Linux File System Programming

HGU

File and filesystem in Linux

- File is the most fundamental and basic abstraction in Linux
- Linux follows ‘Everything-is-a-file’ philosophy (File, Directories, devices)
- Interaction often involve reading and writing file
- a File must first be opened for read or write.
- Open file are identified by a unique descriptor, file descriptor (fd)
- file descriptor is shared with user space, for file access

Regular Files: file offset

- Regular Files are considered just as *Byte Stream*
  - no further organization or formatting is specified for a file. The bytes may have any values, and they may be organized within the file in any way.
- File Position
  - Read or Write Operation can be done in any Location within a file
  - The Location within a File is called the *File Position* or *File Offset*
  - When a file is first opened, the *File Position* is set zero.
  - As bytes are read from or written to the file, the file position increases accordingly, byte-by-byte.
  - The file position may also be set manually to a given value

Regular File : length

- The size of a file is measured in bytes, and is called its *length*.
- The length is simply the number of bytes in the *linear array* that make up the file
- ***Length*** can be changed by '***truncation***' operation
  - A file can be truncated to a new size *smaller than its original size*, which results in bytes being *removed from the end of the file*.
  - A file can also be "*truncated*" to a new size *larger than its original size*. In that case, the new bytes are filled with zeros.
  - A file may be empty (have a *length of zero*).

Regular File : Concurrent File Access

- A Single File can be opened more than once, by a different or even the same process.
- Each open instance of a File is given a unique file descriptor
- Processes can share their file descriptors, allowing a single descriptor to be used by more than one process
- Multiple Processes are free to read from and write to the same file at the same time. The results are generally *unpredictable.*

i-node

- Instead of filename, file is referenced by i-node number (i-number) in Kernel
- An i-node stores metadata associated with a file:
  - modification timestamp,
  - owner,
  - type,
  - length,
  - the location of the file's data—but no filename!
- The i-node is both a physical object, located on disk in Unix-style filesystems, and a conceptual entity, represented by a data structure in the Linux kernel.

Directory and Link

- **Accessing a file via its i-node number** is cumbersome (and also a potential security hole),
  - so files are always opened from user space ***by a name***
- Directories are used to act as a *mapping* of human-readable names to i-node numbers.
  - A name and i-node pair is called a ***link.***
  - Directory is a file that contains only a *mapping of names to i-nodes*
- When an application requests that a given filename be opened, the kernel opens the directory containing the filename and searches for the given name.
  - From the filename, the kernel obtains the i-node number.
  - From the i-node number, the i-node is found.
  - From the i-node, metadata of the file, including **on-disk location of the file** is found.
- Directories can be nested, creating a hierarchy, enabling pathnames like /home/user1/test

The Stat Family

- a family of functions for obtaining the **metadata of a UNIX file**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat (const char *path, struct stat *buf); // returns information of file of path
int fstat (int fd, struct stat *buf);  // returns information about the file of fd
int lstat (const char *path, struct stat *buf); // lstat() is identical to stat()
                                                // except in the case of symbolic link (link itself)
```

struct stat

```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino;  /* file's inode number */
    mode_t st_mode; /* file's mode bytes (permissions) */
    nlink_t st_nlink; /* number of hard links at the file*/
    uid_t st_uid; /* user ID of owner of the file */
    gid_t st_gid; /* group ID of owner of the file */
    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; /* last access time */
    time_t st_mtime; /* last modification time */
    time_t st_ctime;  /* last status change time */
};
```

stat structure declared in <bits/stat.h>

Sample Code (Retrieving the Size of a File)

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

int main (int argc, char *argv[]) {
    struct stat sb;
    int ret;
    if (argc < 2) {
        fprintf(stderr, "usage: %s <file>₩n", argv[0]);
        return 1;
    }
    ret = stat(argv[1], &sb);
    if(ret) {
        perror("stat");
        return 1;
    }
    printf("%s is %ld bytes₩n", argv[1],sb.st_size);
     return 0;
}
```

```
yk@peace:~/systemprj/fileio_test$ ls -l stat1
-rwxrwxr-x 1 yk yk 8904  7월  30 11:35 stat1
yk@peace:~/systemprj/fileio_test$ ./stat1 stat1
stat1 is 8904 bytes
```

Permissions

chmod() and fchmod() set a file's permissions to mode

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

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
// On success, all three calls return 0.
// On Failure, returns -1
```

Sample Code to change file mode

```
#include <sys/types.h>
#include <sys/stat.h>
int ret;
main(int argc, char *argv[]) {
  /* set map.png in the current directory to
   * owner-readable and writable. (same as chmod 600 ./map.png)
   */
  ret = chmod("./map.png", S_IRUSR | S_IWUSR);
  if (ret) {
      perror("chmod");
      return 1;
  }
  return 0;
}
```

Ownership

- change the field of st_uid and st_gid to change the owner and group of a file respectively.

```
#include <sys/types.h>
#include <unistd.h>
// changes the owner and group of a file
// if owner and group is -1, it is not changed by chown and lchown
// On success, all three calls return 0. On Failure, returns -1
int chown (const char *path,  uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
        // lchown changes owner of symbolic link itself not the original file
int fchown (int fd,                uid_t owner, gid_t group);
```

Sample Code to change Ownership

```c
#include <stdio.h>
#include <unistd.h>
#include <grp.h>
int main( ) {
   struct group *gr;
   int ret;

   gr = getgrnam("prof");
   if(gr == NULL) {
      perror("getgrnam");
      return 1;
   }
   /* set mainifest.txt's group to 'prof */
   ret = chown ("chown-test.txt", -1, gr->gr_gid);
   if(ret != 0)
      perror("chmod");
  return 0;
}
```

```
yk@peace:~/systemprj/fileio_test$ ls -l
total 16
-rwxrwxr-x 1 yk yk 8712  7월 30 11:22 chown1
-rw-rw-r-- 1 yk yk  481  7월 30 11:22 chown1.c
-rw-rw-r-- 1 yk yk    0  7월 30 08:55 chown-test.txt
yk@peace:~/systemprj/fileio_test$ sudo ./chown1
[sudo] password for yk:
yk@peace:~/systemprj/fileio_test$ ls -l
total 16
-rwxrwxr-x 1 yk yk   8712  7월 30 11:22 chown1
-rw-rw-r-- 1 yk yk    481  7월 30 11:22 chown1.c
-rw-rw-r-- 1 yk prof    0  7월 30 08:55 chown-test.txt
yk@peace:~/systemprj/fileio_test$
```

Directories

- Directory contains a list of filename, each of which maps to an inode number
  - Each name is called directory entry
  - Each name-to-inode mapping is called a link
- Directory contents are a listing of all the filenames in that directory
- Steps for Opening a file in a given directory
  1. Look up filename in that directory
  2. find corresponding inode number
  3. pass the inode number to the filesystem to find physical location of the file
- Directories can contain **other directories** (subdirectory)
  - All the directories (except /) are subdirectory of their parent directory
  - tow special subdirectories: dot(.) and dot-dot(..)
- pathname
  - absolute pathname : begins with the / directory
  - relative pathname : begins with **current working directory**

Current Working Directory

- Def: **The starting point of** the **relative pathname**
- Every process has a **current working directory** which it initially inherits from its parent process.
- A process can both obtain and change the current working directory

```
#include <unistd.h>
char * getcwd(char *buf, size_t size);
```

- On success, copies the current working directory as an **absolute pathname** into buffer (buf) and returns pointer to buf
- On failure, returns NULL and set errno

```
// cwd.c
main() {
  char cwd[BUF_LEN];

  if (!getcwd (cwd, BUF_LEN)) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
  }
  printf ("cwd = %s₩n", cwd);
}
```

Change the Current Working Directory

- When a user log into a system, the starting current working directory is set as the home directory (/etc/passwd)
- Linux provides two system calls for **changing current working directory**

```
#include <unistd.h>
int chdir (const char *path);
  // changes current working directory by a given path

int fchdir (int fd);// changes cwd with file (fd)
```

- On success returns 0
- On failure returns -1

Sample Code with Current Working Directory

```
char *swd;
int ret;

/* save the current working directory */
if(!(swd = getcwd (NULL, 0)){
     perror ("getcwd");
     exit (EXIT_FAILURE);
}
/* change to a different directory */
if(chdir (some_other_dir) <0) {
    perror ("chdir");
    exit (EXIT_FAILURE);
}
/* do some other work in the new directory...
*/
```

```
/* return to the saved directory */
if(!chdir (swd) ) {
    perror ("chdir");
    exit (EXIT_FAILURE);
}
free (swd)
```

Create Directories

#include <sys/stat.h>

#include <sys/types.h>

int mkdir (const char *_path_, mode_t _mode_);

- mkdir() creates the directory **path**, which may be relative or absolute, with the permission bits **mode** (as modified by the current umask)
  - permission = (mode & ~umask & 01777)
- Return Value
  - 0 on Success
  - -1 on Failure

Remove Directories

include <unistd.h>

int rmdir (const char *path);

- rmdir( ) removes path from the filesystem once its files have been removed.
- Return Value
  - 0 on Success
  - -1 on Failure

```
int ret;
/* remove the directory /home/barbary/maps
*/
ret = rmdir ("/home/barbary/maps");
if (ret)
    perror ("rmdir");
```

Read Directory's Contents

#include <sys/types.h>
#include <dirent.h>
DIR * opendir (const char *name);
- opendir() creates a directory stream representing the directory

#define _BSD_SOURCE /* or _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>
int dirfd (DIR *dir);
- On success dirfd( ) returns the file descriptor backing the directory stream.
- On Failure it returns -1

Reading from a Directory Stream

```
#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir (DIR *dir);
```

- Once created directory stream with opendir(), program can begin reading entries from the directory.
- A successful call to readdir( ) returns entry **one by one** from a given DIR. The dirent structure represents a directory entry.
- readdir() return NULL on End-of-List or Error (errno=EBADF : invalid dir)

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

Sample Code(1) for Reading Directory Stream

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

void display_file_info(
  const char *path,
  const struct dirent *entry)
{
  struct stat st ; //file_stat
  char full_path[1024];

  // Create the full path to the file
  snprintf(full_path,sizeof(full_path),
        "%s/%s", path, entry->d_name);
```

```
drwxr-xr-x 3  4096 Sep 19 12:45 mydir
-rw-r--r--  1  1234 Sep 18 10:30 myfile.txt
-rwxr-xr-x  1  4321 Sep 17 08:20 script.sh
```

```
 // Get file statistics using stat()
 if (stat(full_path, &st) == -1)
        perror("stat");
        return;
  }

// Display file type and permission
   printf("%c%c%c%c%c%c%c%c%c ",
      S_ISDIR(st.st_mode) ? 'd' : '-',
      st.st_mode & S_IRUSR ? 'r' : '-',
      st.st_mode & S_IWUSR ? 'w' : '-',
      st.st_mode & S_IXUSR ? 'x' : '-',
      st.st_mode & S_IRGRP ? 'r' : '-',
      st.st_mode & S_IWGRP ? 'w' : '-',
      st.st_mode & S_IXGRP ? 'x' : '-',
      st.st_mode & S_IROTH ? 'r' : '-',
      st.st_mode & S_IWOTH ? 'w' : '-',
      st.st_mode & S_IXOTH ? 'x' : '-');
```

```
   // Display number of links
   printf(" %lu", st.st_nlink);
   // Display file size
   printf(" %5ld", st.st_size);

    // Display modification time
    char time_buff[80];
    struct tm *time_info;
    time_info = localtime(&st.st_mtime);
    strftime(time_buff, sizeof(time_buff),
            "%b %d %H:%M", time_info);
    printf(" %s", time_buff);

    // Display file name
    printf(" %s\n", entry->d_name);
} // end of display_file_info
```

Sample Code(2) for Reading Directory Stream

```
int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *entry;

    // Use current directory if no directory is specified
    const char *dir_path = (argc > 1) ? argv[1] : ".";

    // Open the directory stream
    if ((dir = opendir(dir_path)) == NULL) {
        perror("opendir");
        return EXIT_FAILURE;
    }
```

```
    // Read directory entries and display them one by one
    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") != 0 &&
            strcmp(entry->d_name, "..") != 0) {
            display_file_info(dir_path, entry);
        }
    }
    // Close the directory stream
    closedir(dir);
    return EXIT_SUCCESS;
}
```

Closing the Directory Stream

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dir);
```

- closedir() closes directory dir that was open by diropen()

Sample Code for Reading Directory Stream

```
int find_file_in_dir (const char *path, const char *file)
{
  struct dirent *entry;
  int ret = 1;
  DIR *dir;

  dir = opendir (path);
  errno = 0;
  while ((entry = readdir (dir)) != NULL) {
   if (!strcmp(entry->d_name, file)) {
       ret = 0;
       break;
   }
 }
 if (errno && !entry)
   perror ("readdir");
 closedir (dir);
 return ret; // 0 on Success, non-Zero on Failure
}
```

- find_file_in_dir
  - searches the directory 'path' for a file
  - Returns 0 if 'file' exists in 'path' and a nonzero value otherwise

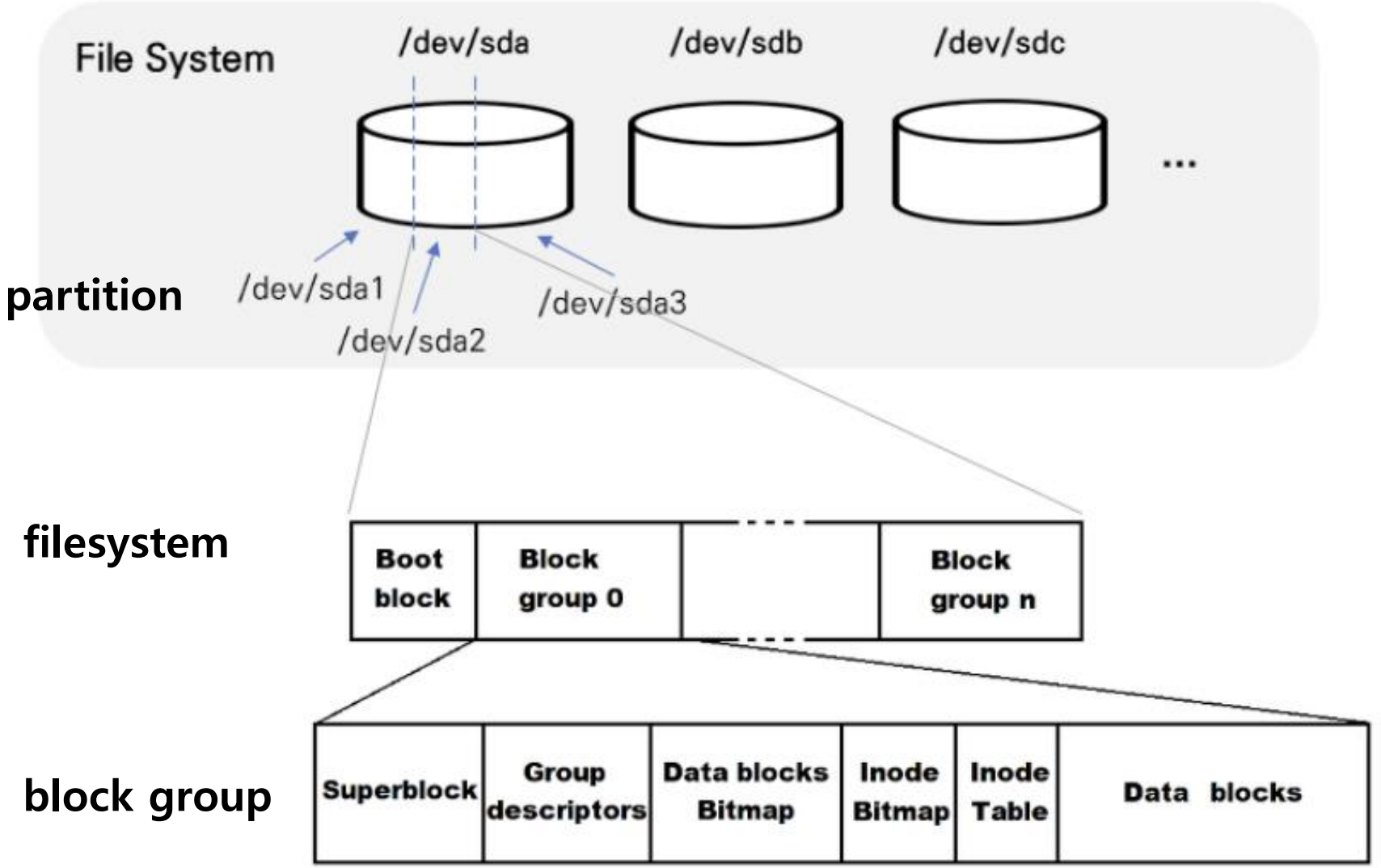
Filesystem API Summary

- **int stat (const char *path, struct stat *buf)**
- **int fstat (int fd, struct stat *buf)**
- **int chmod(const char *path, mode_t mode)**
- **int fchmod(int fd, mode_t mode)**
- **int chown (const char *path, uid_t owner, gid_t group)**
- **int fchown (int fd, uid_t owner, gid_t group)**
- **char * getcwd(char *buf, size_t size)**
- **int chdir (const char *path)**
- **int mkdir (const char *_path_, mode_t _mode_)**
- **int rmdir (const char *path)**
- **DIR * opendir (const char *name)**
- **int closedir (DIR *dir)**
- **struct dirent * readdir (DIR *dir)**

Linux File System Internals

HGU

File System: Disk and Partition



- Disk
- Partition
- Boot block
- Block Groups
  - super block
  - group descriptors
  - datablock bitmap
  - inode bitmap
  - inode table
  - data block

Making Filesystem (Linux Commands)

- Add New disk (SCSI disk)
  - echo "- - -" > /sys/class/scsi_host//host0/scan
- Making Partition
  - fdisk –l : list disk partition states
  - **fdisk /dev/sdb**
- Device should be formatted
  - **mkfs /dev/sdb1**
- Each Device should be mounted (linked) to a specific directory
  - **mount /dev/sdb1 / disk1**
  - .etc/fstab : automounting device list when booting system
- Checking Filesystem
  - df -h
- Mounted Device is accessed through a path from root (/)
  - cd /disk1

What is i-node?

- Linux filesystem has ‘blocks’ which hold data called ‘inode’ (**index node)** describing **metadata** of file: ownership, permission, time, size, and location of file’s data
- **i-node** means both
  - *physical block* located on disk in UNIX-like filesystem and
  - *conceptual entry* represented by data structure in Linux Kernel
- i-node is a **unique identifier for each file** (or directory) in a filesystem.

i-node and file copy and move

- Checking i-node information by Linux commands:

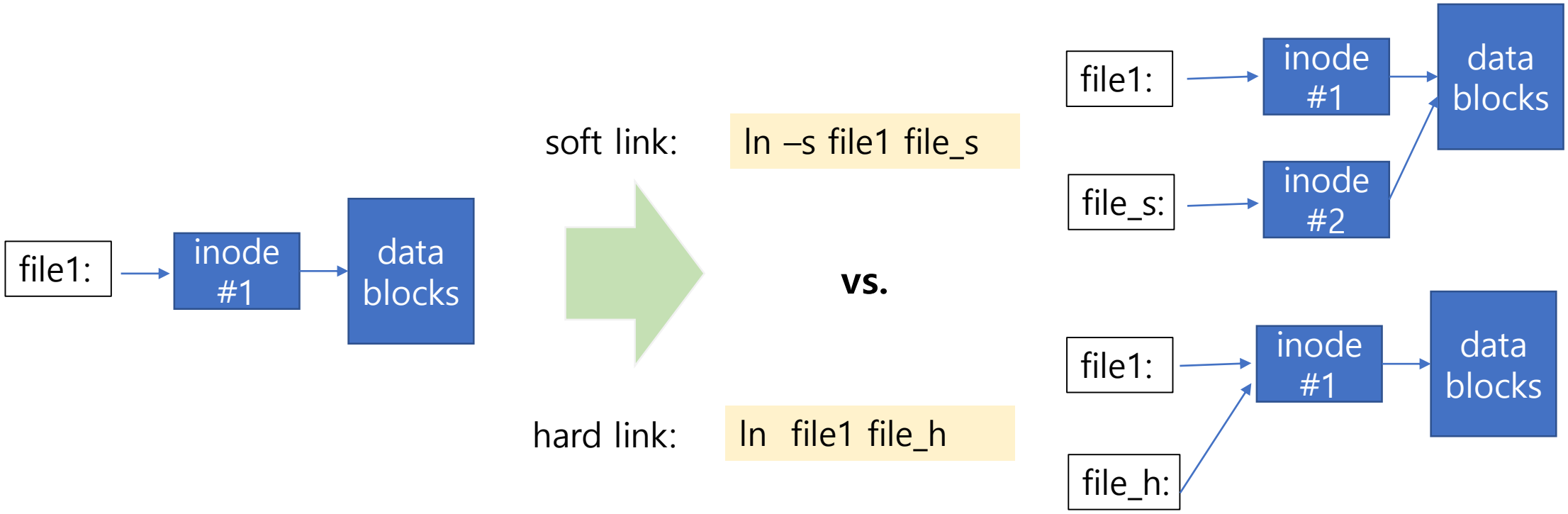
```
yk@peace:~/systemprj/fileio_test$ touch mytest
yk@peace:~/systemprj/fileio_test$ stat mytest
  File: 'mytest'
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 801h/2049d      Inode: 21977514    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1017/      yk)   Gid: ( 1019/      yk)
Access: 2024-08-03 20:29:49.956590670 +0900
Modify: 2024-08-03 20:29:49.956590670 +0900
Change: 2024-08-03 20:29:49.956590670 +0900
 Birth: -
yk@peace:~/systemprj/fileio_test$ ls -li mytest
21977514 -rw-rw-r-- 1 yk yk 0  8월  3 20:29 mytest
```

- i-node with cp and mv:

```
yk@peace:~/systemprj/fileio_test$ ls -i mytest
21977514 mytest
yk@peace:~/systemprj/fileio_test$ cp mytest mytest_cp
yk@peace:~/systemprj/fileio_test$ mv mytest mytest_mv
yk@peace:~/systemprj/fileio_test$ ls -il mytest*
21977512 -rw-rw-r-- 1 yk yk 0  8월  6 07:37 mytest_cp
21977514 -rw-rw-r-- 1 yk yk 0  8월  3 20:29 mytest_mv
```

Link Commands and inode

- **soft link (symbolic link)**: create new i-node, points to same data blocks
- **hard link** : share i-node with different name, increase the reference count of inode



Files in Linux

- Byte String
- Arbitrarily addressable
- contents has no predefined propertird
- protected by access rights
  - r
  - w
  - x
- defined for user, group, others

I-node (Index Node)

- Each file is represented by an inode
- contains
  - owber (UID, GID)
  - Access rights
  - Time of last accessed/modified
  - Size
  - Type (directory, file, deice, pipe, …)
  - pointers to data blocks that atores file’s content
  - No file name!

Directories

- Directories are handled as normal files, but are marked in inode-type as directory
- Directory entry contains
  - Length of the entry
  - Name (variable length up to 255 characters)
  - Inode number
- Multiple directory entries may reference the same Inode number (hard link)
- Users **identify files** via pathnames (".path/to/file") that are mapped to Inode numbers by OS
- If the path starts with "/", it is absolute and is resolved up from the root
- Otherwise the path is resolved relative to the current directory

Directories

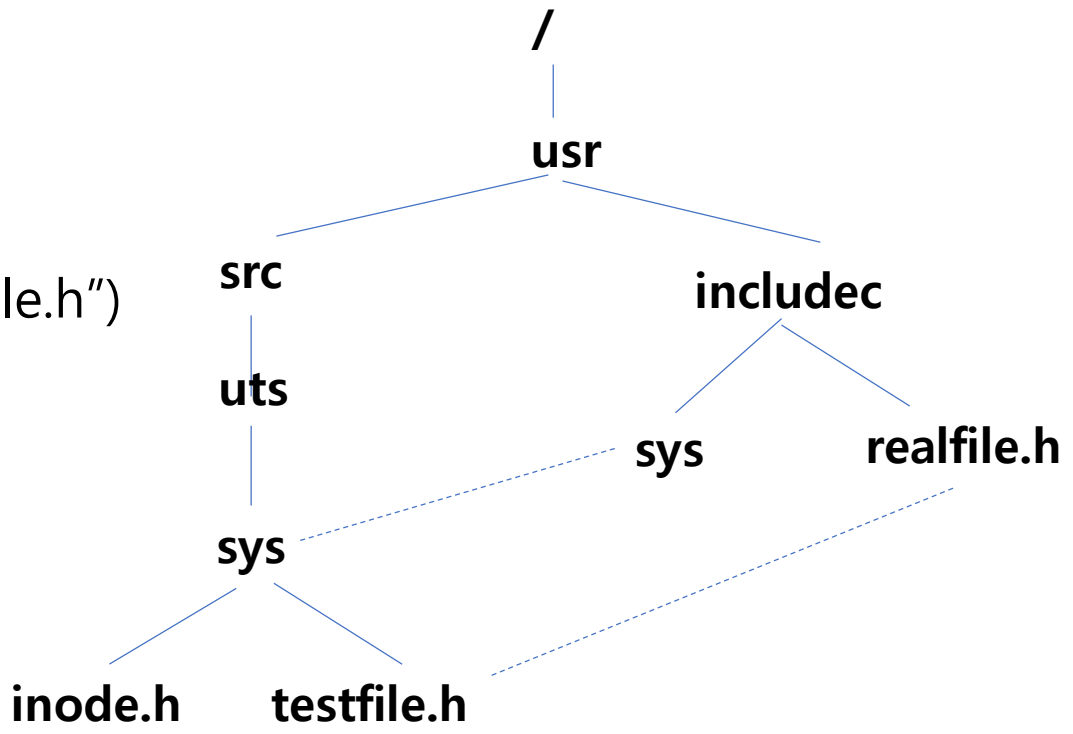
- Each directory contains an entry “.” that represents the Inode of the current directory
- The second entry “..” references parent directory
- The path is resolbed from left to right and the respective name is looked up in the directory

Symbolic Links (softlink)

- To improve shared access to files, UNIX allows use of symbolic links to reference single files and directories via multiple different paths
- symlink(existing_name, new_name) creates an additional path to the resource
- Example:

symlink("usr/src/uts/sys", "/usr/include/sys")

symlink("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h")

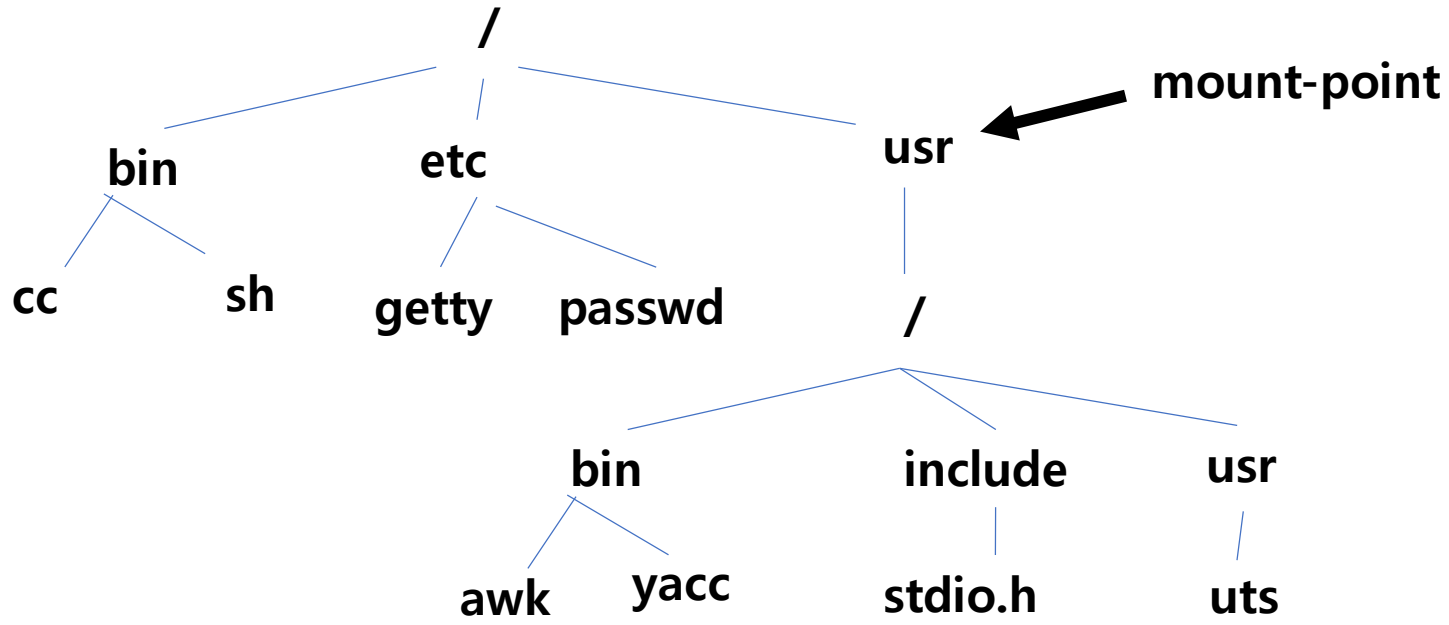


Hard and Symbolic Links

- A hard link is an additional file name
  - there exists another directory entry that points to the same file
  - All hard links point to the same inode
  - Each new hard link increments the link count of the Inode
  - As long as the count /= 0, the file "survives" a remove() and only link count is decremented
  - If last link is removed, the file is deleted and the Inode can be resused
- A Symbolic link (soft link)is a file that contains the path of another file/directory
  - Symbolic links are interpreted and resolved on every access
  - If target of a symbolic link is deleted, the link becomes invalid but remains existent

Logical and Physical System

- A logical File system may consists of multiple physical file systems
- A file system can be hooked into any path of the virtual file system tree with the “mount” command
- Mounted file systems are managed by the OS in a “mount table” that connects path to mount points
- This allows to identify the root Inode of mounted file systems

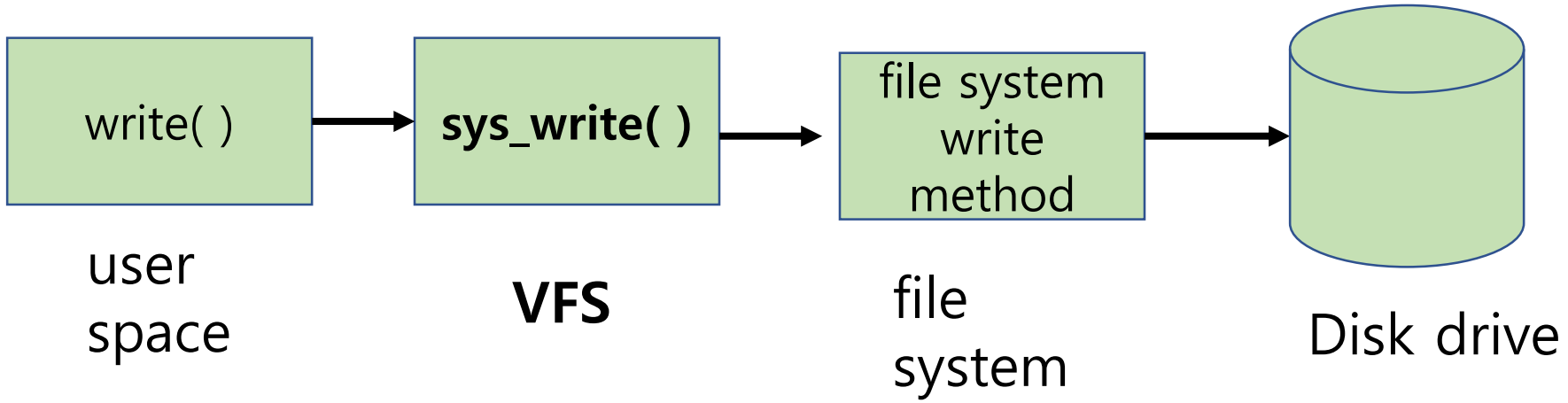


Virtual File System

- The **Virtual File System (VFS**) implements a generic file system interface between the actual file system implementation (in kernel) and accessing applications to provide interoperability

  ➔ Applications can access different file systems on different media via a homogeneous set of UNIX system calls

- Example write(f, &buf, len);
  - translated into system call
  - system call is forwarded to the actual file system implementation



VFS Objects and Data Structure

- VFS us object oriented
- Four base objects
  - Super block: represents specific properties of a filesystem
  - Inode: File description
  - Dentry: the directory entry represents a single component of a path
  - File: representation of an open file that is associated with a process
- VFS handles directories like files
  - Dentry object represents component of a path that may be a file
  - Directories are handled like files as Inode
- Each object provides a set of opeations

Superblock

- Each file system must provide a superblock
  - Contains properties of the file system
  - Is stored on special sectors of disk or is created dynamically
  - Structure is created by **`alloc_super()`** when the file system is mounted

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        dev_t                   s_dev;          /* search index; _not_
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_dirt;
        unsigned long long      s_maxbytes;     /* Max file size */
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        struct quotactl_ops     *s_qcop;
        struct export_operations *s_export_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        struct rw_semaphore     s_umount;
        struct mutex            s_lock;
        int                     s_count;
        int                     s_syncing;
        int                     s_need_sync_fs;
        atomic_t                s_active;
        void                    *s_security;
        struct xattr_handler    **s_xattr;

        struct list_head        s_inodes;       /* all inodes */
        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_io;           /* parked for writebac
        struct hlist_head       s_anon;         /* anonymous dentries
        struct list_head        s_files;

        struct block_device     *s_bdev;
        struct list_head        s_instances;
        struct quota_info       s_dquot;        /* Diskquota specific
```

I-node Object

- Contains information specific to a file
- For typical UNIX file systems, an inode can directly be read from disk
- Special Entries for non-data files
  - i_pipe, i_bdev, i_cdev are resereved for pipe, block device, charac device
- Some entries are not supported by all file systems and may therefore be set to Null

```
struct inode {
        struct hlist_node       i_hash;
        struct list_head        i_list;
        struct list_head        i_sb_list;
        struct list_head        i_dentry;
        unsigned long           i_ino;
        atomic_t                i_count;
        umode_t                 i_mode;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        dev_t                   i_rdev;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        unsigned int            i_blkbits;
        unsigned long           i_blksize;
        unsigned long           i_version;
        unsigned long           i_blocks;
        unsigned short          i_bytes;
        spinlock_t              i_lock;
        struct mutex            i_mutex;
        struct rw_semaphore     i_alloc_sem;
        struct inode_operations *i_op;
        struct file_operations  *i_fop;
        struct super_block      *i_sb;
        struct file_lock        *i_flock;
```

Dentry Object

- UNIX directories are handled like files
- The path /bin/vi contains the directory / and bin as well as file vi
- resolution of path requires introduction of dentry objects
- Each part of path is dentry object
- VFS creates dentry objects on the fly
- No equivalent on disk drive

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;           /* protected
        spinlock_t d_lock;              /* per dentry
        struct inode *d_inode;          /* Where the
                                         * negative *
        /*
         * The next three fields are touched by __d_l
         * so they all fit in a cache line.
         */
        struct hlist_node d_hash;       /* lookup has
        struct dentry *d_parent;        /* parent dir
        struct qstr d_name;

        struct list_head d_lru;         /* LRU list *
        /*
         * d_child and d_rcu can share memory
         */
        union {
                struct list_head d_child;       /* ch
                struct rcu_head d_rcu;
        } d_u;
        struct list_head d_subdirs;     /* our childr
        struct list_head d_alias;       /* inode alia
        unsigned long d_time;           /* used by d_
        struct dentry_operations *d_op;
        struct super_block *d_sb;       /* The root o
```

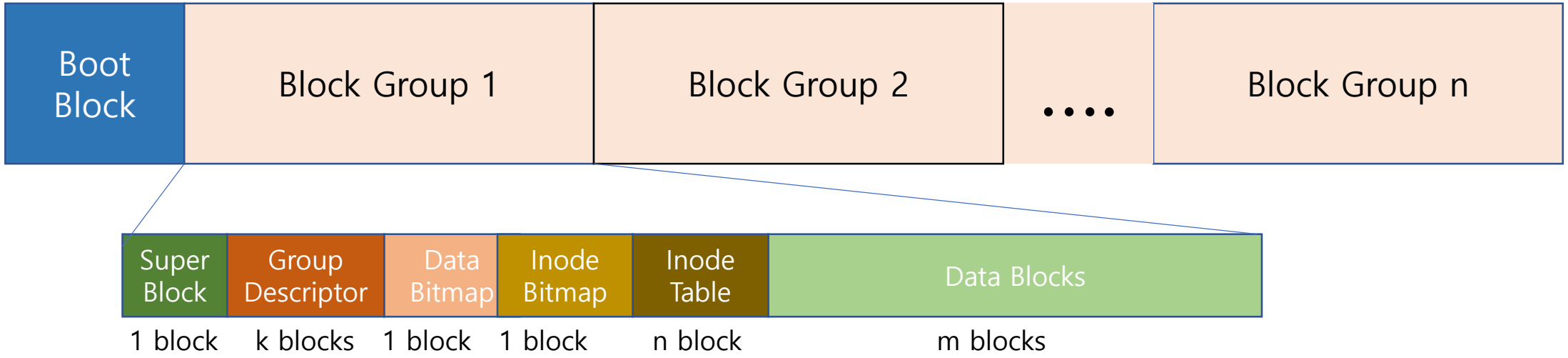
File Object

- File object represents open file
- Interface to applications
- is created as reply to open() system call
- is removed on close() system call
- different processes can open a file multiple times ➔ different file object
- The file object is an in-memory data structure of the OS

```
struct file {
        union {
                struct list_head        fu_list;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        struct file_ra_state    f_ra;
        unsigned long           f_version;
        void                    *f_security;
        void                    *private_data;
        struct list_head        f_ep_links;
        spinlock_t              f_ep_lock;
        struct address_space    *f_mapping;
};
```

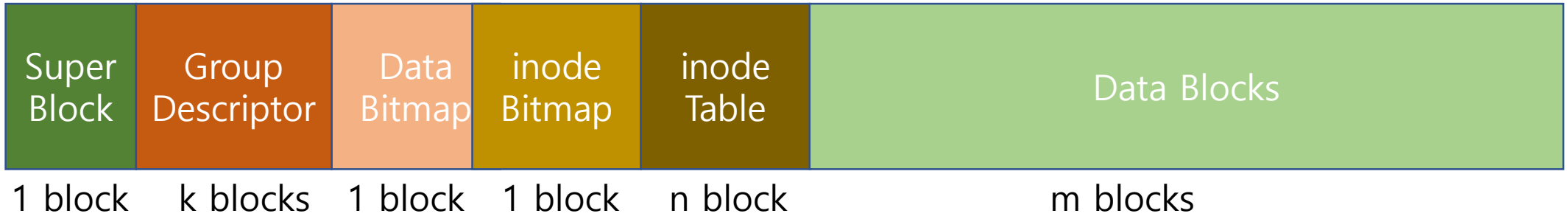
EXT2 Architecure

- EXT2 divides storage system into block groups



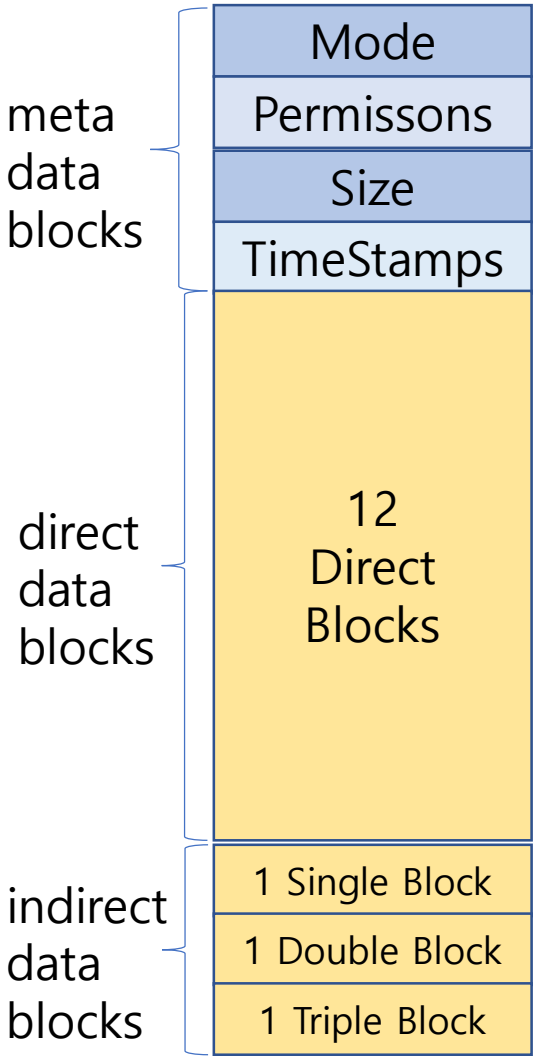
- Boot block is equivalent to first sector on Hard Disk.
- Block Group is basic component, which contains further filesystem components

Metadata for each Block Group



- Super Block: Central Structure, which contains number of free, allocated blocks, state of the filesystem, used block size, …
- Group Descriptor: contains the state, number of free blocks and inodes in each block group. Each block group has group descriptor
- Data Bitmap: 1/0 allocation representation for data block
- inode Bitmap: 1/0 allocation representation for inode blocks
- inode Table: stores all inodes for this block group
- Data Blocks: User data, File Contents

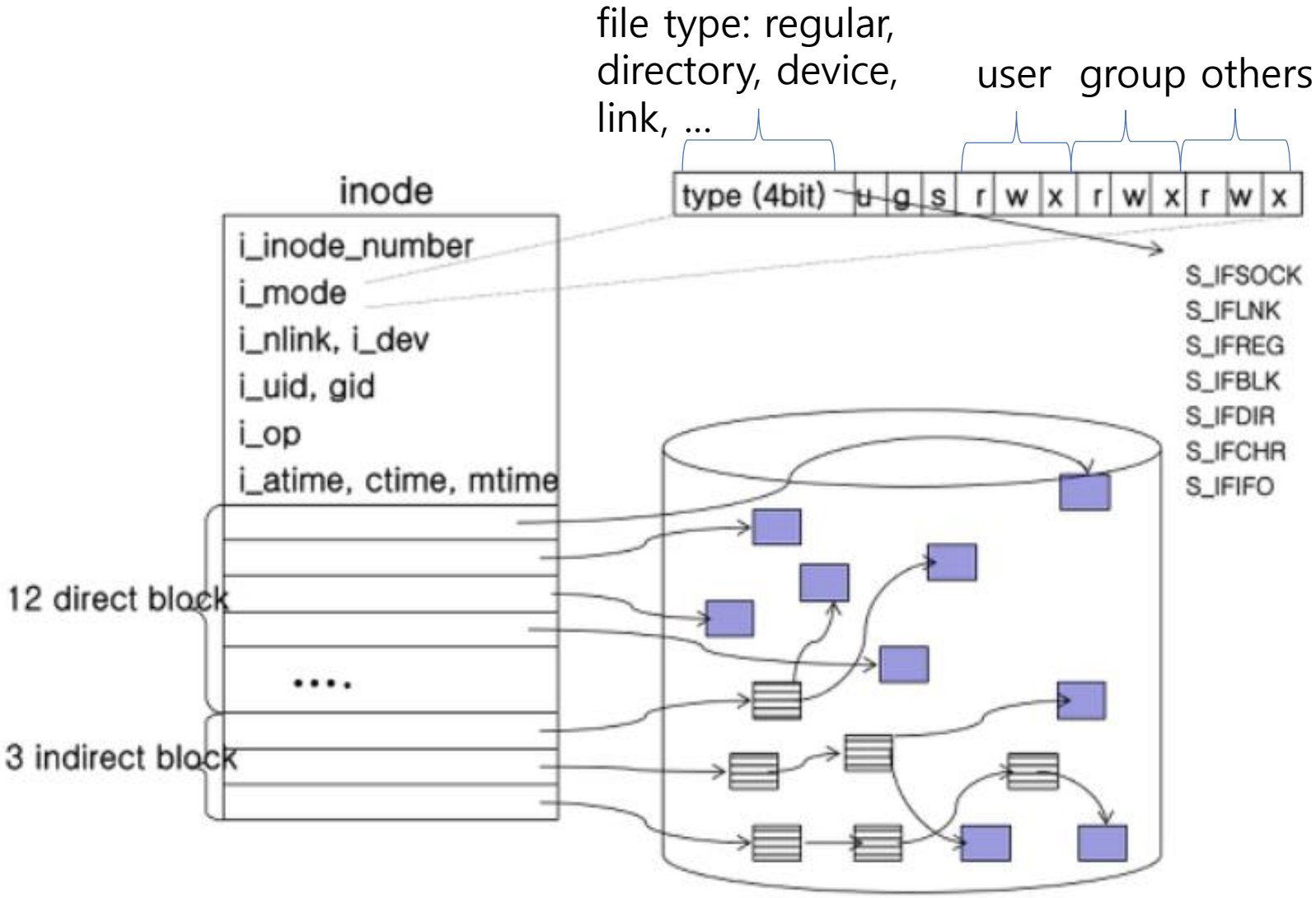
i-node Structure



- mode : ownership of user, group
- Permission: read, write, execute for each ownership
- Size: file size in bytes
- Time Stamps : creation, modification, recent access
- Type: file, directory, device, pipe,…

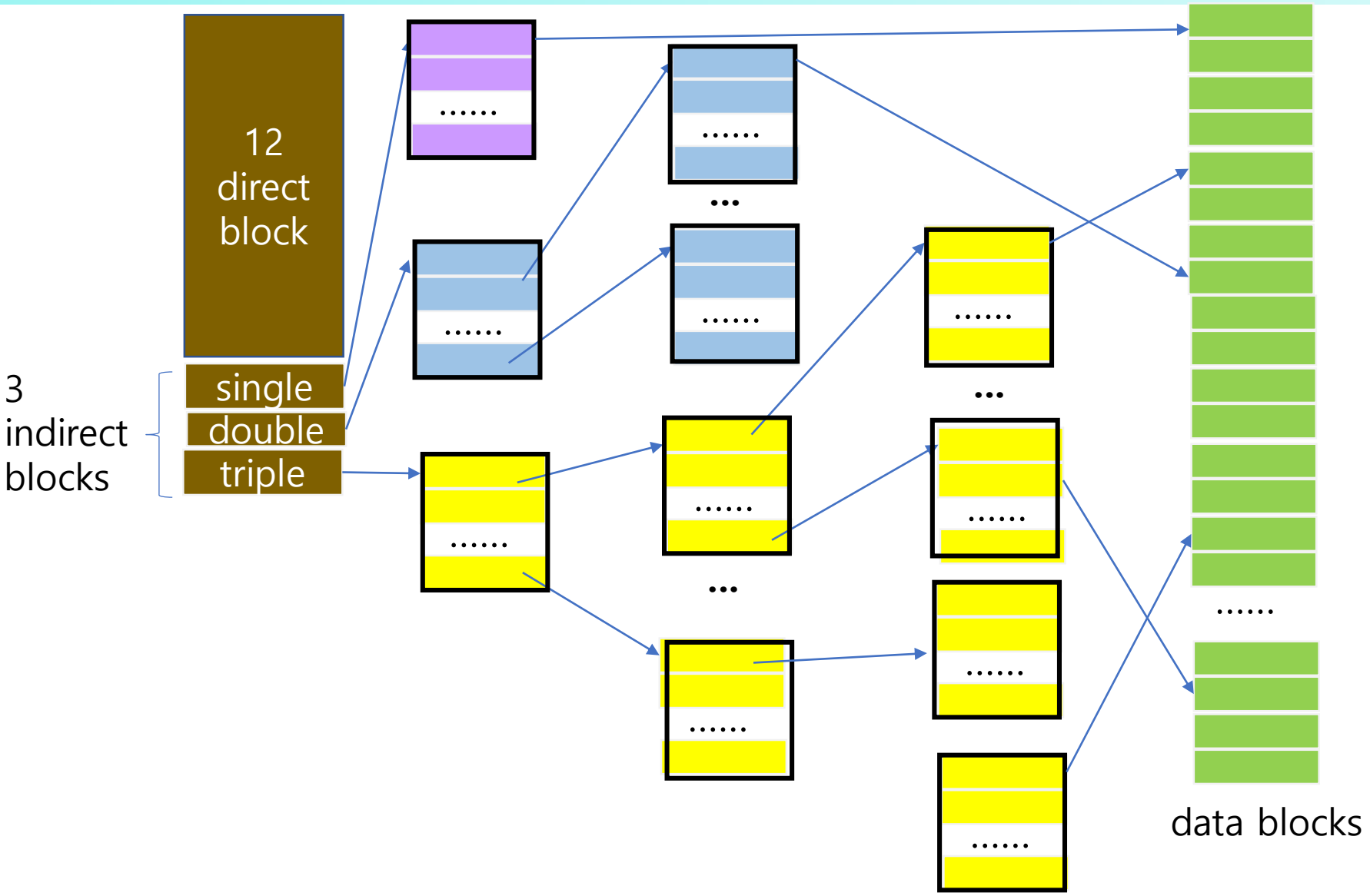
- Direct Blocks: pointer to file data block
- Indirect Blocks: Single, double, triple blocks to file data block

i-node metadata members



- i_mode (16bits): file type
- i_nlink: #of hard links
- i_uid, gid: user id, group id
- i_op : file operation functions
- i_atime, c_time, mtime: time of access, creation, modification
- i_size: file size in bytes
- i_blocks: #of data blocks allocated
- i_block[15] : pointers to data blocks

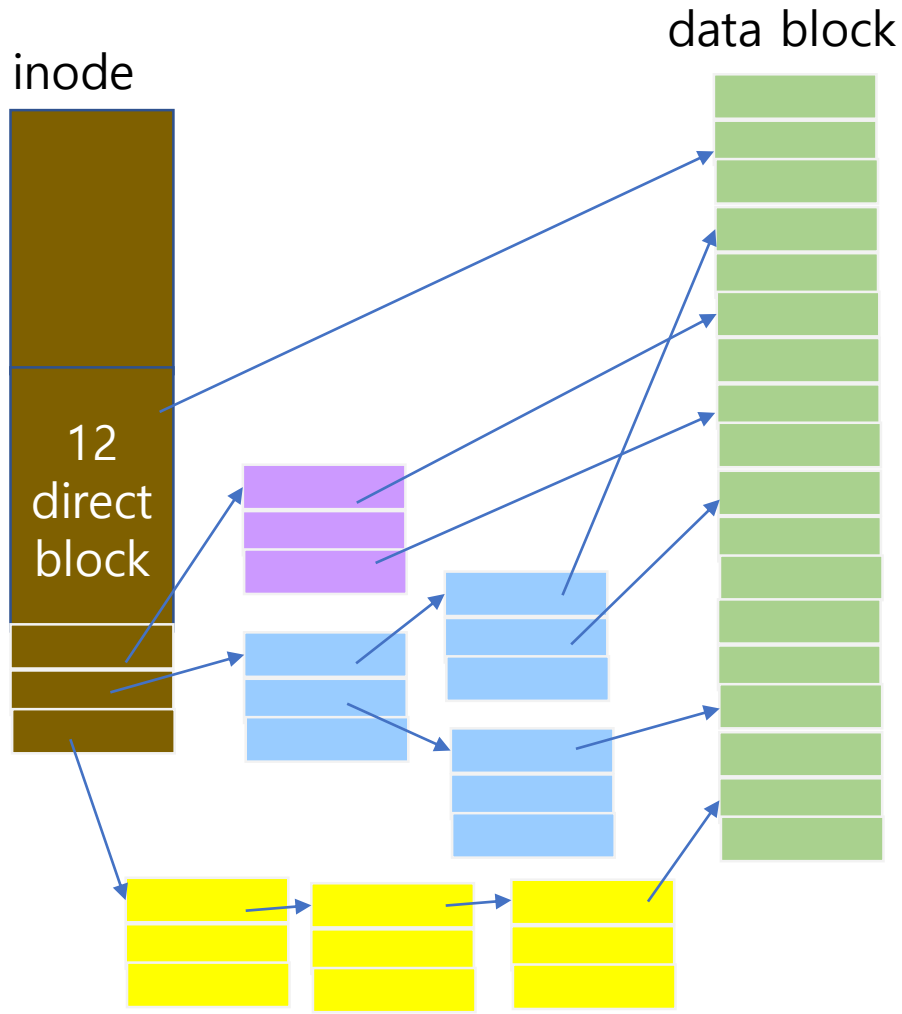
i-node Data blocks



- 12 direct pointers to data block
- 3 indirect pointers
  - 1 single indirect ptrs
  - 1 double indirect ptrs
  - 1 triple indirect ptrs

Limitation in File Size

- Size of a file is restricted by the number of block entries in inode
- Assumption:
  - 700 Mbyte file size and 4KB block size
  - 179,200 block entries (700x1024/4) are necessary and each entry needs 32bits(4 bytes/entry) ➔ 700KB
  - If the Inode size is fixed, then you also need 700 KByte inode for 4Kbytes files.
- EXT2 filesystem supports direct and indirect blocks
  - There is one pointer each for one-time, two-time, and three time indirect blocks



Calculation for Max File Size with Block Sizes

Blocksize (KByte)	1	2	**4**	8
Blocks first stage	12	12	**12**	12
Blocks first indirection	256	512	**1,024**	2,048
Blocks second indirection	65,536	262,144	**1,048,576**	4,194,304
Blocks third indirection	16,777,216	134,217,728	**1,073,741,824**	8,589,934,592
maximum number of entry presented	16,843,020	134,480,396	**1,074,791,436**	8,594,130,956
maximum file size(Gbytes)	16	257	**4,100**	65,568

- Assumption: 4 bytes / Entry
- Increasing the block size increases maximum file size quadratically
- With 4KB/block, upto 4.1 TB file size is possible
- With 8KB/block, upto 65TB file size is possible

Exercise and Problems for Filesystem

Quiz #1: **Removing consecutive repeated characters from string**

- This program will read a line from a file and **remove repeated consecutive characters from the string** and print out the updated line. In this program we will not use another string to copy string after removing the characters; we repeat the updating process line by line manner until all the file is read and print out the result.

```
First Run:
Enter any string: AABBCCDD
String after removing characaters: ABCD

Second Run:
Enter any string: AAAABBBBCCCCDDDD
String after removing characaters: ABCD

Third Run:
Enter any string: AAA AAA BBB CCCC DDDD
String after removing characaters: A A B C D
```

Hints to Quiz#1: Read a string and remove repeated characters.

```
#include <stdio.h>

int main()
{
    char str[100];
    int i,j,len,len1;

    /*read string*/
    printf("Enter any string: ");
    gets(str);

    /*calculating length*/
    for(len=0; str[len]!='₩0'; len++)
            ;
    /*assign 0 to len1 - length of removed characters*/
    len1=0;
    /*Removing consecutive repeated characters from string*/
    for(i=0; i<(len-len1);)  {
        if(str[i]==str[i+1])   {
            /*shift all characters*/
            for(j=i;j<(len-len1);j++)
                str[j]=str[j+1];
            len1++;
        }  else {
            i++;
        }
    }
    printf("String after removing characaters: %s₩n",str);
    return 0;
}
```

Quiz #2,#3: Read/Write Struct From/to a File

- **Q2: Writing Structure to a File using fwrite**

We can use fwrite() function to easily write a *struct person data* in a file. fwrite() function writes the data to the file stream *in the form of binary data block*.

- **Q3: Reading Structure from a File using fread**

We can use fread() function to easily read a *struct person data* stored in a file. This function reads a block of memory from the given stream.

```
struct person {
    int id;
    char fname[20];
    char lname[20];
};
```

Answer to Quiz#2

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

// a struct to be read and written
struct person {
    int id;
    char fname[20];
    char lname[20];
};

int main() {
    FILE* outfile;

    // open file for writing
    outfile = fopen("person.bin", "wb");
    if (outfile == NULL) {
        fprintf(stderr, "₩nError opened file₩n");
        exit(1);
    }
    struct person input1 = { 1, "rohan", "sharma" };

    // write struct to file
    int flag = 0;
    flag = fwrite(&input1, sizeof(struct person), 1,outfile);
    if (flag) {
        printf("Contents of the structure written "
               "successfully");
    }
    else
        printf("Error Writing to File!");

    // close file
    fclose(outfile);

    return 0;
}
```

Answer to Quiz #3

```
// struct from a file
#include <stdio.h>
#include <stdlib.h>

// struct person with 3 fields
struct person {
    int id;
    char fname[20];
    char lname[20];
};

int main( )
{
    FILE* infile;
    // Open person.dat for reading
   if( (infile = fopen("person1.dat", "r")) == NULL) {
        fprintf(stderr, "₩nError opening file₩n");
        exit(1);
    }
```

```
   // reading to read_struct
    fread(&read_struct, sizeof(read_struct), 1, infile);

    printf("Name: %s %s ₩nID: %d", read_struct.fnam  read_struct.lname, read_struct.id);

    // close file
    fclose(infile);

    return 0;
}
```

Quiz #4 Copying a file contents to another file

- Usage of the program in command line

  <program-name> <source-file-name> <destination-file-name>

  ex) copy dataSrc.1 dataDest.1
  - copy: a executable file name
  - dataSrc.1 : existing file name (binary or text whatever format)
  - dataDest.1: new file name to be generated with the contents of dataSrc.1
- We can use fgetc() function and fputc() function to read and write a byte from a file, respectively.

Answer to Quiz#4

```
#include <stdio.h>
#include <stdlib.h> // For exit()

int main(int argc ,char **argv)
{
    FILE *fptr1 = stdin, *fptr2 = stdout;
    int c;

    if(argc < 3) {
        printf("Usage: %s <src> <dest>₩n", argv[0]);
        exit(1);
   }
    // Open one file for reading
    fptr1 = fopen (argv[1], "r");
    if (fptr1 == NULL) {
        printf("Cannot open file %s₩n",argv[1]);
        exit(1);
    }
```

```
    // Open another file for writing
    fptr2 = fopen(argv[2], "w");
    if (fptr2 == NULL)   {
        printf("Cannot open file %s₩n", argv[2]);
        exit(1);
    }

    // Read contents from file
    while ((c = fgetc(fptr1)) != EOF)
    {
        fputc(c, fptr2);
    }
    printf("Contents copied to %s₩n", filename);
    fclose(fptr1);
    fclose(fptr2);
    return 0;
}
```

Quiz#5 **Append Content of one text file to another**

- Given the source and destination text files, the task is to append the content from source file to destination file and then display the content of the destination file.
- **Approach:**
  1. Open **file1.txt** and **file2.txt** with “a+”(append and read) option, so that the previous content of the file is not deleted. If files don’t exist, they will be created.
  2. Explicitly write a newline (“₩n”) to the destination file to enhance readability.
  3. Write content from source file to destination file.
  4. Display the contents in file2.txt to console (stdout).

Input:

file1.text
This is line one in file1
Hello World.
file2.text
This is line one in file2
Programming is fun.

Output:

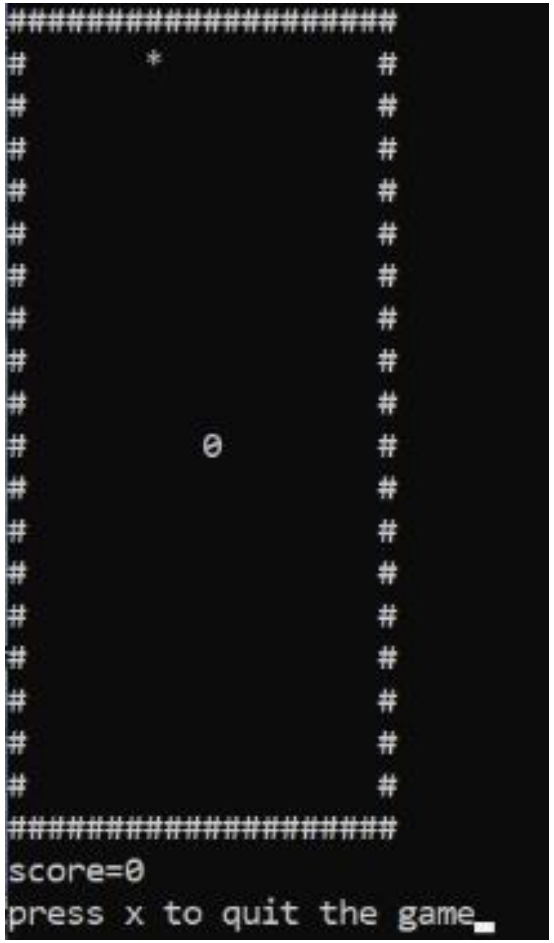
This is line one in file2
Programming is fun.
This is line one in file1
Hello World.

Answer to Quiz#5

```
#include <stdio.h>
// Function that appends the contents
void appendFiles(char source[],
                 char destination[])
{
    // declaring file pointers
    FILE *fp1, *fp2;

    // opening files
    fp1 = fopen(source, "a+");
    fp2 = fopen(destination, "a+");

    // If file is not found then return.
    if (!fp1 && !fp2) {
        printf("Unable to open/"
               "detect file(s)₩n");
        return;
    }
    char buf[100];

    // explicitly writing "₩n"
    // to the destination file
    // so to enhance readability.
    fprintf(fp2, "₩n");

    // writing the contents of
    // source file to destination file.
    while (!feof(fp1)) {
        fgets(buf, sizeof(buf), fp1);
        fprintf(fp2, "%s", buf);
    }
// printing the results to stdout
    rewind(fp2);
    while (!feof(fp2)) {
        fgets(buf, sizeof(buf), fp2);
        printf("%s", buf);
    }
}

int main()
{
    char source[] = "file1.txt",
         destination[] = "file2.tx

    // calling Function with file names.
    appendFiles(source, destination);
    return 0;
}
```

Problem #1: Snake Game in C

- Below given some functionalities of this game:
  - The snake is represented with a **0**(zero) symbol.
  - The fruit is represented with an ***(asterisk) symbol.
  - The snake can move in any direction according to the user with the help of the keyboard (**W**, **A**, **S**, **D** keys).
  - When the snake eats a fruit the score will increase by 10 points.
  - The fruit will generate automatically randomly within the boundaries.
  - Whenever the snake will touch the boundary the game is over.



Ans(1). to Prob-1

```
// C program to build the
complete
// snake game
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int i, j, height = 20, width = 20;
int gameover, score;
int x, y, fruitx, fruity, flag;
```

```
void main()
{
    int m, n;
    setup();
    while (!gameover) {
        // Function Call
        draw();
        input();
        logic();
    }
}
```

```
// Function to generate the fruit
// within boundary
void setup()
{
    gameover = 0;

    // Stores height and width
    x = height / 2;
    y = width / 2;
label1:
    fruitx = rand() % 20;
    if (fruitx == 0)
        goto label1;
label2:
    fruity = rand() % 20;
    if (fruity == 0)
        goto label2;
    score = 0;
}
```

```
// Function to draw the boundaries
void draw()  {
  system("cls");
  for (i = 0; i < height; i++) {
    for (j = 0; j < width; j++) {
      if (i == 0 || i == width - 1 || j == 0
          || j == height - 1) {
          printf("#");
      }  else {
          if (i == x && j == y)
              printf("0");
          else if (i == fruitx && j == fruity)
              printf("*");
           else
              printf(" ");
      }
    }
    printf("₩n");
  }
  // Print the score after the
  // game ends
  printf("score = %d", score);
  printf("₩n");
  printf("press X to quit the game");
}
```

Ans(2) to Prob-1

```
// Function to take the input

void input() {
    if (kbhit()) {
        switch (getch()) {
        case 'a':
            flag = 1;
            break;
        case 's':
            flag = 2;
            break;
        case 'd':
            flag = 3;
            break;
        case 'w':
            flag = 4;
            break;
        case 'x':
            gameover = 1;
            break;
        }
    }
} // end of input function
```

```
// Function for the logic
// behind each movement
void logic()
{
    sleep(0.01);
    switch (flag) {
    case 1:
        y--;
        break;
    case 2:
        x++;
        break;
    case 3:
        y++;
        break;
    case 4:
        x--;
        break;
    default:
        break;
    }
```

```
    // If the game is over
    if (x < 0 || x > height
        || y < 0 || y > width)
        gameover = 1;

    // If snake reaches the fruit
    // then update the score
    if (x == fruitx && y == fruity) {
    label3:
        fruitx = rand() % 20;
        if (fruitx == 0)
            goto label3;

    // After eating the above fruit
    // generate new fruit
    label4:
        fruity = rand() % 20;
        if (fruity == 0)
            goto label4;
        score += 10;
    }
} // end of logic function
```

Problem-2: BMP file

- **Problem**

1. **Dump BMP file header information: width, rows, and file size**
2. **Dump the BMP file data as hexadecimal as xxd command do**
3. **Change the BMP file to make it grayscale**
   - **Change Red and Blue value as the same value of Green for all pixels**
4. **Encrypt a message in an BMP image (Steganography)**
   - **Use LSB bit of each pixel's blue and red color to store information**

BMP file format

- BMP file format : 54 bytes header + image data
- Each pixel consists of Blue, Green, and Red color (8bits * 3 = 24bits)
  - each pixel data : B G R
- Each row data should be multiple of 4 byes (padding is required for each row data to adjust multiple of 4 byte size per row)

BMP file format

Component	Size
Header	54 byte
Palette (optional)	0 byte (for 24-bit RGB image)
Image Data	file size – 54 (for 24-bit RGB image)

- header size : 54 bytes fixed
- header begins with a magic#: **0x4d42** (16bits)
- important header fields:
  - size: file size in byte
  - width_px: number of pixels per row
  - height_px: number of rows
  - bits_per_pixel: number of bits per pixel
    - 24 for BGR colors (8bits * 3 colors)
  - image_size_bytes : the size of image data
    - file size – header size

<BMP Header>

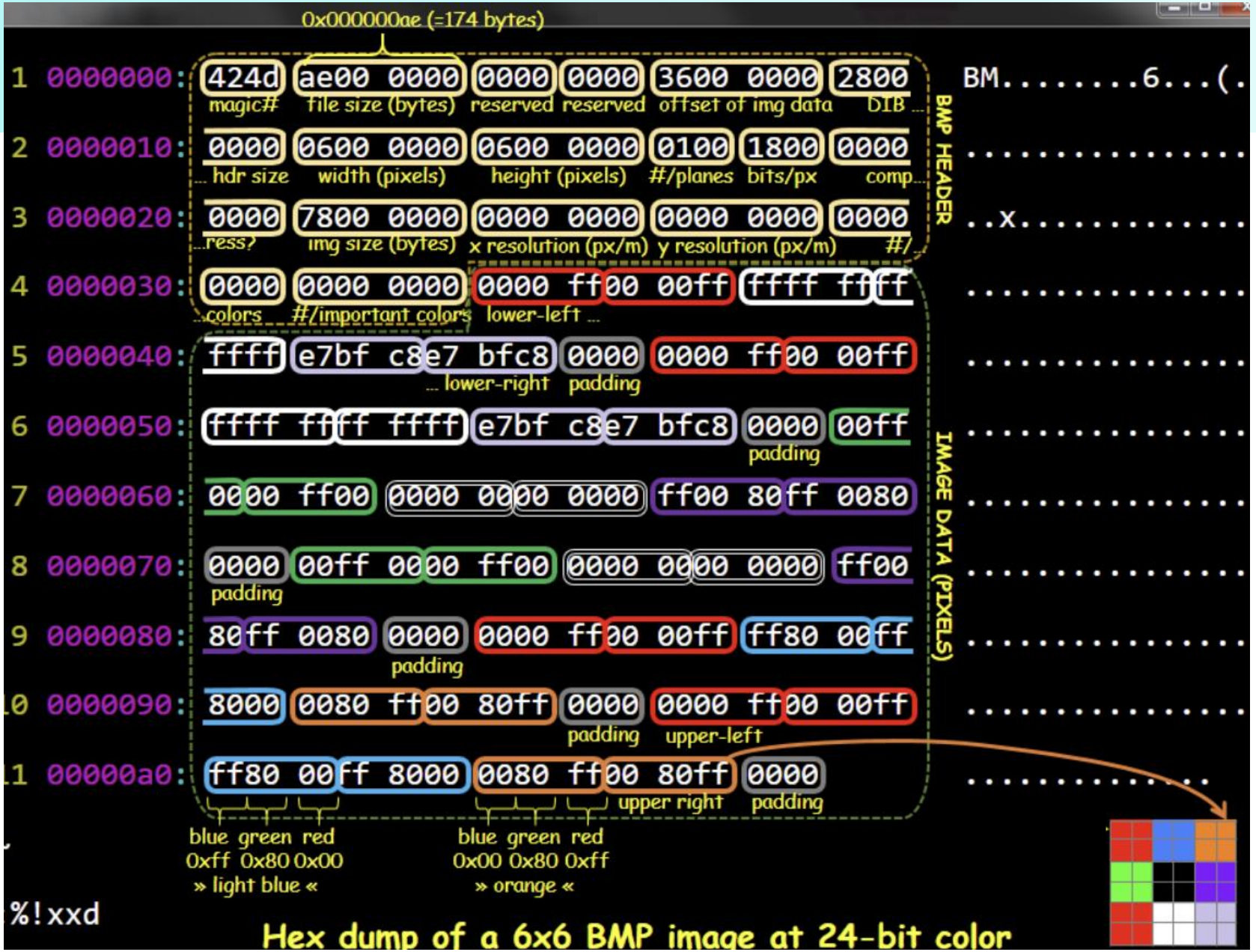
```
typedef struct {
  uint16_t  type;
  uint32_t  size;
  uint16_t  reserved1;
  uint16_t  reserved2;
  uint32_t  offset;
  uint32_t  dib_header_size;
  int32_t   width_px;
  int32_t   height_px;
  uint16_t  num_planes;
  uint16_t  bits_per_pixel;
  uint32_t  compression;
  uint32_t  image_size_bytes;
  int32_t   x_resolution_ppm;
  int32_t   y_resolution_ppm;
  uint32_t  num_colors;
  uint32_t  important_colors;
} BMPHeader;
```

BMP format

$ xxd <BMP_file>

- header size = 54 bytes
- image size = 6 pixels x 6 pixels = **36 pixels size**
- Each row = 6 * 3 bytes = 18 bytes → **20 bytes** (for 4 bytes alignment)
- image data size = 20 bytes/row * 6 = **120 bytes**
- file size = header size + image data

  = 54 + 120 = **174**



Endianness of BMP file: Little Endian

- little-endian: LSB comes first (Intel x86, x86-64)
- Big-endian : MSB comes first (IBM 360, Motorola 68000)

```
#include <stdio.h>
// testing endianness with 4 byte value
int main()
{
  FILE *fp = fopen("data","w+");

  unsigned int data = 0x12345678;
  fwrite(&data, sizeof(data), 1, fp);

  fclose(fp);
  return 0;
}
```

```
yk@peace:~/systemprj/file$ xxd data
00000000: 7856 3412
```

<little-endian order>

03:	78
02:	56
01:	34
00:	12

BMP file is little-endian format

05:	00
04:	00
03:	00
02:	ae
01:	4d
00:	42

file size → 00 00 00 ae

magic# → 4d 42

- magic#: 0x4d42
- file size: 0x000000ae
  = A*16+E = 174

bmp.h

#pragma pack(1) directive ensures that the header structure is really 54-byte long by using 1-byte alignment.

#pragma pack(1)

```
typedef struct {              // Total: 54 bytes
  uint16_t  type;             // Magic identifier: 0x4d42
  uint32_t  size;             // File size in bytes
  uint16_t  reserved1;        // Not used
  uint16_t  reserved2;        // Not used
  uint32_t  offset;           // Offset to image data in bytes from beginning of file (54)
  uint32_t  dib_header_size;  // DIB Header size in bytes (40 bytes)
  int32_t   width_px;         // Width of the image
  int32_t   height_px;        // Height of image
  uint16_t  num_planes;       // Number of color planes
  uint16_t  bits_per_pixel;   // Bits per pixel
  uint32_t  compression;      // Compression type
  uint32_t  image_size_bytes; // Image size in bytes
  int32_t   x_resolution_ppm; // Pixels per meter
  int32_t   y_resolution_ppm; // Pixels per meter
  uint32_t  num_colors;       // Number of colors
  uint32_t  important_colors; // Important colors
} BMPHeader;

 typedef struct {
   BMPHeader header;
   unsigned char* data;
 } BMPImage;
```

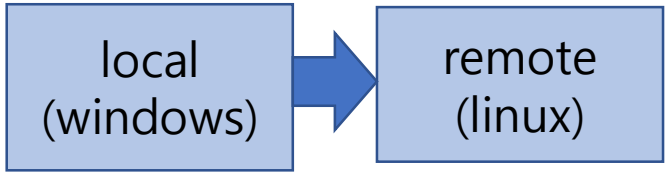
Hint to Prob-2: The Skeleton Code of bmp.c

```
#include <stdio.h>
#include <stdlib.h>
#include "bmp.h"

int main (int argc, char *argv[])
{
    BMPImage  bmp_img;
    FILE *fp1 = fopen(argv[1], "r");
    FILE *fp2 = fopen(agrv[2], "w");

    read_header(fp1, &(bmp_img.header));
    bmp_img.data = read_data (fp1, bmp_img);
    change_color_grayscale(bmp_img.data, bmp_hd.width_px, bmp_hd.height_px, bmp_hd.);
    write_data(fp2, &bmp_img);
    return (1);
}
```

How to transfer files to/from remote host

- local에서 원격 host로 이미지 파일 보내기(scp)
  - **scp** *<source-file> <user_id>@<remot-host>:<target-file>*

```
C:\Users\yk\Pictures>scp khj.jpg yk@peace.handong.edu:image.jpg
yk@peace.handong.edu's password:
khj.jpg

C:\Users\yk\Pictures>
```



- 원격 host에서 local로 이미지 파일 가져오기(scp)
  - **scp** *<user_id>@<remot-host>:<source_file> <target-file>*

```
C:\Users\yk\Pictures>scp yk@peace.handong.edu:image.jpg k.jpg
yk@peace.handong.edu's password:
image.jpg
```