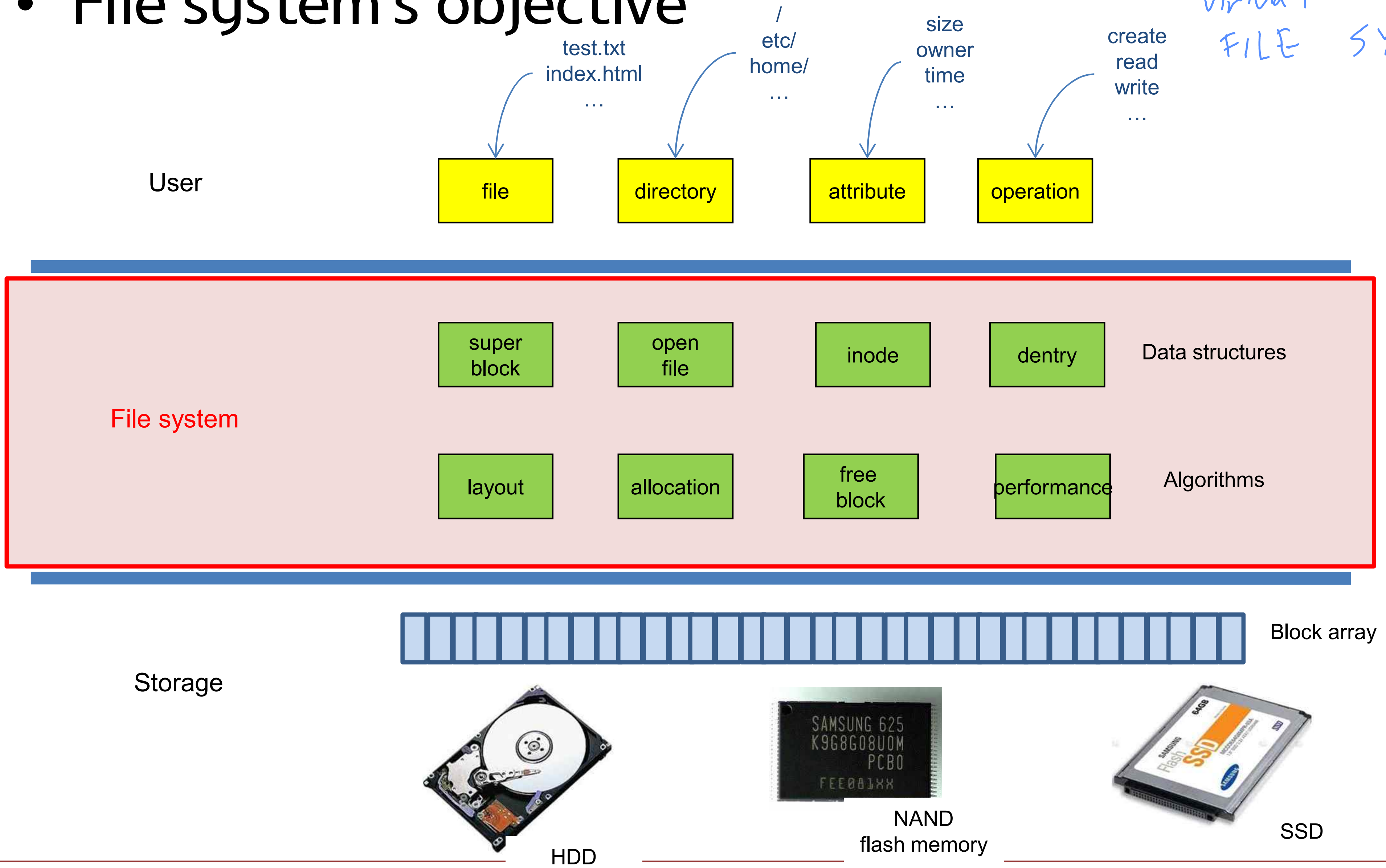
- File system of the storage management viewpoint
  - File system implementation
  - How to map the logical file system to the storage device?
  - Layout, data structures, and algorithms
  - It is required to understand the storage internals

File System (Cont.)

VFS

Virtual FILE SYSTEM

- File system's objective



File

- A linear array of bytes

- Each file has low-level name as inode number
  - The user is not aware of this name

- File system has a responsibility to store data persistently on disk

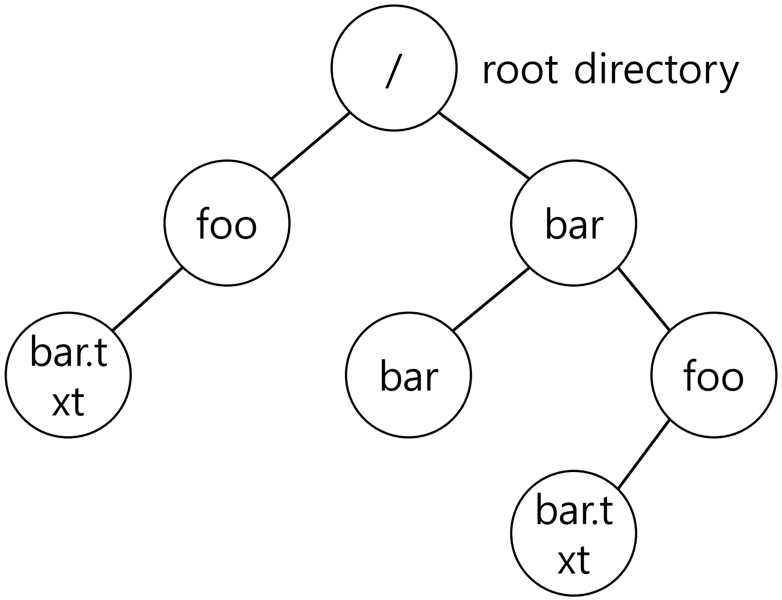
Directory

- Directory is like a file, also has a low-level name
  - It contains a list of (user-readable name, low-level name) pairs
  - Each entry in a directory refers to either *files* or other *directories*

pair

- Example)
  - A directory has an entry ("foo", "10")
    - ➢ A file "foo" with the low-level name "10"

Directory Tree (Directory Hierarchy)



An Example Directory Tree

Valid files (absolute pathname) :

/foo/bar.txt
/bar/foo/bar.txt

Valid directory :

/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories

Creating Files

- Use `open()` system call with `O_CREAT` flag

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

  - `O_CREAT` : create file
  - `O_WRONLY` : only write to that file while opened
  - `O_TRUNC` : make the file size zero (remove any existing content)

  - `open()` system call returns file descriptor
    - *File descriptor* is an <u>integer</u>, and is used to access files

Reading and Writing Files

- An Example of reading and writing 'foo' file

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

➢ echo : redirect the output of echo to the file foo

➢ cat : dump the contents of a file to the screen

How does the cat program access the file foo?

We can use strace to trace the system calls made by a program

위의 동작이 어떻게 진행되는 건가?
strace로 확인가능.

Reading and Writing Files (Cont.)

64bit offset

```
prompt> strace cat foo
…
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)           = 6
write(1, "hello\n", 6)             = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)                  = 0 // 0: no bytes left in the file
close(3)                           = 0
…
prompt>
```

– `open` (file descriptor, flags)
  - Return file descriptor (3 in example)
  - File descriptor 0, 1, 2, is for standard input/ output/ error
– `read` (file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it read
– `write` (file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it write

Reading and Writing Files (Cont.)

- Writing a file (A similar set of read steps)
  - A file is opened for writing (`open()`)
  - The `write()` system call is called
    - Repeatedly called for larger files
  - `close()`

Reading And Writing, But Not Sequentially

- An open file has a current offset
  - Determine **where** the next read or write will begin reading from or writing to within the file

- Update the current offset
  - **Implicitly:** A read or write of `N` bytes takes place, `N` is added to the current offset
  - **Explicitly:** `lseek()`

Reading And Writing, But Not Sequentially (Cont.)

```
off_t lseek(int fildes, off_t offset, int whence);
```

- `fildes` : File descriptor
- `offset` : Position the file offset to a particular location within the file
- `whence` : Determine how the seek is performed

- examples

```
off_t curpos;
curpos = lseek(fd, 0, SEEK_CUR);   // get the current offset

lseek(fd, 0, SEEK_SET);
lseek(fd, 0, SEEK_END);
lseek(fd, -10, SEEK_CUR);
lseek(fd, 100, SEEK_END);
```

Writing Immediately with `fsync()`

- The file system will **buffer** writes in memory for some time
  - Ex) 5 seconds, or 30
  - Performance reasons
- At that later point in time, the write(s) will **actually be issued** to the storage device
  - Write seem to complete quickly
  - Data can be lost (e.g., the machine crashes)

Writing Immediately with `fsync()` (Cont.)

- However, some applications require more than eventual guarantee
  - Ex) DBMS requires force writes to disk from time to time Database management system
- `off_t fsync(int fd)`
  - Filesystem forces all dirty (i.e., not yet written) data to disk for the file referred to by the file description
  - `fsync()` returns once all of theses writes are complete

Writing Immediately with fsync() (Cont.)

- Example of `fsync()`

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert (fd > -1)
int rc = write(fd, buffer, size);
assert (rc == size);
rc = fsync(fd);
assert (rc == 0);
```

  - In some cases, this code needs to `fsync()` the directory that contains the file `foo`

Renaming Files

- `rename(char* old, char *new)`
  - Rename a file to different name
  - It implemented as an **atomic call**
    - Ex) Change from `foo` to `bar`:

```
prompt> mv foo bar          // mv uses the system call rename()
```

    - Ex) How to update a file atomically:

```
int fint fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Getting Information About Files

- `stat(), fstat():` Show the file metadata
  - **Metadata** is information about each file
  - Ex) Size, Low-level name, Permission, …
  - `stat` structure is below:

```
struct stat {
    dev_t st_dev;           /* ID of device containing file */
    ino_t st_ino;           /* inode number */
    mode_t st_mode;         /* protection */
    nlink_t st_nlink;       /* number of hard links */
    uid_t st_uid;           /* user ID of owner */
    gid_t st_gid;           /* group ID of owner */
    dev_t st_rdev;          /* device ID (if special file) */
    off_t st_size;          /* total size, in bytes */
    blksize_t st_blksize;   /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;     /* number of blocks allocated */
    time_t st_atime;        /* time of last access */
    time_t st_mtime;        /* time of last modification */
    time_t st_ctime;        /* time of last status change */
};
```

Getting Information About Files (Cont.)

- To see stat information, you can use the command line tool `stat`

```
prompt> echo hello > file
prompt> stat file

File: 'file'
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

  – File system keeps this type of information in a `inode` structure

Removing Files

- rm is Linux command to remove a file
  - rm call unlink() to remove a file

```
prompt> strace rm foo
…
unlink("foo")                = 0      // return 0 upon success
…
prompt>
```

Why it calls `unlink()`? not "`remove` or `delete`"
We can get the answer later

Making Directories

- `mkdir()` : Make a directory

```
prompt> strace mkdir foo
…
mkdir("foo", 0777)                      = 0
prompt>
```

  - When a directory is created, it is empty
  - Empty directory have two entries:
    . (itself), .. (parent)

```
prompt> ls -a
./        ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

Reading Directories

- A sample code to read directory entries (like `ls`)

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");                   // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)         // read one directory entry
    {
        // print outthe name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                             // close current directory
    return 0;
}
```

  - The information available within `struct dirent`

```
struct dirent {
    char            d_name[256];             /* filename */
    ino_t           d_ino;                   /* inode number */
    off_t           d_off;                   /* offset to the next direct */
    unsigned short  d_reclen;                /* length of this record */
    unsigned char   d_type;                  /* type of file */
}
```

Deleting Directories

- `rmdir()`: Delete a directory
  - Require that the directory be **empty**
  - If you call `rmdir()` to a non-empty directory, it will fail
    - I.e., Only has "." and ".." entries

Hard Links

- `link(old pathname, new one)`
  - Link a new file name to an old one
  - Create another way to refer to *the same file*
  - The command-line link program : `ln`

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Hard Links (Cont.)

- The way `link` works:
  - **Create** another name in the directory
  - **Refer** it to the same inode number of the original file
    - ➢ The file is not copied in any way
  - Then, we now just have two human names (`file` and `file2`) that both refer to the same file

Hard Links (Cont.)

- The result of `link()`

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

  - Two files have **same inode** number, but two human name (file, file2)
  - There is **no difference** between file and file2
    - Both just links to the underlying metadata about the file

Hard Links (Cont.)

- Thus, to remove a file, we call `unlink()`

```
prompt> rm file
removed 'file'
prompt> cat file2          // Still access the file
hello
```

  - *reference count*
    - ➢ Track how many different file names have been linked to this inode
    - ➢ When `unlink()` is called, the reference count decrements
    - ➢ If the reference count reaches **zero**, the file system free the inode and related data blocks → truly "delete" the file

Hard Links (Cont.)

- The result of `unlink()`
  - `stat()` shows the reference count of a file

```
prompt> echo hello > file                    /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...             /* Link count is 1 */
prompt> ln file file2                        /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...             /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...             /* Link count is 2 */
prompt> ln file2 file3                       /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...             /* Link count is 3 */
prompt> rm file                              /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...             /* Link count is 2 */
prompt> rm file2                             /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...             /* Link count is 1 */
prompt> rm file3
```

Symbolic Links (Soft Link)

- Symbolic link is more useful than Hard link
  - Hard Link cannot create to a directory
  - Hard Link cannot create to a file to other partition
    - Because inode numbers are only unique within a file system
- Create a symbolic link: `ln -s`

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- What is different between *Symbolic link* and *Hard Link*?
  - Symbolic links are **a third type** the file system knows about

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...        // Actually a file it self of a different type
```

  - The size of symbolic link (`file2`) is **4 bytes**

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../                // directory
-rw-r-----  1 remzi remzi    6 May 3 19:10 file               // regular file
lrwxrwxrwx  1 remzi remzi    4 May 3 19:10 file2 -> file      // symbolic link
```

    - A symbolic link holds the <u>pathname</u> of the linked-to file as the data of the link file

Symbolic Links (Cont.)

- If we link to a longer pathname, our link file would be bigger

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May 3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 -> alongerfilename
```

Symbolic Links (Cont.)

- Dangling reference
  - When remove an original file, symbolic link points noting

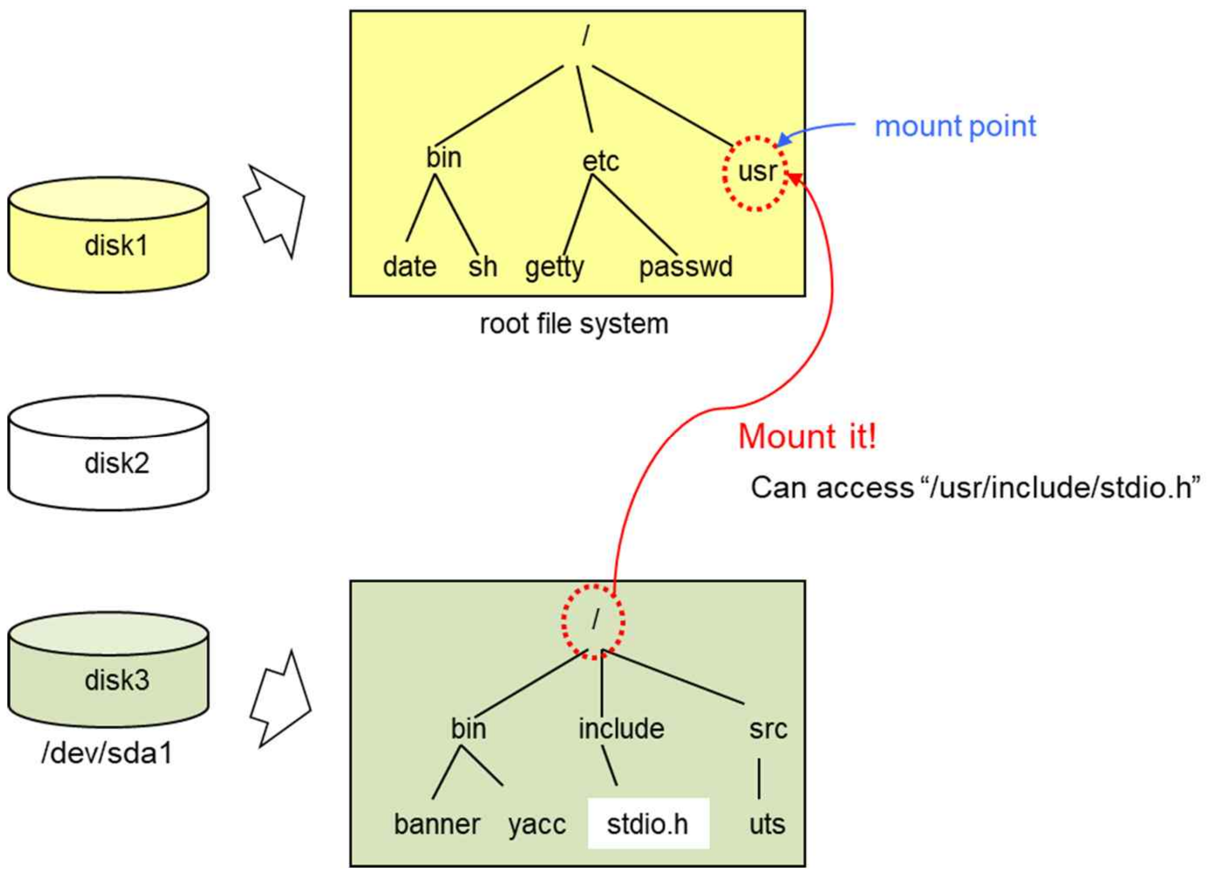
```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file                // remove the original file
prompt> cat file2
cat: file2: No such file or directory
```

Making and Mounting a File System

- `mkfs` tool : Make a file system
  - Write an <u>empty file system</u>, starting with *a root directory*, on to a disk partition
  - Input:
    - ➢A device (such as a disk partition, e.g., `/dev/sda1`)
    - ➢A file system type (e.g., `ext3`)
  - Example
    - ➢`prompt> mkfs  -t  ext3  /dev/sda1`

Making and Mounting a File System (Cont.)

- `mount()`
  - Take an existing directory as a target **mount point**
  - `prompt> mount -t ext3 /dev/sda1 /usr`



Making and Mounting a File System (Cont.)

- `mount` **program**: show **what is mounted** on a system

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

- ➢ `ext3`: A standard disk-based file system
- ➢ `proc`: A file system for accessing information about current processes
- ➢ `tmpfs`: A file system just for temporary files
- ➢ `AFS`: A distributed file system